



Java Performance on HPCx

Seung Hwan Jin and Lorna Smith

EPCC, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, UK

Java offers a number of benefits as a language for High Performance Computing (HPC), especially in the context of the Computational Grid. For example, Java's high level of platform independence and networking features are important in the Grid arena where application codes are expected to execute on a diverse and dynamic set of resources.

There are however a number of issues surrounding the use of Java for HPC, principally performance and parallelism. The design of the Java platform is substantially different from traditional HPC compilers, thus providing a unique set of challenges to achieve comparative performance with traditional HPC languages.

The document carries out a benchmarking activity, to evaluate the performance of different Java execution environments and to compare and contrast this performance with equivalent benchmarks in C.

1. Benchmarking Java Performance

1.1 Overview

The aim of this section is to examine the performance of a range of Java execution environments and to identify performance issues specific to Grande applications (where Grande applications are applications that require large amounts of memory, bandwidth or processing power). We begin by introducing the Java Grande Forum Benchmark Suite (JGFBS) that will be our primary metric for the Java performance analysis and evaluation throughout. This is followed by a summary of the platforms used within the study and by a short review of related work. Following this, we present and discuss the benchmark results across various test platforms, including HPCx. We discuss the performance characteristics

of each Java execution environment and then compare and contrast these results with those obtained from the C compilers.

1.2 Benchmarking Methodology

The Java Grande Forum Benchmark Suite is a set of benchmarks designed to measure different execution environments of Java against each other and native code implementations [5, 2].

1.2.1 The JGFBS as Performance Metrics

The JGFBS consists of three sections and each section is composed of multiple benchmarks as follows:

- Section 1: Low-level Operation Benchmarks
 - Arith: Measures the performance of arithmetic operations
 - Assign: Measures the cost of assigning to different types of variables
 - Cast: Tests the performance of casting between different data types
 - Create: Measures the performance of creating objects and arrays
 - Loop: Measures loop overheads
 - Math: Tests the performance of the API methods of Java Math class
 - Method: Measures the cost of a method call
 - Serial: Measures the performance of Java serialization
 - Exception: Tests the cost of creating, throwing, and catching exceptions
- Section 2: Kernel Benchmarks
 - Series: Computes Fourier coefficients of a function on a finite interval.
 - LUFact: Solves an $N \times N$ linear system using LU factorization
 - SOR: Performs successive over-relaxation on an $N \times N$ grid
 - HeapSort: Sorts an array of N integers using a heap sort algorithm
 - Crypt: IDEA encryption and decryption on an array of N bytes.
 - FFT: Performs Fast Fourier Transformation of N complex numbers
 - Sparse: Multiplies a dense vector by an $N \times N$ sparse matrix stored in compressed-row format.
- Section 3: Large Scale Application Benchmarks
 - Search: Solves a game using a alpha-beta pruned search techniques
 - Euler: Solves the time-dependent Euler equations for flow in a channel
 - MolDyn: Simulates the interaction of N argon atoms in a cubic volume
 - MonteCarlo: A financial simulation using Mote Carlo techniques
 - RayTracer: Measures the performance of a 3D ray tracer

Section 1 of the JGFBS measures the performance of low-level operations that are found in most Java programs. Section 2 tests the performance of specific operations that are frequently used in Grande applications. Finally, Section 3 is intended to represent real Java applications.

The JGFBS reports benchmark results in two forms: execution time and temporal performance [1, 2, 3]. Another useful metric is relative performance that is the ratio of temporal performance to that of a reference platform [1, 3]. We produce Java Grande Forum (JGF) numbers with this metric [1].

