Updating the Java Grande Forum Benchmark Suite

Peter Sanderson

August 19, 2011

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2011
Abstract

Java has never offered performance to match that of traditional High Performance Computing (HPC) programming languages. This was confirmed by the Java Grande Forum at a time when great interest was being shown in Java for Grande applications; codes that require large amounts of processing, memory, I/O, bandwidth, or a combination of these properties. The Java Grande Forum Benchmark Suite (JGFBS) was developed to measure the performance of Java in a given execution environment when executing so-called Grande applications. The result was conclusive: Java was not, at the time, a viable option for the development and execution of Grande codes. There are numerous inherent benefits within Java that are not present in traditional HPC programming languages, but performance is key and whilst ever it is lacking Java will not be used by the HPC world. However, Java is continually updating and improving. The release of Java 5 has introduced a large amount of concurrent functionality into Java that was not present at the time of the initial release of the JGFBS. Updating the parallel component of the benchmark suite to incorporate new concurrent functionality and benchmarking modern execution environments to measure Java’s improvement may lead to renewed interest and investment in the use of Java for Grande applications.
Contents

Chapter 1 Introduction .................................................................................................................. 1

1.1 Previous Research ................................................................................................................. 3

Chapter 2 Goals, Methodology and Framework ........................................................................... 5

2.1 Primary Goals ....................................................................................................................... 5

2.2 Secondary Goals ................................................................................................................. 5

2.3 Goals Summary .................................................................................................................... 6

Chapter 3 The Existing Parallel Component of the Java Grande Forum Benchmark Suite ........... 8

3.1 Section I ................................................................................................................................ 8

3.1.1 Barrier Benchmark (JGFBarrierBench) ............................................................................. 9

3.1.2 Fork-Join Benchmark (JGFForkJoinBench) ...................................................................... 9

3.1.3 Synchronisation Benchmark (JGFSyncBench) ............................................................... 9

3.2 Section II .............................................................................................................................. 9

3.2.1 Crypt Benchmark (JGFCryptBenchSize\{A,B,C\}).......................................................... 9

3.2.2 LUFact Benchmark (JGFLUFactBenchSize\{A,B,C\})..................................................... 9

3.2.3 Series Benchmark (JGFSeriesBenchSize\{A,B,C\})....................................................... 10

3.2.4 SOR Benchmark (JGFSORBenchSize\{A,B,C\}).......................................................... 10

3.2.5 SparseMatMult Benchmark (JGFSparseMatMultBenchSize\{A,B,C\})................. 10

3.3 Section III ........................................................................................................................... 10
3.3.1 Molecular Dynamics Simulation Benchmark (JGFMolDynBenchSize[{A,B}]) .......................................................... 10

3.3.2 Monte Carlo Benchmark (JGFMonteCarloBenchSize[{A,B}]) .......... 10

3.3.3 Ray Tracer Benchmark (JGFRayTracerBenchSize[{A,B}]) .......... 10

3.4 Invoking Parallelism .................................................................. 10

Chapter 4 The Updated Parallel Component of the Java Grande Forum Benchmark Suite ................................................................................. 12

4.1 Concurrency Package (java.util.concurrent) .................................. 13

4.1.1 CyclicBarrier (java.util.concurrent.CyclicBarrier) .................. 13

4.1.2 Thread Pools ........................................................................... 14

4.1.3 Atomic Objects (java.util.concurrent.atomic) .......................... 14

4.2 Utility Classes ............................................................................ 14

4.2.1 JGFInstrumentor .................................................................... 14

4.2.2 JGFTimer .................................................................................. 15

4.2.3 JGFParser ................................................................................. 15

4.3 Section I ....................................................................................... 16

4.3.1 Driver (JGFAll) ........................................................................ 16

4.3.2 Barrier Benchmark (JGFBenchmark) ....................................... 16

4.3.3 Fork-Join (JGForkJoinBench) and Synchronisation (JGFSyncBench) ...... 16

4.3.4 Atomic Benchmark (JGFAtomBench) ......................................... 16

4.3.5 Thread Pool Benchmark (JGFTThreadPoolBench) ....................... 16

4.4 Section II ..................................................................................... 17

4.4.1 LU Factorisation Benchmark (JGFLUFactBenchSize[{A,B,C}]) ....... 17

4.4.2 SOR Benchmark (JGFSORBenchSize[{A,B,C}]) ......................... 17

4.5 Section III ................................................................................... 17
4.5.1 Molecular Dynamics Simulation Benchmark (JGF MolDynBenchSize[{A,B}]) ................................................................. 17
4.5.2 Monte Carlo Benchmark (JGF MonteCarloBenchSize[{A,B}]) ............... 17
4.5.3 Ray Tracer Benchmark (JGF RayTracerBenchSize[{A,B}]) ...................... 18
4.6 Testing and Verification of Updates ...................................................... 18

Chapter 5 Results and Analysis .................................................................. 19
5.1 JVM Comparisons ................................................................................. 20
  5.1.1 Section I ......................................................................................... 20
  5.1.2 Section II ....................................................................................... 23
  5.1.3 Section III ...................................................................................... 28
  5.1.4 JVM Comparisons Summary ............................................................ 30
5.2 Ness: Benchmarking the Updated Parallel Component ......................... 31
  5.2.1 New Benchmarks ........................................................................... 36
  5.2.2 Ness Summary ............................................................................... 38
5.3 Eddie: Benchmarking the Updated Parallel Component ......................... 38
  5.3.1 New Benchmarks ........................................................................... 43
  5.3.2 PE16 versus Ness ........................................................................... 45
  5.3.3 PE16 versus PE12 ........................................................................... 45
5.4 Language Comparisons ......................................................................... 48
  5.4.1 Ness .............................................................................................. 48
  5.4.2 Eddie ............................................................................................. 49
  5.4.3 Language Comparisons Summary .................................................... 50
5.5 Sequential versus Threaded .................................................................... 50
  5.5.1 Summary ....................................................................................... 52
<table>
<thead>
<tr>
<th>Chapter 6 Conclusions</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 7 Future Work</td>
<td>55</td>
</tr>
<tr>
<td>Appendix A Additional Figures</td>
<td>57</td>
</tr>
<tr>
<td>References</td>
<td>68</td>
</tr>
</tbody>
</table>
List of Figures

Figure 5.1: Performances of Java environments when executing Synchronisation benchmark with data set A .................................................................22
Figure 5.2: Performances of Java environments when executing Barrier benchmark with data set A .................................................................22
Figure 5.3: Performances of Java environments when executing Fork-Join benchmark with data set A .................................................................23
Figure 5.4: Performances of Java environments when executing Series benchmark with data set A .................................................................25
Figure 5.5: Performances of Java environments when executing LU Factorisation benchmark with data set A .........................................................26
Figure 5.6: Performances of Java environments when executing Crypt benchmark with data set A .................................................................26
Figure 5.7: Performances of Java environments when executing SOR benchmark with data set A .................................................................27
Figure 5.8: Performances of Java environments when executing SparseMatMult benchmark with data set A .........................................................27
Figure 5.9: Performances of Java environments when executing Molecular Dynamic benchmark with data set A .........................................................29
Figure 5.10: Performances of Java environments when executing Monte Carlo benchmark with data set A .........................................................29
Figure 5.11: Performances of Java environments when executing Ray Tracer benchmark with data set A .........................................................30
Figure 5.12: Performances of the original and updated Barrier benchmarks on Ness ...33
Figure 5.13: Performances of the original and updated LU Factorisation benchmarks on Ness .................................................................33
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.14</td>
<td>Performances of the original and updated SOR benchmarks on Ness</td>
<td>34</td>
</tr>
<tr>
<td>5.15</td>
<td>Performances of the original and updated Molecular Dynamic benchmarks on Ness</td>
<td>34</td>
</tr>
<tr>
<td>5.16</td>
<td>Performances of the original and updated Monte Carlo benchmarks on Ness</td>
<td>35</td>
</tr>
<tr>
<td>5.17</td>
<td>Performances of the original and updated Ray Tracer benchmarks on Ness</td>
<td>35</td>
</tr>
<tr>
<td>5.18</td>
<td>Performance comparison of the Thread Pool and Fork-Join benchmarks on Ness</td>
<td>37</td>
</tr>
<tr>
<td>5.19</td>
<td>Performance comparison of the Atomic and Synchronisation benchmarks on Ness</td>
<td>37</td>
</tr>
<tr>
<td>5.20</td>
<td>Performances of the original and updated Barrier benchmarks on Eddie (PE16)</td>
<td>40</td>
</tr>
<tr>
<td>5.21</td>
<td>Performances of the original and updated LU Factorisation benchmarks on Eddie (PE16)</td>
<td>41</td>
</tr>
<tr>
<td>5.22</td>
<td>Performances of the original and updated SOR benchmarks on Eddie (PE16)</td>
<td>41</td>
</tr>
<tr>
<td>5.23</td>
<td>Performances of the original and updated Molecular Dynamics benchmarks on Eddie (PE16)</td>
<td>42</td>
</tr>
<tr>
<td>5.24</td>
<td>Performances of the original and updated Monte Carlo benchmarks on Eddie (PE16)</td>
<td>42</td>
</tr>
<tr>
<td>5.25</td>
<td>Performances of the original and updated Ray Tracer benchmarks on Eddie (PE16)</td>
<td>43</td>
</tr>
<tr>
<td>5.26</td>
<td>Performance comparison of the Thread Pool and Fork-Join benchmarks on Eddie (PE16)</td>
<td>44</td>
</tr>
<tr>
<td>5.27</td>
<td>Performance comparison of the Atomic and Synchronisation benchmarks on Eddie (PE16)</td>
<td>44</td>
</tr>
<tr>
<td>5.28</td>
<td>Performance comparison of the different execution environments on Eddie for the Synchronisation and Atomic benchmarks</td>
<td>46</td>
</tr>
</tbody>
</table>
Figure A5.8.2: Performances of Java environments when executing SparseMatMult benchmark with data set C .................................................................62

Figure A5.9: Performances of Java environments when executing Molecular Dynamic benchmark with data set B ..................................................................................................................62

Figure A5.10: Performances of Java environments when executing Monte Carlo benchmark with data set B ..................................................................................................................63

Figure A5.11: Performances of Java environments when executing Ray Tracer benchmark with data set B ..................................................................................................................63

Figure A28.1: Performances of different execution environments on Eddie when executing Barrier benchmarks ..................................................................................................................64

Figure A28.2: Performances of different execution environments on Eddie when executing Thread Pool and Fork-Join benchmarks ..................................................................................................64

Figure A29: Performances of different execution environments on Eddie when executing LU Factorisation benchmarks ...........................................................................................................65

Figure A30: Performances of different execution environments on Eddie when executing SOR benchmarks ......................................................................................................................65

Figure A31: Performances of different execution environments on Eddie when executing SparseMatMult benchmarks ..............................................................................................................66

Figure A32: Performances of different execution environments on Eddie when executing Molecular Dynamic benchmarks ...........................................................................................................66

Figure A33: Performances of different execution environments on Eddie when executing Ray Tracer benchmarks ..............................................................................................................67
Acknowledgements

First and foremost, thanks go to Malcolm Illingworth for supervising this project. This was Malcolm’s debut as a project supervisor; he has been approachable, insightful and reliable throughout.

The teaching staff at the EPCC also deserves thanks for the efforts over the last 12 months. They are surprisingly passionate and extremely knowledgeable about their subjects.