1.2.2 Test Platform Specification

Table 1 outlines the benchmark execution environments and systems.

| | Sun Fire E15000 | IBM HPCx | PC (single-CPU) |
|-----------------|---|--|-----------------------------------|
| CPU | UltraSPARC III 0.9GHz | Power4 1.3GHz | Pentium4 2.4GHz |
| FPU Performance | Peak 1.8Gflops/CPU | Peak 5.2Gflops/CPU | Peak 4.8Gflops |
| DRAM | Shared 48GB/48CPU | Shared 32GB/32CPU | 512MB |
| Cache | L1 (I/D): 32KB/64KB L2: 8MB | L1 (I/D): 64KB/32KB L2: Shared 1.44MB | L1 (I/D): L2: 512KB |
| OS | SunOS 5.9 | AIX 5.1 | Window 2000 Pro |
| JDK | Sun SDK 1.4.1_02 (Sun SDK 1.4.2 beta) ² | IBM SDK 1.3.0 ¹ | Sun SDK 1.4.1_03 IBM SDK 1.3.0 |
| C Compiler | Sun CC | XL C | GCC 3.2 |

Table 1 Specifications of the benchmark platforms.

1.3 Related Work

Bull et al. [3] demonstrated the use of the JGFBS as a Java performance metric and highlighted the wide difference in performance observed for individual benchmarks across the various systems. This group also [1,11] attempted to compare the performance of Java and C for a subset of the benchmark suite and laid a basis for language comparison using the JGFBS.

Mathew et al. [7] focused on an investigation of single processor performance of Java applications using the JGFBS and their own benchmark programs. They produced a good

¹ This version does not support 64-bit data mode.

² After this version, Sun JVM supports 64-bit data mode.

analysis of the benchmark results on an extensive range of platforms, which tried to explain Java performance variations on various platforms for the different benchmarks. They also identified problems with the low-level benchmarks and optimizing compilers.

Zhang and Seltzer [10] proposed their own approach to the problems of synthetic benchmarks like the JGFBS by constructing an application-specific benchmarks called HBench. They argued that traditional benchmarking failed to address the relevance between the benchmark programs and real programs. They argued their approach to the benchmark design and implementation was able to predict performance of Java applications on a range of JVMs correctly.

Lastly, Gregg et al. [4] showed dynamic profiles of the application benchmarks of the JGFBS to prove their methodology of Java performance analysis. They described dynamic bytecode analysis of the Grande applications and reasoned that a JIT compiler might not increase speedup for the applications because their test statistics showed most of the Java method execution time was spent in the non-API bytecode of the applications.

1.4 JVM Performance

Benchmark results have been obtained for five Java execution environments on three different types of platforms. On the Sun Fire E15000 (an SMP machine with 48 UltraSPARC III 0.9GHz CPUs running SunOS 5.9) we varied the JVM execution options³ to test if there is any variance in performance. We used Classic VM on HPCx (an IBM SMP cluster with 1024 Power4 1.3GHz CPUs running IBM AIX 4.3) and we used the Sun SDK 1.4.1_03 and IBM SDK 1.3.0 on the PC (Pentium4 running MS Windows 2000 Professional).

Overall, performance of the Java platforms increases as the processor speed increases as shown in Figure 2.1. From this figure we can also see that:

- Sun HotSpot Server VM shows a better performance on the Sun system than HotSpot Client VM.
- Sun Client VM outperforms Server VM on the PC.
- 64-bit data mode shows slightly worse performance than 32-bit mode on the Sun.
- There is virtually no difference in overall performance between the IBM and Sun JVMs if we take into consideration individual processor core speeds.

However, the overall benchmark performance figure fails to give a true picture of Java

³ Execution flags for Sun HotSpot VM: -client, -server, -d32, and -d64

performance for each execution environment. Figure 1 shows the overall performance (by columns) and the individual performance of each section (by lined-points) of the benchmark suite. Although the overall JGF Number shows, possibly due to the CPU speeds, an upward trend from left to right across the different Java platforms, the individual JGF numbers for Section 2 and 3 do not follow the same trend. For example, the Sun Windows JVMs shows higher performance on Section 2 while the IBM Windows JVM shows higher performance on Section 3. This illustrates one of the difficulties in analyzing benchmark performance with a single number for a range of benchmarks [9].