Finally I would like to thank the inspiration behind the name of the machine Logie; an incredibly giving person, which this small gesture of naming a machine in their honour does not do them justice. Ironically, both Logies share similar memory limitations.
Chapter 1

Introduction

Java, like all living languages, is ever evolving and improving. Due to its developments during the last decade, and the fundamental principles of Java that make it so widely used, High Performance Computing (HPC) pioneers are aiming to establish if Java is a viable option for so called Grande codes and applications. A Grande code/application can be defined as one that requires a high amount of resources, whether that is processing, computational, I/O, bandwidth, memory or any combination of these properties.

C and Fortran are traditionally the languages used by HPC programmers for the development of Grande codes as they have always offered the greatest performance, coupled with the relative ease of programming complexity. The performance difference between these traditional languages and Java used to be in the multiple orders of magnitude. Grande applications, by their very nature, will often have a very large run time and whilst there is a significant discrepancy between the performance offerings, Java will not be a viable option for Grande applications.

However, those traditional HPC languages have drawbacks that Java addresses when performance and speed are not the primary concerns of the developer. Java is inherently portable; applications can be deployed or migrated from machine to machine seamlessly when there is a compatible Java execution environment available, without further configurations. In theory this is true for C/Fortran when there are the necessary compiler and modules available on each machine. In reality many, many hours are wasted attempting to port C/Fortran codes. Grande applications will very often outlive the machine they are developed for, and therefore porting the code becomes essential for the application’s life to continue - hence Java's portability is a huge benefit. Java additionally has numerous programming benefits over the traditional HPC languages. Java has been designed to enforce an object-oriented (OO) programming style which promotes code re-use and reduces development time. Java also is a higher-level language than C and Fortran which usually lowers the amount of time required for new developers to become proficient developers. These are the primary reasons why Java is extensively used for programming non-HPC/Grande applications and also why Java is generally taught in schools and universities whereas the more traditional HPC languages are forgotten. The number of Java developers is growing while the number of C and Fortran developers is decreasing. If Java ever offers comparable performance to C and Fortran it is feasible to suggest it may become the most widely used language for Grande applications.
In the final years of the 20\textsuperscript{th} century the Java Grande Forum (JGF) was established. This initiative aimed to develop recommendations on how to update Java for the benefit of HPC and/or establish standards for Grande libraries and services. Grande requirements are outside the scope of the original design specifications of Java, but the JGF believed there was enough potential in Java to fulfil their goals. On behalf of the JGF, the Edinburgh Parallel Computing Centre (EPCC) developed the Java Grande Forum Benchmark Suite (JGFBS, “benchmark suite”) to measure the performance of Java in a given execution environment. The primary aspects of an execution environment consist of the host machine’s hardware, software and Java components. The hardware aspects are: number and performance of processors, memory, network bandwidth and architecture. The software aspect is the operating system, and the Java components are the versions of the Java Runtime Environment (JRE) and Java Virtual Machine (JVM). The premise for benchmarking an execution environment is to highlight the aspects that enable higher performing Grande codes therefore encouraging development of these aspects for HPC use.

There are a number of components of the JGFBS. The original component is a sequential benchmark suite [1]. This suite was designed to benchmark the performance of Java for executing Grande codes in a sequential manner; a logical predecessor and stepping-stone to a parallel benchmark suite. There were six principle criteria which the sequential suite should adhere to, it should be: representative, interpretable, robust, portable, standardised and transparent. In order to meet these objectives there are types of benchmark in the suite that measure low-level operations, simple computational kernels and real-world applications, respectively named Section I, Section II and Section III benchmarks. Many HPC computer systems execute jobs in a batch system process. With this being the case it was a design decision not to have any graphical components in the JGFBS to maximise portability.

Following the completion of the sequential component of the benchmark suite the EPCC developers, and affiliates, developed parallel versions [2] of a subset of the sequential benchmarks. A parallel version is obviously necessary to measure the performance offered by Java for Grande applications as, by their very nature, Grande applications will almost certainly be parallel applications. This parallel benchmarks are the second component of the benchmark suite. The parallel benchmarks extend on the sequential benchmarks by utilising Java's threading capabilities throughout. The low-level operations, Section I, of the parallel suite differ from the sequential suite to stress and measure the costs of utilising the threading functionality. The authors concluded that Java's threading capabilities were scalable, but the software support was immature at the time. In the succeeding years, Java's software support has matured significantly; there is now far greater concurrency support and functionality available within Java.

In addition to the sequential and parallel Java components, the JGFBS contains a language comparison component. The language comparison component consists of sequential C implementations of a subset of the simple kernels and real-world applications from sections II and III of the sequential suite. It would not be relevant, or even possible, for C implementations of the low-level operations of Section I
within the Java benchmark suites to be part of the language comparison suite as those benchmarks are Java, rather than Grande, specific. This language comparison suite aims to allow for an accurate performance comparison of Java and C in execution environments identical, apart from the Java and C aspects. The comparison should indicate how viable Java really is for Grande applications.

The final component of the JGFBS is a set of ‘MPJ’ benchmarks. MPJ is a Java class library that utilises mpiJava [3] to wrap existing MPI libraries that are written in C, in an attempt to provide message passing functionality for Java. MPJ was intended to allow the use of Java on distributed memory architectures. A limitation of MPJ is the reliance on C libraries; reducing the portability that is a foundation of Java. Efforts to create pure Java MPI implementations have been made with varying success, but none have been widely accepted by the HPC community. Additionally, in its current state, MPJ requires specific hardware and software configurations that are unlikely to be available without an administrator’s intentional attempts to make his/her environment support MPJ.

1.1 Previous Research

The authors of the study [2] are some of the primary contributors to the work carried out by EPCC towards the JGFBS. The study is a presentation of the parallel component of the benchmark suite and includes benchmarks results and analysis of Java’s performance at the time of the publication. The results of the low-level operation benchmarks revealed that multi-threaded Java was not suitable for anything other than most coarse-grained tasks. Results from the computational kernels, and real-world applications, were more positive with the demonstration of scalability being particularly well-received, however there was an overriding concern relating to the immaturity of Java. A comparison of Java and C is not exhibited in [2] which would have given a base for all future comparisons of Java and a traditional HPC language.

The research presented in [4] shows an evaluation of Java's potential for Grande applications by use of the sequential component of the benchmark suite in execution environments of the time (2003) and a comparison with the language comparison component. It was surmised that, even as far back as 2003, Java was well optimised for low-level operations but was not so impressive when executing the kernels and real-world applications. It is reported in [4] that C outperformed Java by a factor of 3 to 5 in the real-world application benchmarks. Java fared better in the computational kernels benchmarks being outperformed by a maximum factor of 2, this however could still be interpreted as significant. This research [4] provides an interesting platform to gauge, by how much, performance gain Java has achieved in the last eight years.

Building on the initial efforts of the Java Grande Forum, there have been efforts to track the state of Java for HPC. In 2008 a publication [5] recognised newer computer architectures and JVMs had not been properly benchmarked to evaluate the, then, current state of Java for HPC. By 2008 JVMs offered far improved multi-threading support and had the potential to take full advantage of multi-core CPUs. In order to
perform their evaluation, the authors of [5] implemented the NAS Parallel Benchmarks (NPB) executed on a variety of JVMs from multiple vendors. Detailed information and specification of the NPB can be found in [6]. Similarly to [4] the authors of [5] presented a performance comparison of Java and a traditional HPC programming language, but differing form [4] the traditional language is Fortran. The authors of [5] conclude that Java offers good computational performance but bad scalability. The cost factor of a Java implementation ranged from 1.5 to 6, as the size of the architecture increased.

We have seen the reasons why Java needs a benchmarking suite, and how the JGFBS has been developed and evolved from a sequential benchmark suite to include parallel, language comparison and MPJ components. We have seen how the performance of Java has steadily increased over the years and how the performance gap with the traditional languages has narrowed. Now we will see if the gap still exists once the parallel benchmark suite is updated to include the recent functional updates in the Java programming language and how the most modern Java environments will perform. Is Java finally fulfilling its perceived potential and is it ready to offer itself as a viable option for Grande applications to the HPC world?
Chapter 2

Goals, Methodology and Framework

2.1 Primary Goals

The primary goal of the project is to update the parallel component of the Java Grande Forum Benchmark Suite, to incorporate new functionality of the concurrency package released with Java 5. Once the updates are complete it will be practical to benchmark the performance of Java in a given execution environment, and determine the viability of using Java for Grande applications.

It will be necessary to add extra benchmarks to Section I of the parallel component of the benchmark suite. These benchmarks will measure the cost and performance of the new low-level functionality, introduced via the concurrency package. Previous contributors to the benchmark suite have warned there may be a small number of subtle bugs that need eradicating. Lastly, the benchmark suite does not record the data it produces to file. It is essential that the suite must be modified to record data in order to get a meaningful collection of data for analysis.

The final facet of the primary goal is to determine if Java is a viable option for the development of Grande codes. To achieve this, execution environments must be benchmarked with the updated parallel benchmarks, and results compared with results from benchmarks obtained from the language comparison component of the benchmark suite. Benchmarking Java and C in the same execution environment, with identical hardware, will be necessary.

2.2 Secondary Goals

Java was benchmarked in 2001 by the developers of the benchmark suite and was seen to be promising but immature. It is possible to quantify by how much Java has matured in the successive 10 years. To achieve this, execution environments will be benchmark with the Java aspects being the only variable; a subset of the Java environments presented in [2] by the developers of the parallel component of the benchmark suite and a modern Java environment released in February 2011. It will not be possible to benchmark the Java environments that predate the release of Java 5 after the suite has been updated, as it will include functionality of the concurrency package. Thus, the benchmarking efforts intended to determine the Java improvements will be carried out before the parallel suite is updated.
It will be obligatory to analyse the performance gain of utilising functionality within Java’s concurrency package. This will require a comparison of the benchmark data recorded from the updated parallel suite with benchmark data from the original parallel suite.

Comparing alternative execution environments will hopefully reveal telling characteristics of a good execution environment for Grande applications. Benchmarking environments of different hardware and architectures will give useful information relating to the best and most important aspects of a HPC machine.

Java does not place any restrictions on the number of threads an application can create. The JVM’s thread scheduler assigns threads to be executed on cores in the host environment. A programmer does not have control over the thread scheduler so must put their faith in it performing a sensible allocation of one thread to one core, where possible. However, a programmer has no guarantee the scheduler will not overload a core, and assign a single core multiple or all the threads of an application, even if there are other unused and available cores. Ideally the number of threads won’t exceed the number of cores, and a 1:1 ratio of threads to cores will be maintained by the scheduler. However, in order to measure how performance is affected by undesirable thread scheduling, benchmarks will be performed with the number of threads exceeding the number of cores in the execution environment; simulating bad thread scheduling.

Parallel programming techniques will incur an overhead cost. This may be a small start-up penalty or more costly if the implementation of the technique is inefficient, whether that be because of a mistake by the programmer or a deficiency of the programming language. An analysis of benchmark data from the sequential and parallel, running on a single thread, components of the JGFBS will allow the parallel overhead costs to be quantified.