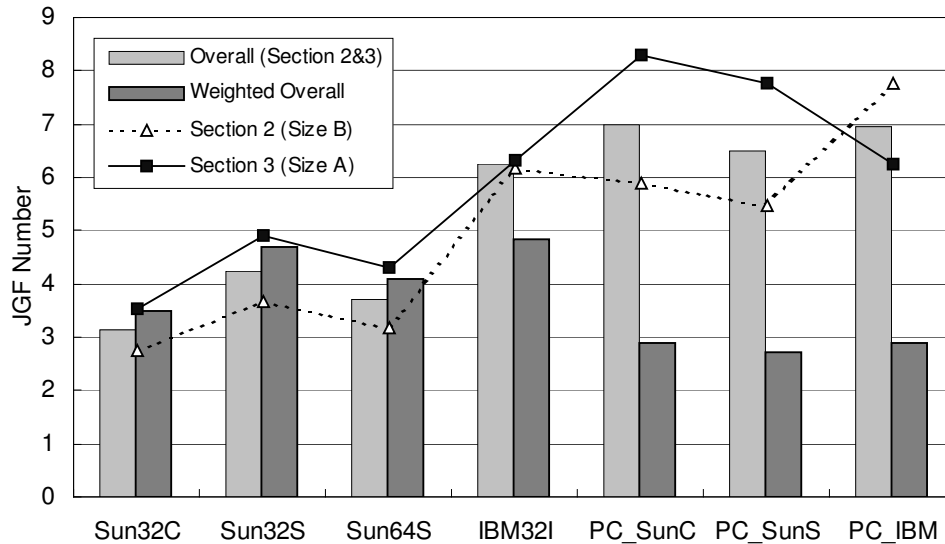


Figure 1 Java performance represented by JGF Number. Overall JGF number is a geometric mean of JGF numbers for Section 2 and 3. Weighted overall JGF number is the overall JGF number divided by a weight factor of CPU speed (1GHz = 1). A bigger JGF number means better performance. (Sun32(64)C(S) = Sun HotSpot Client(Server) VM in 32(64)-bit mode running on Sun Fire E15000, IBM32I = IBM Classic VM on IBM HPCx, PC_SunC(S) = Sun HotSpot Client(Server) VM running on PC, PC_IBM = IBM Classic VM running on PC).

We also produced weighted performance figures for the overall JGF numbers as an effective means of summarizing performance results from different benchmark platforms. We have taken account of the clock-speed differences in the test platforms, giving weight number 1 to 1GHz of processor clock speed. This is simply an attempt to compare JVM performance on benchmark platforms with different CPU clock rates. CPU performance is directly related to benchmark performance, and can come from many sources such as clock rate, processor organization, compiler enhancements, etc [8]. Considering the overall weighted JGF numbers in Figure 1, it is clear that the weighted JGF number is similar for the IBM and Sun, but decreases for the PC.

While the JGF number provides a simple overview of performance, to understand the

performance in greater detail we need to consider the individual benchmarks.

1.4.1 Low Level Operations Performance

Table 2 shows the results from section 1 of the benchmark suite, which consists of nine low-level benchmarks. We can see that the Sun HotSpot Server VM is highly optimized for Assign, Exception, Loop, and Method operations, and that IBM Classic VM is aggressively optimized for Arith and Create operations. This suggests that Java technology is able to obtain relatively high performance for many low-level operations that are commonly used in HPC applications. In fact, optimizing compiler techniques have become common in most Java virtual machines. For instance, the Sun HotSpot Server VM implements a fully optimizing compiler tuned for the performance profile of typical server applications [6].

| Section 1 | Sun32C | Sun32 S | Sun64 S | IBM32 | PC_C | PC_S | PC_I |
|-----------|--------|----------|----------|----------|-------|----------|----------|
| Arith | 3.22 | 4.9 | 4.69 | Infinity | 0.61 | 0.63 | Infinity |
| Assign | 3.3 | 55.08 | 41.99 | 17.12 | 12.16 | 169.91 | 45.53 |
| Cast | 2.93 | 9.82 | 1234.59 | 3.4 | 6.9 | 7.33 | 7.53 |
| Create | 2.52 | 6.27 | 4.87 | Infinity | 3.95 | 6.85 | Infinity |
| Exception | 3.55 | Infinity | 35.81 | 6 | 8.15 | Infinity | 6.59 |
| Loop | 1.44 | Infinity | Infinity | 2.62 | 3.69 | Infinity | Infinity |
| Math | 3.49 | 3.09 | 2.49 | 11.64 | 3.82 | 3.85 | 12.6 |
| Method | 2.73 | 392.39 | 388.01 | 3.38 | 8.75 | 1301.76 | 23.69 |