2.3 Goals Summary

This project has three clearly defined high-level goals; two of these include multiple sub-goals, which could be broken down even further. In the expected chronological order of completion, the goals (primary goals in bold) are:

- Determine the intrinsic performance gain Java has achieved since the parallel component of the JGFBS was initially released. This involves benchmarking and analysing the performance of legacy and modern Java environments.
- Update the parallel component of the JGFBS. Introduce functionality from Java’s concurrency package, add new benchmarks, fix bugs and introduce data recording functionality.
- Compare the performance offered by Java to that of a traditional HPC language.
- Benchmark Java in numerous execution environments to:
a. determine the performance gain of utilising functionality of the concurrency package by analysing benchmark results from the original and updated parallel suites;
b. determine what non-Java aspects of an execution environment contribute to high performing Grande codes;
c. determine how performance of Java applications is affected by increasing the number of threads beyond the number of cores contained in the host environment;
d. quantify the overhead cost of using parallel programming techniques.
Chapter 3

The Existing Parallel Component of the Java Grande Forum Benchmark Suite

The benchmarks within the parallel component of the Java Grande Forum Benchmark Suite [2] are parallel implementations of a subset of benchmarks from the sequential component of the benchmark suite [1]. The deliverable parallel component consists of a set of benchmarks and utility classes; two classes that record the performance and timings of the benchmarks, and an interface for each of the three sections of the benchmarks.

All benchmarks implement one of the interfaces specified in the component’s utilities; there is a separate interface for each of the three sections of benchmarks. These interfaces guarantee consistency throughout the sections and the suite as a whole, reduce development time and increase coding efficiency.

The benchmarks in sections II and III are executed with an optional data set size. The data set size will affect the performance of the benchmarks by stressing their load tolerance. There are three options that reflect the data set in Section II - sizes A, B and C; and two in Section III - sizes A and B.

All benchmarks have one or more accompanying drivers that execute the benchmark. Section I of the suite also contains a driver that executes all the benchmarks in the section. Sections II and III contain drivers that execute each benchmark in their section for a specific data set. The driver is necessary to allow the benchmarks to be executed independently or as a section, and if necessary, size group. The driver also segregates the model and the controller in sections II and III. Through an argument received from the launcher, the drivers dictate how many threads the benchmark should utilise for a given execution. If a driver receives no argument from the launcher the default is to execute benchmarks using a single thread, resulting in the benchmark(s) being executed in a sequential manner.

3.1 Section I

Section I consists of low-level micro benchmarks that have been designed to gauge the cost and scalability of Java’s implementation of core parallel functionality. Thread
creation, termination, barriers and synchronisation can be considered to be Java’s core parallel functionality.

3.1.1 Barrier Benchmark (JGFBenchmarkBench)

This benchmark is designed to measure the performance of barrier synchronisation. When the original parallel suite was developed Java threads had no native barriers. This limitation of Java resulted in the authors of the parallel suite producing two of their own barrier implementations: Simple Barrier and Tournament Barrier.

Simple Barrier uses a counter which is shared between threads. Each thread increments the counter and then yields, unless it is the thread that increments the value of the counter to equal the number of threads that share the counter, then the thread will reset the counter and awake the other threads.

3.1.2 Fork-Join Benchmark (JGFForkJoinBench)

The fork-join benchmark measures the time spent creating (forking) and killing (joining) threads by the JVM. To effectively gather this measure, threads are created that perform an arbitrary task, a task that is robust against optimisation and cannot be disregarded by the compiler, and then killed. The cost of running the arbitrary task is subtracted from the cost of the forking and joining to give the desired measure.

This benchmark offers no useful data if executed in a sequential manner.

3.1.3 Synchronisation Benchmark (JGFSyncBench)

Java natively provides two basic techniques for data synchronisation: synchronised blocks and synchronised methods. Protecting the integrity of the data using either technique will incur an overhead cost; the performance of both techniques is measured by this benchmark.

3.2 Section II

The benchmarks in this section are parallel implementations of simple kernels that represent typical numerical computations that are commonplace in real scientific applications.

3.2.1 Crypt Benchmark (JGFCryptBenchSize[\{A,B,C\}])

IDEA (International Data Encryption Algorithm) encryption and decryption is applied to an $N$ byte array. The work is distributed between threads in a block fashion.

3.2.2 LUFact Benchmark (JGFLUFactBenchSize[\{A,B,C\}])

LUFact solves an $N \times N$ linear system by LU Factorisation and then a triangular rule, where $N$ is the size of the data set. The only part of the computation that is parallelised is the factorisation; the work is distributed between the threads in a cyclic
manner.

This is a Java implementation of the Linpack benchmark.

3.2.3 Series Benchmark (JGFSeriesBenchSize[[A,B,C]])

The first $N$ Fourier coefficients of the function $f(x) = (x + 1)^n$ in the interval 0,2 are calculated by the Series benchmark. The work is simply spread between threads.

3.2.4 SOR Benchmark (JGFSORBenchSize[[A,B,C]])

Where $N$ is the size of the data set used, the benchmark performs 100 iterations of successive over-relaxation on an $N \times N$ grid. The work is distributed in a block fashion.

3.2.5 SparseMatMult Benchmark (JGFSparseMatmultBenchSize[[A,B,C]])

An unstructured sparse $N \times N$ matrix, where $N$ is the size of the data set, for 200 iterations of computation. The parallelisation occurs in a block fashion.

3.3 Section III

Section III is a set of benchmarks that emulate real world applications.

3.3.1 Molecular Dynamics Simulation Benchmark (JGFMolDynBenchSize[[A,B]])

The benchmark models particles interacting in a Lennard-Jones potential. The parallelisation is in a cyclic fashion for best load balancing.

3.3.2 Monte Carlo Benchmark (JGFMonteCarloBenchSize[[A,B]])

Using Monte Carlo techniques to perform a financial simulation, this benchmark calculates product prices deduced from an underlying resource. This benchmark uses block distribution for parallelisation.

3.3.3 Ray Tracer Benchmark (JGFRayTracerBenchSize[[A,B]])

This benchmark emulates a 3D ray tracer. There are 64 spheres in the scene and the resolution of the rendering is $N \times N$ pixels, where $N$ is the size of the data set. For best initial load balance this benchmark has been parallelised with a cyclic distribution.

3.4 Invoking Parallelism

The benchmarks must be used in parallel to achieve a fundamental objective of the suite; gauge the parallel performance of Java. This involves using Java threads.
1. Runnable thobjects[] = new Runnable[nthreads];
2. Thread[] th = new Thread[nthreads];
3.
4. for (int i = 1; i < nthreads; i++) {
   5.    thobjects[i] = new Runner(i);
   6.    th[i] = new Thread(thobjects[i]);
   7.    th[i].start();
   8. }
9.
10. thobjects[0] = new Runner(0);
11. thobjects[0].run();
12.
13. for (int i = 1; i < nthreads; i++) {
   14.    try {
   15.        th[i].join();
   16.    } catch (InterruptedException e) {
   17.        }
   18. }

Algorithm 3.1: Method of creating, utilising and killing threads.

The method of creating, utilising and killing threads throughout the suite is illustrated by Algorithm 3.1. Lines 10-11 of Algorithm 3.1 illustrate how the master thread is utilised to perform parallel tasks. If the first loop of Algorithm 3.1 iterated between 0, rather than 1, and the desired number of threads, there would already be the full allocation of threads performing the parallel task. The master thread would then have sleep and would have no straightforward means of being awakened.

The method of Algorithm 3.1 is not particularly efficient if the whole method is within another loop as every thread is created and terminated multiple times.
Chapter 4

The Updated Parallel Component of the Java Grande Forum Benchmark Suite

The parallel component of the JGFBS has undergone a significant overhaul in order to take advantage of the newest features offered by Java's concurrency package. Benchmarks in all three sections of the suite have been updated and new benchmarks in Section I have been introduced. Additionally the utility classes have been modified to optionally output data to file, and to accommodate the updates in the benchmark classes and the improved performance of the hardware that will execute the updated suite.

The suite did not have any means of writing the recorded performance data to file. This functionality is crucial in order to be able to refer to the performances of past executions, and to gather sufficient data that will enable the calculation of a fair average of the performance and timings of the suite, both as it was originally and how it is now the updates have been completed. However, the suite primitively output the performance and timing data to the terminal. The output messages were in the following formats for Section I benchmarks and for Sections II and III, respectively:

```
<benchmark name> <performance> <operation type>/s

<benchmark name> <performance> <operation type>/s <time taken>
```

It was not necessary to stop the suite from outputting this data to the terminal so this feature has been retained.

The suite now offers the functionality to record the performance and timings data to a comma-separated values (“CSV”) file. If a \texttt{-e} flag is set as the second argument passed to the application from the launcher (the first argument is reserved for setting the number of threads the suite will utilise) all the drivers will invoke the functionality that writes performance and timings to file. The raw data in the CSV files are almost the same format as what is output to the terminal, except the values are delimited by commas rather than spaces. The CSV files are placed in the same directory as the benchmark classes and are keyed in the format '\texttt{<a>,<b>,<c>,<d>,<e>,<f>.csv}', where:

- \texttt{<a>}  - version number of the JVM that is executing the suite;
There was no error logging in the original JGFBS suite. All caught exceptions were naively ignored, without even a terminal message alerting the user of the occurrence. The updated suite records all caught exceptions in an error log which is written to file for reference. If errors have occurred, users are given a visual notification when the application terminates informing how many have been caught and an instruction to refer to the error log file. The error log is placed in the same directory as the benchmark class that has thrown the exception(s) and is keyed in a similar format to the data log CSVs: '<a>,<b>,<c>,<d>,<e>,<f>_errorlog.txt' (the parameters match the values of the data log key). This should allow a user to identify corresponding data and error logs. All classes in the updated suite perform error logging, including ones that have not undergone modifications for other purposes.

4.1 Concurrency Package (java.util.concurrent)

This subsection details the objects and devices from Java’s concurrency package [7] that have been introduced within the benchmarks of the parallel component of the JGFBS. The concurrency package contains a large number of classes that are useful for concurrent programming. The first release of the concurrency package was in Java 5; a number of years after the release of the JGFBS.

4.1.1 CyclicBarrier (java.util.concurrent.CyclicBarrier)

A CyclicBarrier object [8] is synchronisation device that forces a number of threads to wait at a common point. A barrier is a common concept in parallel programming but the cyclic barrier is the first attempt at an intrinsic Java barrier implementation.

Due the lack of an intrinsic alternative at the time of development, the parallel component of the JGFBS contains frequent use of pseudo barriers, the most common being instances of a Tournament Barrier. The Tournament Barrier implemented, based on the specifications of [9], depends on a more complicated variant of Algorithm 4.1.
The pattern described by Algorithm 4.1 is inherently flawed. There are no guarantees given by Java’s memory model that ensures the JVM will not re-order the writes of Thread 0 (lines 5-6) or the reads by Thread 1 (lines 6-7). Additionally the value of wait could be stored in a register on Thread 1 in order to reduce reads, but this would be result in wait not being updated on Thread 1.

A safe version of a Tournament Barrier can be implemented but would be undesirably slow and a bottleneck. To circumvent the problems associated with the Tournament Barrier implementation, all the instances of a Tournament Barrier have been replaced with a CyclicBarrier in the updated parallel suite.

4.1.2 Thread Pools

A Java thread pool [10] is a collection of worker threads that perform Runnable [11] or Callable [12] tasks. The size of the collection is limitless but it is sensible to size the collection equal to the number of cores the application will be running on. A thread pool creates worker threads just once; threads are then used, and can be reused an unlimited amount, to perform tasks. The method reduces the cost of thread creation as each worker thread is only created once. The fork-join method that is abundant throughout the original parallel component of the JGFBS manually binds a specific task instance to a thread which must be terminated (joined) on completion. The fork-join method requires threads to be recreated (forked) for all subsequent tasks.