Table 2 Comparison of JGF numbers from the results of section 1 benchmarks. The numbers are JGF numbers for each benchmark within Section 1. (Sun32(64)C(S) = Sun HotSpot Client(Server) VM in 32(64)-bit mode running on Sun Fire E15000, IBM32 = IBM Classic VM in 32-bit mode running on IBM HPCx, PC_C(S) = Sun HotSpot Client(Server) VM running on PC, PC_I = IBM Classic VM running on PC)

As shown in Table 2, different JVMs are tuned for different Java operations. Deriving an overall performance figure is not practical due to the large unrealistic numbers achieved in some of the benchmark results. Hence, Section 1 of the JGFBS does not allow us to conduct an overall comparison of the different Java platforms. However, the results from Section 1 are useful for predicting the Java performance of real applications that utilizes such operations.

1.4.2 Kernel Operations Performance

Figure 2 shows the JGF numbers for the Section 2 benchmarks on the various test platforms. It is a detailed view of Figure 1. Java performance characteristics are diverse across the execution environments. Sun Java platforms show a consistent behavior for

most of benchmarks within Section 2 while IBM platform demonstrates considerable variations in performance across the benchmarks. Performance on the PC environment is even more interesting.

For the IBM platform, the Crypt performance is poor in comparison to the other execution environments. The performance figures for the same benchmark are highest for the PC Java platform. In general, the PC shows better performance for integer arithmetic operations than Sun and IBM platforms (see Section 1.4.1). Since the majority of the Crypt benchmark consists of integer addition, multiplication, and division, it is perhaps not surprising that the Crypt benchmark performance is higher on the PC platform.

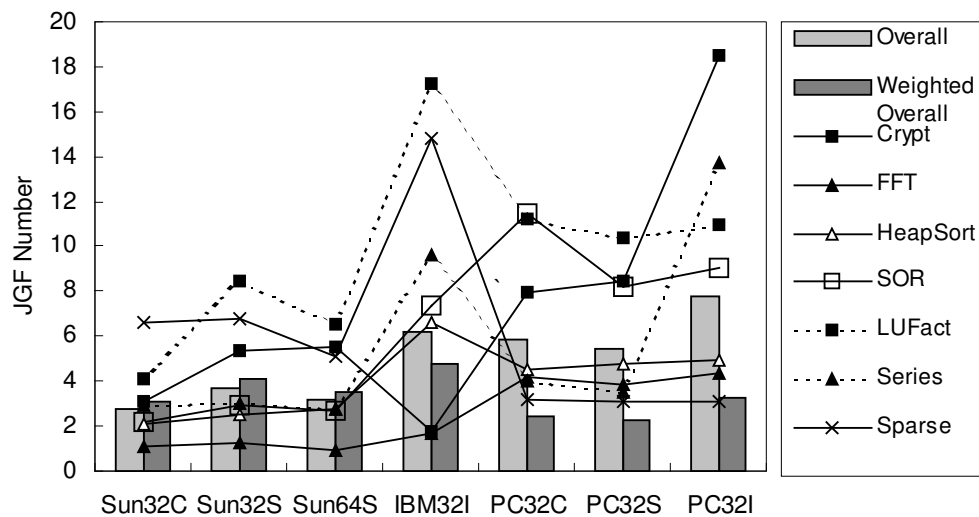


Figure 2 Benchmark results for Section II (Size B) on various Java execution environments: Comparison by benchmarks.

The execution time for the LUFact benchmark is shortest on the IBM platform. The IBM Windows VM, which runs on a processor of higher clock rate, has significantly lower performance for the benchmark than the IBM AIX VM. In addition, the IBM AIX JVM has the highest performance for the Series benchmark, whose performance is 3 times higher than the performance on the Sun environments. RISC-processor platforms seem to have high performance for the Sparse benchmark.

Figure 3 shows a different view of the same data used in Figure 2 and highlights the performance characteristics of each Java platform across the set of section 2 benchmarks. There are many striking performance characteristics highlighted in Figure 3. Firstly, the performance behaviors of the Sun Solaris JVMs are similar to that of the IBM AIX JVM except for the Crypt benchmark. These JVMs run under a native environment of their own vendors. Secondly, the PC platform scores the highest JGF number for the Crypt benchmark but scores the lowest performance figure for the Sparse benchmark. Thirdly,

comparison between the IBM platform and the PC platform shows that the IBM SDK is optimized for the type of application represented by the Series benchmark. Finally, comparison of the Sun platform with the PC shows that the Sun platform is not efficient for running the SOR benchmark.

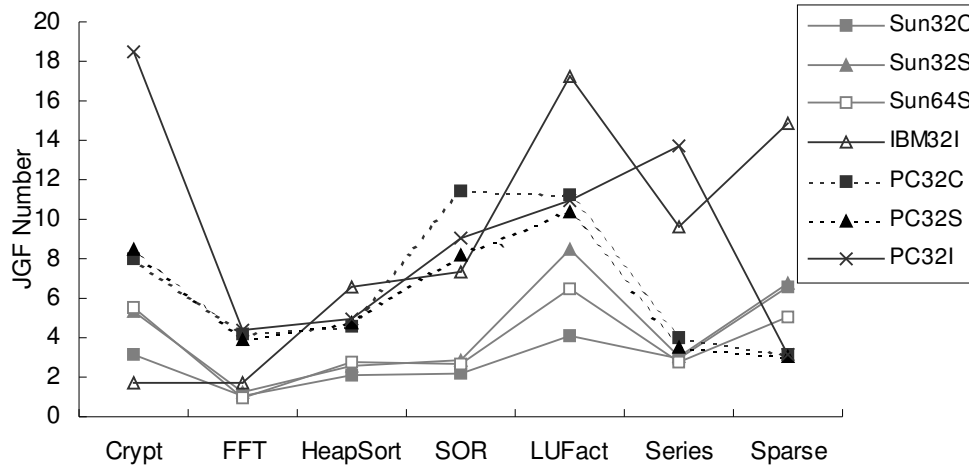


Figure 3 Benchmark results for Section II (Size B) on various Java execution environments: Comparison by platforms.

1.4.3 Large Scale Application Performance

Figure 4 shows that the performance of the RayTracer and MonteCarlo benchmarks is reasonably similar across execution environments, while the performance of the MolDyn benchmark varies significantly. The performance characteristics of the MolDyn benchmark, together with the MonteCarlo benchmark, have made a significant contribution to the overall performance figure as shown in Figure 4. The geometric mean has the property that the mean value is most affected by the smallest factor. Moreover, the MolDyn benchmark on the Windows platform seems to go against the processor clock speed. As a result, the overall trend of effective performance figures appears to coincide with that of the MolDyn benchmark JGF number.

Another point that is clear from the performance characteristics of the MolDyn benchmark is that the benchmark fails to predict computing performance for various Java execution environments. A simple consideration of hardware and compiler configurations does not pose a sensible understanding of the performance behavior on the PC.

The IBM AIX VM (IBM32) has the largest variation in performance and the 32 bit HotSpot Client VM on the Sun platform (Sun32C) has the smallest performance variation across the section 3 benchmarks. IBM Windows VM draws a most peculiar performance behavior in Section 3 because of the MolDyn benchmark run as shown in Figure 5.