There are numerous instances of thread pools within the updated parallel suite that have replaced the more expensive fork-join method.

4.1.3 Atomic Objects (java.util.concurrent.atomic)

An atomic sequence is a core concept of parallel programming; providing lock-free and thread-safe access and updates to variables. The concurrency package contains a toolkit of classes [13] that provide atomic functionality for single variables.

4.2 Utility Classes

4.2.1 JGFInstrumentor

Originally the suite would output a title (header) message to the terminal that contained the name and version of the suite, the section of the suite that was being executed, the data set key being used and the number of threads that the suite would
utilise during the given execution. For the benefit of the users, the title message has been amended to include a time-stamp. Additionally, for the benefit of the users, a footer message has been introduced that simply informs what time the benchmarking has finished. These time-stamps give an indication as to how long the benchmarking process has taken and should alarm a user if it has been unexpectedly fast or slow. Each driver class outputs the header and footer messages via JGFInstrumentor.

This class is responsible for keeping track of how many errors have occurred in the benchmarking process. If any have occurred there will be a message displayed after the footer to alert the user and present the error count with a suggestion to refer to the error log.

JGFInstrumentor has been updated to enable the data logging that is offered by all benchmarks.

The parallel suite originally measured timings of benchmark operations in milliseconds. Due to the improvements in the aspects of the execution environments over recent years, it has been necessary to increase the accuracy of the timings by changing the measure precision to nanoseconds.

4.2.2 JGFTimer

The performance measures and timings are recorded and output from this class. In order to allow the data logging there have been extra methods, derived from pre-existing ones, created to write the data to file.

4.2.3 JGFParse

This is a new non-benchmarking class that has been introduced to parse the data logged from the benchmarks. It is conceivable to execute the benchmark suite hundreds, even thousands, of benchmarks to gather enough data to calculate fair measures of performance. This class accepts a number of arguments to allow a specified subset of data logs. JGFParse parses the subset of CSV data logs, calculates the average performances and timings and finally, outputs the values to a new CSV file. The output CSV file is keyed ‘AVG - <a>,<b>,<c>,<d>.csv’, where:

- <a> - version number of the JVM that is executing the suite;
- <b> - host name of the machine that is running the JVM and the suite;
- <c> - section number of the suite whose data the file corresponds to;
- <d> - the data set used;

JGFParse has been coded in the same style as the rest of the suite. The class contains the same layout as benchmarks in sections I, II and III and achieves parallelism to maximise performance by utilising thread pool functionality afforded by Java’s concurrency package.

Unlike the rest of the classes in the utilities domain, JGFParse contains a `main` method and can be directly executed. It can be viewed as a plug-in application that has been
included in the suite for its extremely useful functionality. Without JGFParse the data logging aspect of the suite would be expendable.

4.3 Section I

4.3.1 Driver (JGFAll)

JGFAll is the driver class for the whole section. It executes all the benchmarks in the section consecutively. It has been amended to include calls to the new benchmarks, JGFAtomicBench and JGFTThreadPoolBench, and switch on the data logging if the -e flag is received from the launcher and print the footer message, both via the JGFInstrumentor utility class.

4.3.2 Barrier Benchmark (JGFBenchmark)

The original implementation of this benchmark class incorporated the use of a Tournament Barrier, this class has been replaced with the use of a CyclicBarrier.

4.3.3 Fork-Join (JGFJoinBench) and Synchronisation (JGFSyncBench)

These classes have been updated to, via JGFInstrumentor, perform error logging and, if not run via the JGFAll driver, print a footer message.

4.3.4 Atomic Benchmark (JGFAtomicBench)

This is a new benchmark class that aims to measure the performance of Java's atomic classes, a small toolkit found in the concurrency package. The atomic classes support lock-free and thread-safe updates of variables, guaranteeing data integrity. The atomic concept is a crucial facet of parallel programming that has been introduced to Java.

This benchmark is based on the benchmarks in the JGFSyncBench class. Each benchmark performs updates to a single shared variable; the synchronisation is manually controlled in the JGFSyncBench benchmarks by using the synchronized keyword whereas the atomic objects automatically control the synchronisation in the JGFAtomicBench class.

This benchmark, unlike all others in Section I, contains a method to self-validate as it is trivial to calculate how many updates should have occurred on, and therefore the correct value of, the atomic variable.

4.3.5 Thread Pool Benchmark (JGFTThreadPoolBench)

Another new benchmark, JGFTThreadPoolBench is designed to measure the performance of Java thread pools [10]. Java thread pools offer an efficient method of creating and reusing threads as opposed to manually creating them on demand and joining them after they have completed their job. The manual method is used extensively throughout the original version of the suite and remains commonplace in the updated suite, but has been replaced by use of thread pools where possible.
This benchmark class is based on the JGFForkJoinBench benchmark. The action of threads in this benchmark is to simply perform a calculation that will not be optimised away by the interpreter and then return to thread pool to be reused if required. The threads in the JGFForkJoinBench benchmark perform the same calculation but instead are then manually killed by using a thread’s intrinsic `join()` method. Using this new benchmark we can therefore accurately measure the performance benefit of Java thread pools.

4.4 Section II

Only two benchmarks have received threading updates.

4.4.1 LU Factorisation Benchmark (JGFLUFactBenchSize[{A,B,C}])

The LU Factorisation has had its use of the naïve Tournament Barrier replaced by the CyclicBarrier.

This benchmark caused an error when executed on the oldest JVM used. For an unknown reason the benchmark would cause the application to freeze on reaching a certain, seemingly harmless line of code in the benchmark. The line of code and bug in question predates the updates carried out to the suite and has not been resolved. There were no problems executing the benchmark on any other JVM. This has resulted in the relevant driver classes to check the JVM version and skip this benchmark if necessary.

4.4.2 SOR Benchmark (JGFSORBenchSize[{A,B,C}])

This benchmark used a custom synchronisation method that produced the same effect as an OpenMP barrier. This effect can be achieved by a CyclicBarrier. Subsequently the custom synchronisation method has been succeeded by a CyclicBarrier.

4.5 Section III

4.5.1 Molecular Dynamics Simulation Benchmark (JGF MolDynBenchSize[{A,B}])

This is another benchmark that used the Tournament Barrier whose use has been succeeded by the use of a CyclicBarrier.

4.5.2 Monte Carlo Benchmark (JGF MonteCarloBenchSize[{A,B}])

Originally, there were a fixed number of thread objects that were created (forked) for every block of work and killed (joined) after its block had been completed. The updated benchmark capitalises on the accomplished functionality of the concurrency package and adopts a thread pool that exhibits efficient thread reuse.

A subclass of this benchmark contained what has become a bug. The class (RatePath) contained a variable named `enum`, a keyword of Java. `enum` has not always been a keyword, and was not when the original suite was released. Consequently the suite
would compile and run for JVMs predating Java release 1.4 only. The fix was trivial: rename the variable with an arbitrary non-keyword.

4.5.3 Ray Tracer Benchmark (JGFRayTracerBenchSize[(A,B)])

The parallelisation in this benchmark was achieved by a cyclic distribution of the work to be done. The scene is optionally divided into a number of intervals, each containing a number of rows that require processing. The original implementation of this benchmark processed the scene as a single interval and assigned a number of rows (the amount of rows divided by the amount of threads) to each thread. Threads were created, completed their work and then killed. Threads were not reused and the work load was not dynamically balanced at runtime. The updated implementation divides the scene into multiple single-row intervals and utilises a thread pool to repeatedly reuse threads to work on these single-row intervals until the work is completed. Hence, this implementation has automatic dynamic load balancing built it.

4.6 Testing and Verification of Updates

Verifying the correctness of the updates is not always a trivial matter. The benchmarks in Section I perform calculations and actions that are meaningless as they are actually counting the cost and measure the performance of parallelising Java. These benchmarks do not therefore offer any output or result that can easily be validated. An exception to this is the new atomic benchmark which does offer an obvious checksum, and therefore contains a validate() method.

However, the benchmarks in sections II and III are based on real applications and their outputs can be validated. All benchmarks in these sections contain validate() methods so we can be confident the changes to these codes do not interfere with the correctness of the benchmark.

The code updates in all the sections have been stringently checked for logical errors, they obviously do not contain bugs that could be trapped at compile time.

Each benchmark does now report errors. This error reporting indicates an obvious failed benchmark and that result should be ignored.

JGFParser has undergone a significant amount of stress/load, system and integration testing throughout its development. The JGFParser can calculate the averages of over 6000 data logs in less than 1 second. The validity of the JGFParser class has been ascertained by performing manual average calculations of a relatively small number of data logs. The performances and timings were consistent between the manual calculations and the JGFParser.
Chapter 5

Results and Analysis

The JGF suites are designed to benchmark an execution environment. The Java aspect of an execution environment is the Java Runtime Environment (JRE) and the JVM, which interprets, using a Just-In-Time (JIT) interpretation technique, and executes Java codes. The other primary aspects of an execution environment are a host machines hardware and software configurations: CPU(s), memory, network bandwidth, interconnect and operating system.

Benchmark results presented in this chapter (except where explicitly stated in section 5.1) have been recorded from executions of the sequential, parallel and language comparison suites on one of two machines: Ness [14] and Eddie [15]. Thousands of executions of the sequential, parallel (original and updated) and language comparison suites have been processed and averaged to gain a fair set of results.

Ness is a shared memory system consisting of two back-end X4600 SMP 16 processor-core nodes with 2GB of memory per core, running the Scientific Linux 5.5 operating system. Ness’s system architecture ideally lends itself to running threaded applications such as the parallel benchmark suite. Use of one entire node is the maximum permitted per job on Ness, i.e., the maximum number of cores that an execution of the parallel suite can utilise is 16.

Eddie is the compute component of Edinburgh Compute and Data Facility (ECDF). The compute component offers a number of parallel environments that have been benchmarked by the sequential, parallel and/or language comparison benchmark suites in an effort to determine the finer aspects of a parallel environment that will result in Java applications achieving the maximum of their potential.

The Java aspect of the execution environments has remained constant throughout all Java executions on Ness and Eddie; JRE 1.6.0_24 (“1.6.0”) and accompanying HotSpot 64-bit server JVM. A JVM that has HotSpot JIT interpretation attempts to adaptively optimise codes by constantly analysing the execution profile to determine the most frequently used parts of the code. The constant analysis allows for highly specific and effective optimisation of the most frequently used code sections. The lesser used sections of codes are optimised in more general terms and re-optimised less often. JVMs, like C compilers, offer vast amounts of options that vary behaviour, performance and debugging settings. To keep the results fair, none of the optional
settings are changed from their defaults for any executions of the Java suites on either Ness or Eddie.

5.1 JVM Comparisons

When the parallel component of the benchmark suite was initially released the then performance of Java time was benchmarked [2]. A subset of the Java execution environments benchmarked included Sun JDK 1.2.1_004 (“1.2.1”), Sun JDK 1.3.0 and Sun JDK 1.3.1, all 32-bit environments. In this subsection the aim is to discover if a modern Java execution environment will result in improved benchmark results, without any code updates to the suite itself. This experiment should show if Java’s intrinsic performance has increased. The modern Java execution environment being used for this test is 1.6.0.