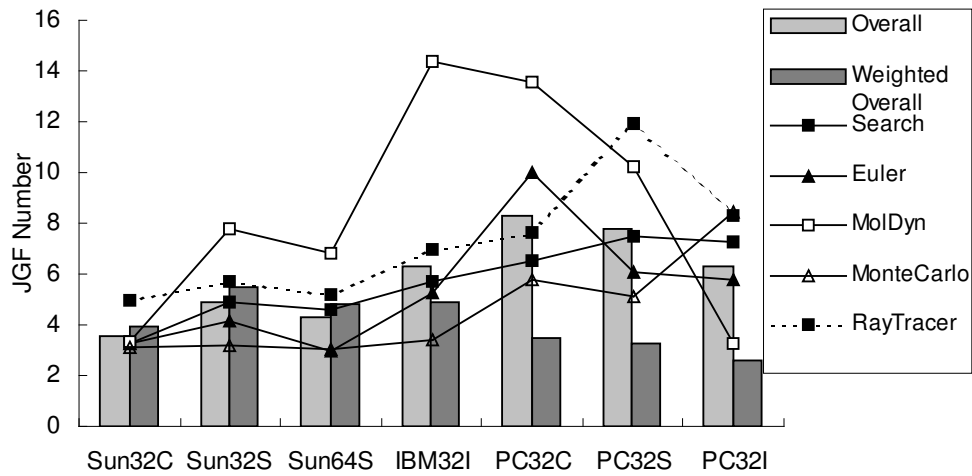


Figure 4 Benchmark results for Section III (Size A) various Java platforms: Comparison by benchmarks.

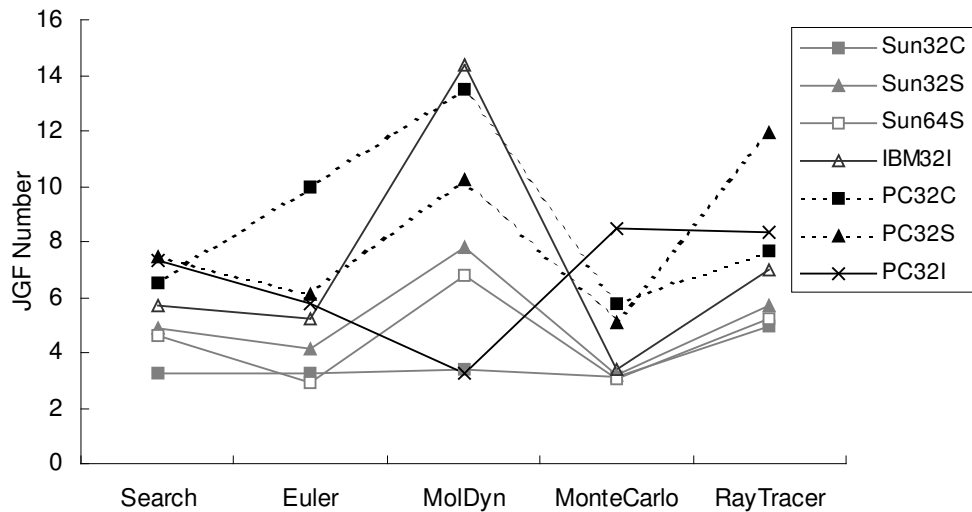


Figure 5 Benchmark results for Section III (Size A) on various Java platforms: Comparison by platforms.

1.5 Summary

Overall performance increases with processor speed. However this fails to give a true picture of Java performance across these platforms, as performance varies significantly across the benchmarks. Relatively high performance is observed for many of the low level operations, reflecting the development of Just in Time compilers across many of the JVMs in recent years. The larger benchmarks show a diverse range of performance

characteristics. The performance of some of these can be explained by the performance of the low-level operations (for example the Crypt benchmark), although this is not applicable across all benchmarks.

2. Language Performance

In the previous section we compared the performance of different Java execution environments. In this section, we have tested the JGF Language Comparison Benchmarks (JGFLCB), which are a subset of the JGF sequential benchmarks written in C, on the same platforms. There are some advantages as well as practical difficulties in these kind of comparisons, as C and Java have different design philosophies, programming styles and language constructs.

| | C Execution Options | Java Execution Options |
|--|---|--|
| Sun Fire 15k: SunOS 5.9 0.9GHz UltraSPARC III | CC -fast -qarch=v8plusb CC -fast -qarch=v9plus | Sun SDK 1.4.1_02 -O -client Sun SDK 1.4.1_02 -O -server |
| IBM HPCx: AIX 4.3 1.3GHz Power4 | XLC -O3 -q32 XLC -O3 -q64 | IBM SDK 1.3.0 -O |
| PC: Windows 2000 Prof. 2.4GHz Pentium4 | GCC 3.2.5 -O (GNU-make 3.8) | Sun SDK 1.4.1_02 -O -client Sun SDK 1.4.1_02 -O -server IBM SDK 1.3.0 -O |

Table 3 C and Java optimization options of compilation and execution on the test platforms.

Table 3 summarizes the compile and run-time options used on each test platform, while Figure 6 shows the Section 2 C benchmark results for these systems. While it is clear that the relative performance of the different benchmarks varies across the platforms, the IBM system generally outperforms the PC, which in turn outperforms the Sun. Hence unlike the Java results, performance does not increase with processor speed. This is likely due to the generic nature of the gcc compiler, compared with the vendor optimized xlc and cc compilers.

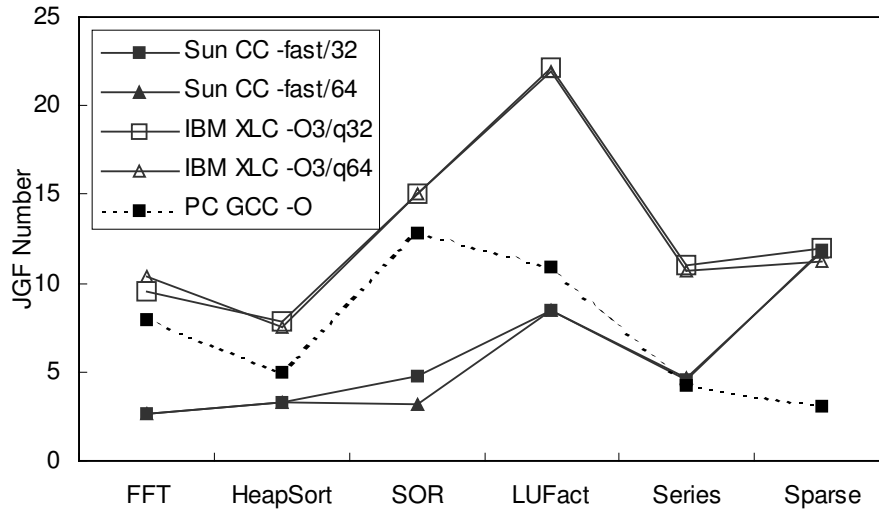


Figure 6 Benchmark results for Section II (Size B) on various C execution environments.

2.1 Language Comparison on the Sun Platform

On the Sun system, we tested two C execution environments together with three Java execution environments. These results are shown in Figure 7. In general, the Sun C environments outperform the Sun Java environments for the benchmarks by less than a factor of three.

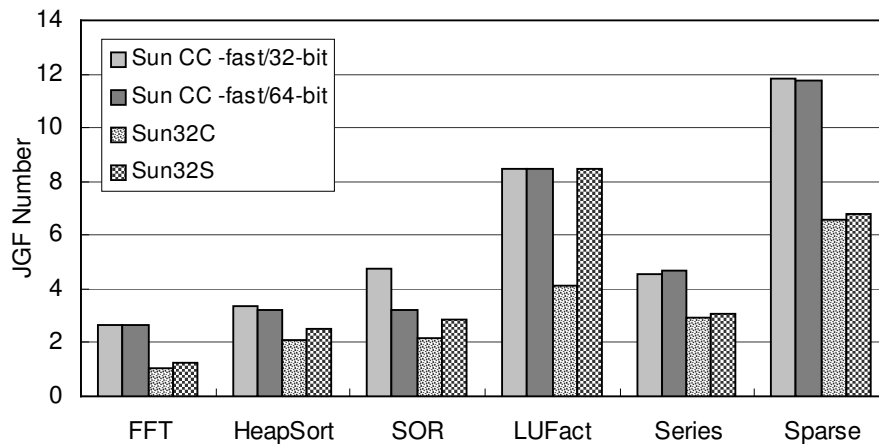


Figure 7 Performance comparison of Sun execution environments for Section II (Size B) benchmarks: Comparison by platforms.