Like 1.6.0, the JVMs of 1.3.0 and 1.3.1 utilise HotSpot JIT interpretation, whereas 1.2.1 has a “classic” JIT interpreter that optimises codes in a generic manner just once. The classic interpreter will very rarely achieve the same level of optimisation as a HotSpot interpreter.

The available Java execution environments on Ness and Eddie are out of our control. Neither machine had any of the “legacy” Java execution environments (1.2.1, 1.3.0 and 1.3.1) installed; this limitation resulted in the use of an alternative machine for the experiments of this subsection which we had total control over. The alternative machine, named Logie, is a regular desktop PC with a dual-core (two cores) 2.13GHz processor and 2GB memory. Logie runs the 64-bit operating system Windows 7 and had all but the core operating system processes terminated before the executions of the JGF suite were invoked. Due to the memory limitations of, the JVM required the initial and maximum heap sizes to be increased from the default. The parameters \(-Xms768m\) (initial heap size of 768MB) and \(-Xmx1024m\) (maximum heap size of 1GB) were set for every execution of the suite on Logie in every execution environment. Logie does not offer anything resembling the performance of the HPC machines Ness and Eddie, but it does offer a fair and consistent host execution environment to benchmark the four different Java execution environments, 1.2.1, 1.3.0, 1.3.1 and 1.6.0.

5.1.1 Section I

Figure 5.1 illustrates a massive performance increase when executing the Synchronisation benchmarks in the 1.6.0 Java environment. The environment offers circa 30 times and almost 2 orders of magnitude greater performance than that of slowest environments, 1.2.1 and 1.3.0; a trend that is apparent regardless of the number of threads being utilised. In an intermediary Java release - version 1.4 - the thread management capabilities were significantly improved [16]. Attaining the finer details of what this entailed is not a straightforward task; it is very difficult to directly contact Oracle. The increased performance displayed by 1.6.0 when executing this benchmark is indicative of the thread management improvements that were introduced after the release of the legacy JVMs. The performance of the 1.3.1 environment is significantly higher than 1.3.0 throughout, which is difficult to account for. The trends of all the performances are similar, degrading notably as the number of threads increase.
The modern environment also performs better in the Barrier benchmarks as shown in Figure 5.2, although the performance gap is far less extreme than witnessed in the Synchronisation benchmark. Both simple and Tournament Barriers, in all four environments, display matching trends. Again as the number of threads increase the performance drops. In all four environments the Tournament Barrier outperforms the simple barrier. This is to be expected as the tournament barrier is flawed. Regardless, the tournament benchmark stays relevant for this experiment as the goal is to find out if a modern environment performs better than its predecessors, whilst the benchmark is consistent throughout all environments it is relevant.

There is no discernable performance difference between any of the environments when comparing the Fork-Join benchmark results. The relative performance equalities are illustrated on Figure 5.3. This suggests there have been no updates to the thread creation method utilised by the modern environment. A more interesting comparison is presented later in this chapter when the performance of the Fork-Join benchmark is examined alongside the new thread pool benchmark, which is an alternative method of creating and reusing threads.

It is feasible that the 64-bit operating system of Logie had a better working partnership with the modern 64-bit Java execution environment than the legacy 32-bit alternatives. However this partnership is probably not the decisive factor as to why 1.6.0 outperformed its rivals for the majority of the benchmarks (its worst performance was, for all intents and purposes, equal to the performance of the other environments in the Fork-Join benchmark). The more likely reason for 1.6.0 excelling is the thread management improvements that were introduced in Java 1.4.

All benchmarks in this section demonstrate a decreasing performance trend that is inversely proportional to the number of threads utilised. This is to be consistently expected because of the limitation of Logie’s dual-core processor that is executing the benchmarks. Additionally, these benchmarks are all a test of thread management so as the thread number increases the management complexities increase, which will also contribute to the decrease in performance.

Each benchmark shows the same trends in each Java execution environment, which is indicative of the modern JVM and the legacy ones interacting with the hardware and software in consistent a manner.
Figure 5.1: Performances of Java environments when executing Synchronisation benchmark with data set A

Figure 5.2: Performances of Java environments when executing Barrier benchmark with data set A
An intriguing set of results has been recorded from the Series benchmark. The oldest Java execution environment, 1.2.1, always outperforms the other environments by an approximate factor of 2.5. This can be seen in Figure 5.4, A5.4.1 and A5.4.2. As previously mentioned, the JVM of 1.2.1 performs classis interpretation. There is a possibility that the HotSpot technique is actually hindering the performance of the JVMs in the other environments, but this is not definitive or easy to explain. Section II benchmarks can be executed using one of three data sets. The trends and performance for each environment are unaffected by the data set used, with one exception: the results of 1.6.0 executing data set C; the performance is roughly half what was produced for data sets A and B. Each execution environment’s performance output doubles when the benchmark uses a second thread, but plateaus after the number of threads increases beyond 2. The initial increase is good; it displays good scalability as we move from a sequential execution to a parallel one, but disappointing that it does not scale beyond 2 threads. However, this is not unexpected and is probably due to the limitation of Logie’s CPU hardware. Interestingly the performances do not drop when the number of threads exceeds the number of CPU cores, which is beneficial.

Unfortunately, no results could be recorded for the LU Factorisation benchmark for the 1.2.1 Java execution environment. The results recorded from the other execution environments were unusual. Figures 5.5, A5.5.1 and A5.5.2 display the performance outputs for this benchmark using data sets A, B and C, respectively. The performance
trends for Java execution environments 1.3.0 and 1.3.1 executing data set A match, steady loss of performance, with 1.3.1 outperforming 1.3.0 by an approximate factor of 1.5 when the benchmark is utilising more than one thread. The performance trend for 1.6.0 in Figure 5.5 shows a large drop as the number of threads increase and is outperformed by 1.3.1 and eventually 1.3.0. The results of the benchmark using data sets B and C are consistent, corresponding trends of all execution environments are similar between data sets. Java execution environment 1.6.0 outperforms the others when the data set is not A, regardless of the number of threads in use. There is a possibility that the executions, particularly for data set A as the trends are inconsistent with those of other data sets, were somehow compromised but this is hard to account for and merely speculation.

The performance trends for Crypt, Figures 5.6, A5.6.1 and A5.6.2, demonstrate an interesting scalability trend: irrespective of the Java execution environment and data set used, the performance approximately doubles as the number of threads is increased from 1 to 2, but then plateau in the same manner as the series benchmark. The performance of the Java execution environments can be ranked in this order for all data sets: first - 1.6.0, second - 1.2.1, third - 1.3.0 and fourth - 1.3.1. There is a negligible difference in performance between both 1.3 environments, but 1.3.1 is always superior. The legacy Java execution environments demonstrate no differing performance output whatever data set is used, whereas 1.6.0 performs better as the size of the data is increased, improving from an approximate factor of 3 better than the worst performing Java execution environments in data set A to an approximate factor of 4.5 for data sets B and C.

Figures 5.7 (data set A), A5.7.1 (data set B) and A5.7.2 (data set C) display the performance recorded from the SOR benchmark executions from each of the Java execution environments. The trends displayed in each figure match almost exactly. The performance output is higher on the smallest data set and lowest on the largest. Java environment 1.2.1 outperforms the other environments when the benchmark is executed utilising data sets B and C, but by an inconsequential amount. The classic interpretation technique is perhaps better suited to this type of code where adaptive optimisation is not going to offer any benefit. The HotSpot JIT interpreter is possibly wasting CPU cycles attempting to further optimise the code with no reward and so slightly decreasing the performance of the codes. The performance output of every Java environment, for every data set, increases by approximately 50% as the number of threads increases from 1 to 2. For all subsequent increases in threads a drop in performance is witnessed; a particularly sharp drop as threads are increased from 2 to 4 which may be contributing to the large decrease in performance as more threads are introduced.

The SparseMatMult benchmark results show reasonable scalability; Figures 5.8, A5.8.1 and A5.8.2 display the performance outputs from the Java environments for data set A, B and C, respectively. All Java environments for each data set improve by a factor of at least 2, but nothing approaching linear speedup is achieved anywhere. The best improvement is from 1.6.0 when executing data set B. The speedup Figure does not show that 1.6.0 was always the best performer of the environments, significantly better than the worst performer 1.2.1. Java environment 1.6.0 outperforms 1.2.1 by
approximate factors of 6, 7 and 2.5 when running on 16 threads for data sets A, B and C, respectively. Both 1.3 Java environments display reasonable performance as compared to 1.6.0, sometimes surpassing the modern environment when the number of threads is low. When utilising the largest data set, there is negligible performance difference between the legacy Java environments. The conclusion reached from this benchmark is that the HotSpot interpreters outperform the 1.2.1’s classic interpreter when the data set is small and medium, but as the data set gets large there is impact from the optimisation, and so we begin to see the merit of the thread management improvements that are included in 1.6.0.

The benchmarks results in this section do not display uniform performance trends from the Java environments. Primarily 1.6.0 is the highest performing Java environment indicating the successes of HotSpot techniques and the thread management improvements but there are occasions when 1.2.1, which executes classic JIT interpretation and predates the thread management improvements, outperforms the others. The trends for each Java environment remain mostly consistent between the data sets of the same benchmark.

![Graph showing performances of Java environments](image)

**Figure 5.4:** Performances of Java environments when executing Series benchmark with data set A
Figure 5.5: Performances of Java environments when executing LU Factorisation benchmark with data set A

Figure 5.6: Performances of Java environments when executing Crypt benchmark with data set A
Figure 5.7: Performances of Java environments when executing SOR benchmark with data set A

Figure 5.8: Performances of Java environments when executing SparseMatMult benchmark with data set A
5.1.3 Section III

All of the Java environments present a similar trend in the molecular dynamic simulation benchmark; a definitive performance increase when increasing the number of threads from 1 to 2 followed by a plateau. Figures 5.9 and A5.9 show Java environment 1.6.0 is usually outperforming the environments 1.3.0 and 1.3.1, which are equally (to all intents and purposes) worst performing, by an approximate factor of 2 to 3. When executing with the larger data set, performances of all Java environments are improved when the size of the data set is larger, and interestingly the performances of all the legacy Java environments show signs of converging as the number of threads increase but 1.2.1 remains marginally higher performing throughout.

Figures 5.10 and A5.10 illustrate that the trends witnessed from the Monte Carlo simulation benchmark are very similar to the ones of the molecular dynamic simulation; each Java environment’s performance approximately doubles and nearly achieves linear speedup as the number of threads is increased to 2 from 1 but then the performances plateau. The performances of the Java environments can be ranked 1.6.0, 1.2.1, 1.3.1, 1.3.0 regardless of the data set used, albeit each Java environment performs better when utilising the larger data set. Even though the performance rankings remain consistent throughout the number of threads and the data set used, there is little to separate the legacy Java environments. 1.6.0 is outperforming approximately by a factor of 1.5 in executions utilising data set A, and this approaches 2 when utilising data set B.

Displayed in Figures 5.11 and A5.11, the Ray Tracer benchmark results, like the other Section III benchmarks, present each Java environment’s performance increasing as the number of threads changes from 1 to 2 by an approximate factor of 2. Once again the performances plateau as the number of threads increase further. There is negligible performance difference between the legacy Java environments, all Java environments slightly improve for the larger data set. Java environment 1.6.0 outperforms the worst performing legacy environment (1.3.0) by more than a factor of 3 when running on 16 threads. Interestingly, 1.6.0 displays a performance loss from 4 to 8 threads that is regained when the number of threads is increased to 16 that is consistent throughout executions utilising both data sets.

The performances of the 1.3.0 and 1.3.1 Java environments in this section are consistently similar.
Figure 5.9: Performances of Java environments when executing Molecular Dynamic benchmark with data set A.

Figure 5.10: Performances of Java environments when executing Monte Carlo benchmark with data set A.
The modern Java environment clearly offers greater performance than the legacy alternatives. A probable reason for the better performance of the modern Java environment is the thread management improvements introduced into Java after the release of the legacy environments. The modern environment’s superior performance is highlighted by the real-world application benchmarks (Section III) where it consistently and significantly outperforms the other environments. Section I and II aren’t so conclusive; they stress the Java environments in ways that are not always meaningful for real applications.

Many of the benchmark results presented show a performance gain as the number of threads is increased up from 1, where the benchmark is effectively being executed sequentially, to 2, where the benchmark is being run in parallel. This is the desired result. However, when the number of threads increased further, which is beyond the number of cores that Logie’s CPU has, the performances often remain consistent. This is a positive reaction that shows good thread scheduling, either by the operating system of the JVM, where idle threads are truly idle and not consuming CPU cycles as they wait.

Figure 5.11: Performances of Java environments when executing Ray Tracer benchmark with data set A

5.1.4 JVM Comparisons Summary

The modern Java environment clearly offers greater performance than the legacy alternatives. A probable reason for the better performance of the modern Java environment is the thread management improvements introduced into Java after the release of the legacy environments. The modern environment’s superior performance is highlighted by the real-world application benchmarks (Section III) where it consistently and significantly outperforms the other environments. Section I and II aren’t so conclusive; they stress the Java environments in ways that are not always meaningful for real applications.

Many of the benchmark results presented show a performance gain as the number of threads is increased up from 1, where the benchmark is effectively being executed sequentially, to 2, where the benchmark is being run in parallel. This is the desired result. However, when the number of threads increased further, which is beyond the number of cores that Logie’s CPU has, the performances often remain consistent. This is a positive reaction that shows good thread scheduling, either by the operating system of the JVM, where idle threads are truly idle and not consuming CPU cycles as they wait.
The HotSpot techniques used by the JVMs of the 1.3 environments might not quite have matured to the standard of 1.6.0, which is a possible reason that they were often outperformed by the 1.2.1 environment with classic interpretation.

If there are applications running in legacy environments, the administrator should strongly consider upgrading to the latest release. Applications should port easily as Java is backwards compatible and the performance benefit would be substantial.

5.2 Ness: Benchmarking the Updated Parallel Component

The parallel benchmarks executions on Ness utilising 1, 2, 4, 8, 12 and 16 threads. When submitting a parallel job to Ness an entire back-end node was requested. This allowed benchmarks to have unrestricted and uncompromised access to a node and all the memory it has allocated. Reserving a full 16-core node gives the JVM access to at least as many cores as threads it will utilise for any given benchmark execution, and in many cases some cores will be surplus. However, we do not have control over the thread management aspect of the JVM but can be optimistic that a sensible allocation of only one thread to a core will be made. Assigning multiple threads to a core whilst leaving other cores idle would not be desirable.

In this subsection we present a comparison of the performance offered by the original parallel benchmarks and the updated versions. Not every benchmark in the suite will be part of this subsection of the analysis as not all the benchmarks in the suite have been updated to include new threading functionality.

The tournament barrier implemented in many of the benchmarks in original parallel suite is flawed (refer to section 4.1.1). Consequently each tournament barrier has been replaced in the updated benchmarks by a CyclicBarrier. A CyclicBarrier object, a member of the concurrency package introduced in Java 5, is a synchronisation device that allows a set of threads to wait at a common point.

Figure 5.12 displays the performance difference between the flawed tournament barrier implementation of the original parallel benchmark and the cyclic barrier of the updated suite, as well as the simple barrier results of the same benchmark executions. The tournament barrier massively outperforms the cyclic barrier by at 2 orders of magnitude. This is disappointing as an initial conclusion would be that cost of a cyclic barrier is possibly too high for Grande applications. The cyclic barrier is also an order of magnitude slower than a simple barrier and there will be many codes that would be able to use a simple barrier implementation instead. The results imply that synchronising using synchronised methods and objects like the simple barrier does is faster than synchronising using a cyclic barrier. All the barrier results displayed by Figure 5.12 show that performance of barriers are reduced as the number of threads increases. This is intuitive as we are simply waiting at a barrier for all the threads to arrive; this is likely to be a longer wait if the wait is for more threads. The cyclic barrier perhaps is overly complicated for this simple benchmark logic; it perhaps has other features that would be useful but are unused here and just become an unnecessary overhead. The overlapping results of the simple barrier performances, whose code has
not been modified, show the fairness of the test and consistency of the execution environment.

The functional threading update to the LU Factorisation benchmark has also been to replace a tournament barrier with a cyclic barrier. Figure 5.13 presents the results of the benchmark before and after the update. The figure displays a general performance decrease after the update, the exception being for executions utilising the largest data set. This does not match the trend of the other data sets or what is to be expected when replacing a quicker, albeit flawed, barrier with a slower one. These executions are probably not bound by the same factors as the smaller data set executions meaning the update is less of a drain on performance output. It is also worth noting that the old tournament barrier executions on data set B far outperform the corresponding results from data sets A and C instead of being in the middle. The performances mostly increase as the number of threads increase until the benchmark is running on 16 threads by which point the performance outputs mostly decrease.

The SOR benchmark is updated to include a cyclic barrier, but unlike the other cases it is to replace existing point-to-point synchronisation and not a tournament barrier. The trends shown on Figure 5.14 indicate that a cyclic barrier performs worse than the point-to-point synchronisation when the data set is small, but as the data set gets larger, the cyclic barrier offers the premier performance. The benchmark shows little scaling, except for data set A where it is good until it drops when the number of threads is 16. Each data set produces matching trends regardless of synchronisation technique.

The updated Molecular Dynamic simulation benchmark consistently offers better performance than the original, as displayed by Figure 5.15. This is interesting because the functional threading update has once again solely been to replace a tournament barrier with a cyclic barrier. It is difficult to speculate as to why this is. Results for both data sets present the same trends for both the original and updated benchmarks. Unfortunately this benchmark does not scale well on Ness, initial improvement is quickly reversed and as the threads increase the performance is drops rapidly.

Figure 5.16 illustrates that the Monte Carlo benchmark shows very good scalability. Almost linear speedup is achieved initially but this slightly tapers off as the number of threads increases. The threaded functional update involved in introducing a thread pool to replace the fork-join method of controlling thread reuse. There is not much increase in performance due to the update but it seems the more efficient thread pool does offer something in the executions that is utilising the smaller data set.

Again excellent early scaling is achieved in the Ray Tracer benchmark and, unlike the Monte Carol benchmark, there are no signs of fading as the number of threads increases. The more efficient thread pool implementation offers better performance than the original method of forking and joining the threads as the data set is bigger, which is probably because there is a far larger data size and therefore threads are reused far more. This shows the benefit of a thread pool, as data size and number of tasks increases. The results of this benchmark are presented on Figure 5.17.
Figure 5.12: Performances of the original and updated Barrier benchmarks on Ness

Figure 5.13: Performances of the original and updated LU Factorisation benchmarks on Ness
Figure 5.14: Performances of the original and updated SOR benchmarks on Ness

Figure 5.15: Performances of the original and updated Molecular Dynamic benchmarks on Ness
Figure 5.16: Performances of the original and updated Monte Carlo benchmarks on Ness

Figure 5.17: Performances of the original and updated Ray Tracer benchmarks on Ness
5.2.1 New Benchmarks

The thread pool benchmark is designed to measure the performance of the thread pool functionality of Java’s concurrency package. A thread pool offers the same functionality of thread creation and reuse as manual thread forking and joining. Comparing the results of the thread pool and Fork-Join benchmarks displays very positive results; thread pool far exceeds the performance of Fork-Join, as shown by Figure 5.18. The thread pool’s performance is initially 3 orders of magnitude higher than that of Fork-Join and approaches 4 as the number of threads increases and the Fork-Join performance degrades. The general result was to be expected but the magnitude of the performance difference impresses. Thread pools are far more efficient at reusing threads; threads do not have to be killed after they have completed their task, they just wait – not consuming CPU cycles – until they are either reused or the pool is shut down. The trend of the thread pool data shows a consistent level of performance regardless of the number of threads in the pool, whereas the Fork-Join performance degrades significantly as the number of threads increases. The difference in consistency can be explained by examining the method of the opposing techniques. The fork-join method creates and terminates threads repeatedly whilst the thread pool creates them just once. An asymptotic analysis of these alternate methods would suggest the thread creation costs for the Fork-Join and thread pool methods, respectively, are:

\[
\Theta = s(n(c) + n(t)) \\
\Theta = n(c) + n(t)
\]

where:

- \( n \) is the number of threads;
- \( c \) is the cost of creating a single thread;
- \( t \) is the cost of terminating a single thread;
- \( s \) is the number of tasks the threads will perform.

Atomic variable updating is a core component of multi-thread programming. Java has always offered means of guaranteeing data integrity when variables are updated via multiple threads by using either synchronised methods or synchronised objects. The concurrency package offers a more automated alternative in the form of Atomic objects that offer the same data integrity guarantees. Figure 5.19 illustrates the performances of Atomic object updates and updates to variables protected by synchronised methods and objects. The synchronised approaches massively outperform the Atomic alternative initially by an approximate factor of 3.5. However, the synchronised approaches inversely scale whereas the Atomic approach remains effectively consistent when the number of threads are increased. The degradation of the synchronised approaches, both of which offer very similar performance throughout, is so severe that they perform worse than the Atomic updates when the number of threads is just 8. The trend of the atomic results suggests it will remain consistent regardless of the number of threads. The Atomic object is clearly not just an automated wrapper for the manual synchronised approaches as one could have easily perceived.
Figure 5.18: Performance comparison of the Thread Pool and Fork-Join benchmarks on Ness

Figure 5.19: Performance comparison of the Atomic and Synchronisation benchmarks on Ness
5.2.2 Ness Summary

The updates to the benchmarks in the parallel suite result in modestly different performances in the benchmarks. There are no huge discrepancies that result from the updates which one could point towards and conclude that because of that improvement Java will be in a position to match or surpass the performance of C.

The updated benchmarks generally perform better than the corresponding originals when the data sets get large; this is to be expected as the new functionality of the concurrency package is designed to allow for better thread interaction and control which will be more abundant when the data set, and tasks to be performed, is large.

The new Section I benchmarks show Java is offering good performance for fundamental parallel programming concepts which is a prerequisite for Grande applications.

5.3 Eddie: Benchmarking the Updated Parallel Component

Eddie supports numerous parallel environments (“PE”) that can execute parallel jobs. A simple setting in submission scripts determines on what model of CPU node the job will be executed. To test alternative environments the parallel suite has benchmarked 12- (“PE12”) and 16-core (“PE16”) environments.

Nodes in the PE12 environment consist of two hex-core processors and are allocated 24GB of dedicated memory. The ratio of gigabytes of memory to cores is 2:1. Nodes in the PE16 environment consist of two quad-core processors; the environment contains two nodes to resulting in a 16-core environment. Similarly to PE12, this environment has 24GB of memory, resulting in a ratio of gigabytes of memory to cores of 1.5:1. All parallel environments on Eddie run the Scientific Linux 5.5 operating system.

The benchmark data gained from the parallel environments hopefully will return interesting results and trends. A specific detail to observe in the comparisons of PE12 and PE16 will be their performances when running with 16 threads. The JVM executing the benchmarks in PE16 will ideally assign each core one thread, but this ratio is not possible in PE12 as there are only 12 cores. Additionally PE16 offers a fair way to briefly compare the hardware performance of Eddie with Ness. PE12 and PE16 will perform benchmarks running with on 1, 2, 4, 8, 12 and 16 threads.

The comparisons between the original parallel benchmarks and the updated benchmarks have been conducted in PE16; this is comparative with the 16-core node on Ness and allows for a brief hardware comparison.

Figure 5.20 displays the performance trends of the original and updated Barrier benchmark. As the number of threads is low, the flawed Tournament Barrier implementation of the original benchmark outperforms the simple barrier, both original and updated ones, and the new CyclicBarrier. Interestingly, as the number of threads increases the tournament barrier performance drops below the simple barriers’ performance. This was unseen on Ness. All barriers’ performances decrease as the
number of threads increase, similarly to Ness. The tournament barrier is performing 2 orders of magnitude higher than the cyclic barrier at the start but drops to one order of magnitude when the number of threads is 8 and the gap narrows further as more threads are introduced. This is still a very large performance gap however. More relevant is the difference between the simple and cyclic barrier performances. That gap stays consistently around an order of magnitude in favour of the simple barrier. Both show a pretty steady decline after parallelism is introduced and the number of threads increases. The similarity of the simple barrier performances of the original and updated suite show the fairness of the results, i.e., evidence the hardware has remained consistent as there have been no code changes between versions of the benchmark.

Figure 5.21 shows the LU Factorisation benchmark shows that the original benchmark initially outperforms the updated benchmark regardless of the data set used. Interestingly, the updated benchmark remains more consistent in performance and does not degrade as much, whereas the original’s performance drops below the corresponding updated benchmark for each data set. The trend of performance is similar for each data set’s original and updated benchmarks, small improvement followed by a drop in performance. This is interesting because the modification to update the benchmark is to replace a tournament barrier with a cyclic alternative. As in the barrier benchmark the performance of the old tournament barrier drops significantly more than the performance of the cyclic; the result is explicable when considering the benchmark is not entirely barrier bound. These trends are completely different from what is witnessed on Ness. Not only does PE16 on Eddie outperform Ness, which is reasonable as the CPUs are more powerful, albeit there is a smaller memory per core ration, but also the trends of this benchmark are far more consistent between original and updated versions. It is hard to explain why the hardware of the environments react in different ways.

The trends displayed by Figure 5.22, which represent the data for the SOR benchmark, show similarities throughout the version of the benchmark but do not between corresponding data sets. The performance remains very consistent in the updated benchmark executions, unfortunately the results do not indicate good scaling but performance does not degrade either. The original benchmarks degrade massively and performances converge as the size of the data set increases. The results of this benchmark are contradictory to those of the barrier benchmark. This benchmark’s update has been to replace a tournament barrier with a cyclic barrier. It is observed that the cyclic barrier always outperforms the tournament barrier. This is hard to explain. Corresponding original and updated benchmark results show almost identical initial performance and improvement as threads increases from 2 to 4. As threads increase the updated benchmark performs gradually worse with the smaller data set and gradually better with the larger data set, but the original benchmark has dramatic performance drop on all data sets; the performance gap between the updated and original benchmark results is an approximate factor of 4 for the smallest data set and approximately 2 for the largest. Once again an inconsistency with Ness is evident (compare with Figure 5.14), and interestingly Eddie does not outperform Ness; PE16 does have a lower memory to cores ratio.

Figure 5.23 shows the molecular dynamic simulation benchmark trends are the most consistent with Ness that has been observed, particularly for the original benchmark,
compare with Figure 5.15. The updated version of the benchmark offers significant performance improvement to the original when the number of threads is increased beyond 2. Like many others from Section II, this benchmark display performance gain when replacing a tournament barrier with a cyclic barrier. This gain is contradictory to what could naturally be extrapolated after analysis of the barrier benchmark in Section I. The improvement is indicative of the more efficient method of thread reuse which is critical as the number of tasks, defined by the size of the data set in this benchmark, increase.

The Monte Carlo benchmark, as illustrated by Figure 5.24, shows disappointing results for the updated benchmark and the concurrency functionality. The updated benchmark, which utilises a thread pool, is outperformed by the original benchmark that performs manual forking and joining. This is hard to explain and conflicts with trends witnessed on Ness. Possibly the more powerful CPUs of PE16 can better handle the demands of manual forking and joining while the number of threads is low and data set is sufficiently small. However the results of this benchmark show Ness was not outperformed by the more powerful CPUs with the performances being relatively equal. Also witnessed on Ness, trends from both data sets for both the original and updated benchmark are very similar. An almost identical analysis can be projected from the data recorded from the Ray Tracer benchmarks, the results of which are represented by the trends on Figure 5.25, as the Monte Carlo benchmarks; the updated benchmark using a thread pool instead of manual forking and joining performs worse. Intriguingly performance output from the Ray Tracer benchmarks are worse that what are seen from Ness.

![Graph](image)

Figure 5.20: Performances of the original and updated Barrier benchmarks on Eddie (PE16)
Figure 5.21: Performances of the original and updated LU Factorisation benchmarks on Eddie (PE16)

Figure 5.22: Performances of the original and updated SOR benchmarks on Eddie (PE16)
Figure 5.23: Performances of the original and updated Molecular Dynamics benchmarks on Eddie (PE16)

Figure 5.24: Performances of the original and updated Monte Carlo benchmarks on Eddie (PE16)
We see very positive results from the thread pool benchmark. This benchmark keeps consistently performing at a high rate, almost 2 orders of magnitude better than the corresponding fork-join benchmark and the performance gap passes 3 orders of magnitude as the thread size increases. The thread pool’s performance does not degrade as fork-join’s does. The comparison can be seen in figure 5.26. The performance of the thread pool benchmark is equivalent to the thread pool benchmark on Ness, and while the fork-join performs slightly better the trend is the same.

Once again we see very good results from the new functionality in the concurrency package, Figure 5.27, as the Atomic benchmark exceeds the performance of the synchronised method and object alternatives. The largest performance gap between atomic and both alternatives exceeds a factor of 5. Both degrade as threads increase but atomic did show early improvement. PE16 once again offers higher performance than Ness’ for the corresponding benchmark although the trends of the atomic updates differ whereas the fork-join is similar.
Figure 5.26: Performance comparison of the Thread Pool and Fork-Join benchmarks on Eddie (PE16)

Figure 5.27: Performance comparison of the Atomic and Synchronisation benchmarks on Eddie (PE16)
5.3.2 PE16 versus Ness

As seen by the presented data PE16 and Ness mostly respond different to the same benchmarks. There are some similarities for simple benchmarks that are stressing the thread management but very little are consistent throughout. This is interesting because it is a natural assumption that while performance may be higher on the more powerful PE16, the trends of the benchmarks would at least be similar in both environments. Figures 5.13 and 5.21, for example, represent the identical benchmarks performed in both environments and yet there is a significant difference in trends. All benchmarks were executed a large number of times, in the order of thousands, and the performance data averaged to gain accurate representations of how Ness and PE16 perform. The hardware and architecture is different in both environments so different strengths and weaknesses are to be expected to an extent but more consistency was envisaged.

5.3.3 PE16 versus PE12

This section of analysis focuses on the updated parallel benchmarks only. This is an attempt to establish if there is a trend difference when the parallel suite is being executed in an environment where there are more threads than cores available, i.e., 16 threads running in PE12, and what we hope will be one thread per core, i.e., 16 threads running in PE16. So what we are looking at is the trend between 12 to 16 threads in these two environments. The results of this section will hopefully reveal how a JVM, and to an extent the operating system, will manage the burden of thread scheduling; a performance drop is almost guaranteed, but to what extent?

In every Section I benchmark the trend is identical between parallel environments. In every benchmark the performance recorded from PE16 is higher, ranging from negligible in the fork-join benchmarks to by up to a factor of almost 5 in the atomic benchmark. However, the specific reaction to the increase in processors when the number of threads is increased from 12 to 16 threads borders on identical for both environments in every Section I benchmark. Figures 5.28, A5.28.1 and A5.28.2 display a cross-section of the Section I benchmarks in the opposing environments.

Although contrary to the Section I benchmarks the PE12 environment outperforms PE16 in every benchmark in sections II and III. This is an interesting shift but not of concern as the hardware of the environments is different. What is interesting is that the PE12 performance dips after the thread count is increased from 12 to 16. Sometimes this dip is significant and other times minor, but it’s observed in every benchmark. This is highlighted in Figures 5.29, 5.30, 5.31, and all corresponding Figures in Appendix (Figures A5.29 to A5.33). Whatever the respective trend of the PE16 benchmarks is when operating on lower number of threads consistently continues when the number of threads moves upwards to 16 threads. The results are somewhat predictable, PE12 does outperform PE16, but it does have more powerful CPUs and more memory per core, but it does not scale to 16 threads as an environment with 16 cores. However predictable, it is interesting to visually represent the findings and it is clear that burden of thread scheduling far outweighs the benefit of reducing the work of each thread by increasing the number of threads beyond the number of cores.
Figure 5.28: Performance comparison of the different execution environments on Eddie for the Synchronisation and Atomic benchmarks.

Figure 5.29: Performance comparison of the different execution environments on Eddie for the Series benchmarks.
Figure 5.30: Performance comparison of the different execution environments on Eddie for the Crypt benchmarks

Figure 5.31: Performance comparison of the different execution environments on Eddie for the Monte Carlo benchmarks
5.4 Language Comparisons

One of the fundamental objectives of the project was to establish a picture of the performance offered of Java today versus what is offered by a more traditional, trusted and widely-used HPC language. At the same time as the original parallel suite was released, the JGF released a language comparison suite. The language comparison suite is a subset of the benchmarks in sections II and III of the parallel suite implementation in C. The limitation of the language comparison suite is that it is not a parallel suite. To get a fair comparison the updated parallel suite must be run on just a single thread. Thus the parallel benchmarks will effectively be run sequentially, but this is a test of Java and C’s performance. The reason for using the parallel benchmarks sequentially and not the sequential Java benchmark suite is because we want to include the updated features that have been introduced, and since we have not updated the sequential suite we must use the parallel one.

To gather full and fair set of results the Java and C benchmark suites have been executed on Ness and Eddie (PE16), with all 16 cores reserved for the executions which allows total access to the given node’s memory and should prohibit external processes compromising the executions.

5.4.1 Ness

Ness offers multiple C compilers. The default C compiler on Ness is the GNU Compiler Collection (GCC) version 4.1.2. However, GCC is widely criticized for not optimising codes as well as proprietary alternatives. There is also The Portland Group compilers (PGI) available on Ness, version 11.2. To display a fully encompassing set of results we have compared the performance of Java against C benchmarks compiled on both the GCC and PGI compilers. The GCC benchmarks were compiled with the -O3 optimisation setting and the PGI benchmarks with the -fast optimisation setting. Each benchmark forms part of either section II or III of the suites and therefore has either three or two (respectively) data sets available.

Figure 5.32 displays the performance ratios of the language comparison C benchmark results against a normalised Java rating. The ratios have been derived by dividing the performance of each benchmark by the performance of the corresponding Java benchmark. Thus the normalised performance of every Java benchmark is 1; the red dashed line on Figure 5.32 further highlights the normalised performance of the Java benchmarks. The results show C generally performs better than Java regardless of the compiler used. Excluding the Series benchmark where C outperforms Java by up to a factor of 4 when the largest data set is utilised, Java shows serious competition to C. Similarly to the Java environment analysis (section 5.1) the results also express that Java’s intrinsic performance has greatly improved in recent years. If these results are compared to the ones of [4] where similar experiments were carried out in 2003, the gap between Java and C has narrowed drastically.

Incidentally the results also indicate that GCC is hardly worse at code optimisation than PGI, but this is outside of the scope of our interest.
Similarly to Ness, Eddie also has the GCC as its default C compiler and also offers a proprietary alternative. However, when trying to compile using the alternative, complexities arose preventing the compilation; highlighting the portability issues so common with C that do not have to be considered when porting Java codes. Thus the only language comparison results gathered from Eddie were from benchmarks compiled with GCC and with \(-O3\) level of optimisation.

Figure 5.33 shows a normalised set of results for the language comparison benchmarks executed on Eddie plotted against the Java benchmark results. Excluding the Series benchmark we see Java offers very competitive performance to that of C. There are more cases of Java slightly outperforming C or the performance gap being negligible. The trends match the trends observed on Ness for the corresponding language comparison benchmarks. It is conclusive that Java is a high performing language.
5.4.3 Language Comparisons Summary

C is still quicker on the whole but Java is bridging the gap. The performance gap today is far smaller than when the Java benchmark suites were initially published. This is evident when comparing the results of the tests in this project with the ones in [4]. It is feasible to believe that Java will soon catch up with C and be a viable alternative for Grande codes. There are a number of reasons to explain Java’s improvement, namely the thread management improvements and the enhancements to the HotSpot techniques of the JIT interpreter. Add the concurrency improvements, which cannot be seen in these sequential executions, and it is possible that Java can be considered a genuine option for Grande codes.

5.5 Sequential versus Threaded

This subsection aims to determine the overhead of parallel coding techniques. Comparing results from the sequential benchmark suite with single thread executions of the parallel benchmark suite will reveal the cost of parallel programming. The sequential and parallel suites only share common benchmarks from sections II and III. The comparable benchmarks are Series, Crypt SOR, SparseMatMult, and MolDyn, MonteCarlo, Raytracer, form sections II and III respectively. Single thread executions
of both the original and updated parallel suites have been compared with the sequential suite on Ness and Eddie (PE16).

Figure 5.34 illustrates the overheads of parallel programming with data gathered from benchmark executions on Ness. The data on Figure 5.34 represents normalised performances with the sequential results equalling 1 and the parallel performances plotted. There are extra packages imported and objects used in the parallel benchmarks that are absent in the sequential benchmarks which account for an obvious overhead. The extra demands of the parallel techniques show generally a slight reduction in performance which is to be expected. The results from the same benchmark experiments carried out in PE16 on Eddie are displayed in Figure 5.35. Once again, very similar performance from the sequential and two parallel suites is observed. Inexplicably, there is a significantly lower performance from the Crypt benchmark when executed the original parallel benchmark that is consistent throughout data sets. The SOR benchmark results show reasonable improvement compared to the sequential benchmark.

![Chart showing performance ratios](image-url)

**Figure 5.34: Sequential component comparison against original and updated parallel component on Ness**
Parallel coding techniques do introduce an extra cost. But the results of this section show the overhead cost is minimal. It could be seen as good practice to utilise parallel coding techniques wherever possible even if an application is intended to be sequential. The application may be parallelised at some point and this coding style would minimise the time lost switching paradigms.

5.5.1 Summary

Parallel coding techniques do introduce an extra cost. But the results of this section show the overhead cost is minimal. It could be seen as good practice to utilise parallel coding techniques wherever possible even if an application is intended to be sequential. The application may be parallelised at some point and this coding style would minimise the time lost switching paradigms.
Chapter 6

Conclusions

This project has presented an updated version of the Java Grande Forum Benchmark Suite. This parallel component of the benchmark suite, designed to gauge the performance of Java in a given execution environment, has been updated to include new functionality, recently introduced in Java, that was not present when the suite itself was initially presented. The updated suite contains new features of Java’s concurrency package; namely cyclic barriers, thread pools and atomic objects. The new atomic benchmark has been added to the parallel suite because it measures Java’s implementation of atomically updated variables, which is a core concept of parallel programming, and cyclic barriers have been introduced throughout because their tournament barrier predecessors were flawed.

Comparisons of performance from the original parallel benchmark suite, and the updated version, show that the new functionality has had mainly positive influence. Results gained from benchmark executions in multiple environments illustrated Java’s impressive potential. The environments did not always respond in the same way to the benchmarks, but there were enough positives throughout to determine that the updates to the benchmark suite have been largely successful. Although it is clear that Java still does not match the performance offered by traditional HPC languages, namely C, the performance gap between languages is closing. Due to the improvements of thread management, HotSpot and new concurrency features, Java may well one day match, or even surpass, C and others. The results of this project show there is a cause for optimism. If and when Java does catch up, or even gets close, greater efforts may come from the HPC world to utilise Java as the benefits are huge. Portability, OO programming, a large pool of ready-made developers, are reasons to migrate to Java. If the performance soon matches the traditional HPC languages, there will not be a reason not to migrate to Java for development of Grande codes.

The performance of Java degrades as the number of threads exceeds the number of cores. This is somewhat intuitive but it is interesting to measure by how much this is apparent. Results presented in this project indicate the loss is significant and that it is advantageous to resist exceeding the number of cores as the performance output will immediately suffer. Executions of benchmarks in a 12-core environment running on 12 threads outperformed runs on 16 threads.

Another objective of the project was to determine if a modern Java environment would perform the original parallel suite better than the Java environments that were available
when the parallel suite was initially released. The answer is unequivocally ‘yes’.
Improved thread management and a matured HotSpot JIT interpreter have contributed
to Java’s intrinsic performance drastically increasing. It is evident that new Java
environments are not just released to include new functionality and bug fixes, but also
to offer increased performance.

Finally, the overhead of including parallel coding techniques when developing Java
applications does not meaningfully degrade an applications performance, if at all.
When developing Java applications it is reasonable to program for a parallel application
even if the application is initially going to be sequentially executed. This style of
coding is good practice and if the code ever is to be utilised in parallel the adaptation
will be a straightforward.
Chapter 7

Future Work

Following the completion of this project it is apparent that there is still scope to continue the work of the EPCC and JGF. Benchmarking Java will require continual effort as new releases are made available and updates to the parallel component of the JGFBS will be required as, and when, new threading functionality is introduced into Java. Java will naturally evolve and its intrinsic performance will increase as the technology matures further. This has been evident in the preceding 10 years as Java has been bridging the performance gap between itself and the traditional HPC languages.

Benchmarking Java on a large number of cores is the next natural step to determine if Java can offer the level of performance needed for Grande applications. This would be difficult to achieve with the JGFBS in its current state, as the necessary hardware configurations for the MPJ component are difficult to achieve. Additionally, an early plan for the project was to execute the benchmarks on HECToR [17], however HECToR’s back-end compute nodes do not have Java installed. Similarly to Ness and Eddie, amending the software modules on HECToR was not permitted.

Further benchmarks could be added to further extend the JGFBS and provide further assurances of the performance of Java. There are numerous other examples of benchmark suites that can offer inspiration. A parallel language comparison suite would also be beneficial for further comparisons of Java with traditional HPC languages. A pure Java implementation, or more easily portable MPJ, component should be added to the JGFBS.

The owners of Java claim [16] that thread management improvements were introduced in release 1.4 of Java. The details of these improvements are unclear. There is, however, evidence that these improvements exist when comparing the benchmark results produced from the modern Java environment to the results of the legacy environments that predate the improvements. If the finer details of these improvements were documented it would perhaps be insightful information when developing threaded Java applications.

The data recorded from the language comparison benchmarks on Eddie was slightly incomplete in comparison with that recorded from Ness. Eddie, like Ness, has a proprietary C compiler (Intel) available, but unfortunately the language comparison benchmark suite could not be successfully compiled using the Intel compiler. Data was still recorded and presented from Eddie after compilation using the GCC compiler.
suite. However, the performance gap of benchmark data recorded from Ness when compilation was achieved using the GCC suite or the proprietary alternative (PGI) was negligible; suggesting the lack of data from a proprietary alternative on Eddie would not be missed, and perhaps the GCC suite is better than it is generally credited.

As control over the software modules installed on Ness and Eddie was not permitted, Logie had to be used to benchmark a modern Java environment against legacy ones. Performing the same set of benchmarks on a HPC machine would provide a clearer picture into the intrinsic development of Java. Logie did provide a fair and consistent environment for the benchmarks, but it was running a noisy operating system that could have slightly compromised the data recorded.

The Java benchmarks were always performed with the default JVM settings, except on Logie where necessary. Investigating and experimenting with the available settings may result in the benchmarks recording higher performance.
Appendix A

Additional Figures

The figures in this appendix are numbered to correspond with the figures in the main thesis and prefixed with an ‘A’.

Figure A5.4.1: Performances of Java environments when executing Series benchmark with data set B
Figure A5.4.2: Performances of Java environments when executing Series benchmark with data set B

Figure A5.5.1: Performances of Java environments when executing LU Factorisation benchmark with data set B
Figure A5.5.2: Performances of Java environments when executing LU Factorisation benchmark with data set C

Figure A5.6.1: Performances of Java environments when executing Crypt benchmark with data set B
Figure A5.6.2: Performances of Java environments when executing Crypt benchmark with data set C

Figure A5.7.1: Performances of Java environments when executing SOR benchmark with data set B
Figure A5.7.2: Performances of Java environments when executing SOR benchmark with data set C

Figure A5.8.1: Performances of Java environments when executing SparseMatMult benchmark with data set B
Figure A5.8.2: Performances of Java environments when executing SparseMatMult benchmark with data set C

Figure A5.9: Performances of Java environments when executing Molecular Dynamic benchmark with data set B
Figure A5.10: Performances of Java environments when executing Monte Carlo benchmark with data set B

Figure A5.11: Performances of Java environments when executing Ray Tracer benchmark with data set B
Figure A28.1: Performances of different execution environments on Eddie when executing Barrier benchmarks

Figure A28.2: Performances of different execution environments on Eddie when executing Thread Pool and Fork-Join benchmarks
Figure A29: Performances of different execution environments on Eddie when executing LU Factorisation benchmarks

Figure A30: Performances of different execution environments on Eddie when executing SOR benchmarks
Figure A31: Performances of different execution environments on Eddie when executing SparseMatMult benchmarks

Figure A32: Performances of different execution environments on Eddie when executing Molecular Dynamic benchmarks
Figure A33: Performances of different execution environments on Eddie when executing Ray Tracer benchmarks
References


[8] Oracle online Java documentation. *CyclicBarrier*, available at: [http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/CyclicBarrier.html](http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/CyclicBarrier.html)


[14] Ness: Parallel machine
http://www.epcc.ed.ac.uk/facilities/ness/

[15] Eddie: Parallel compute facility
http://www.ed.ac.uk/schools-departments/information-services/services/research-support/research-computing/ecd/


[17] HECToR: Supercomputer
http://www.heatcr.ac.uk/abouthector/heatcrbasics/