One of the interesting results is that “performance reversal” is observed between C and Java executions for certain benchmarks, as shown in Figure 8 and 9. For instance, while the Sun C environment shows better performance for the Sparse benchmark than for the LUFact benchmark and higher performance for the Euler benchmark than for the MolDyn

benchmark, the Sun Java environment shows better performance for the LUFact and MolDyn benchmarks than for the Sparse and Euler benchmark, respectively. Understanding of this performance behavior requires further experiments and study.

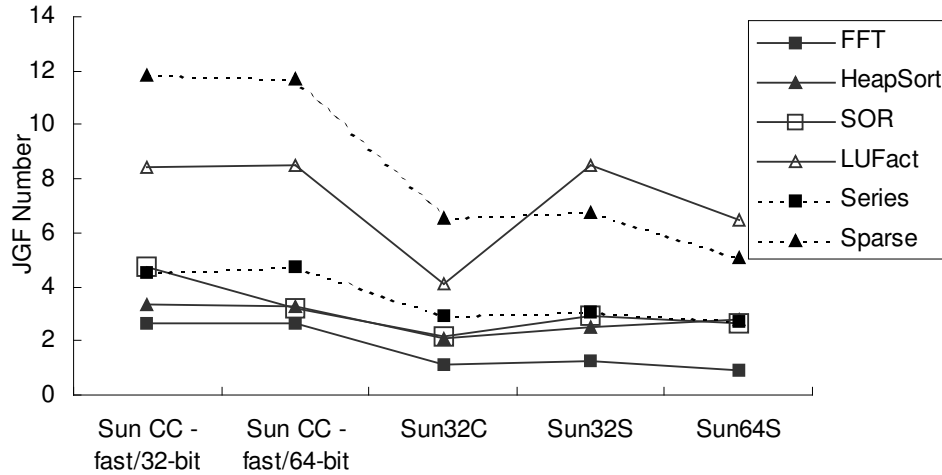


Figure 8 Performance comparison of Sun execution environments for Section II (Size B) benchmarks: Comparison by benchmarks.

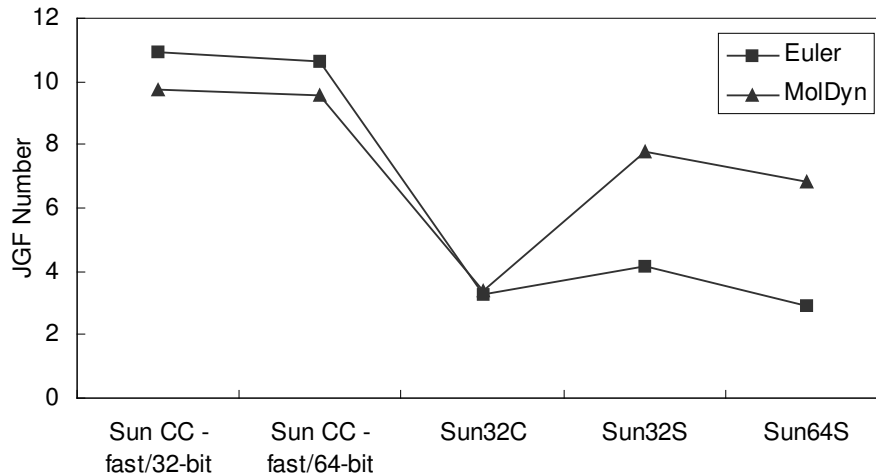


Figure 9 Performance comparison of Sun execution environments for Section III (Size A) benchmarks: Comparison by benchmarks.

2.2 Language Comparison on the IBM Platform

Performance results for the IBM platform are shown in Figure 10. Java performance is good for the HeapSort and Series benchmarks when compared to C on the IBM system, but drops considerably for the FFT benchmark (by a factor of 6).

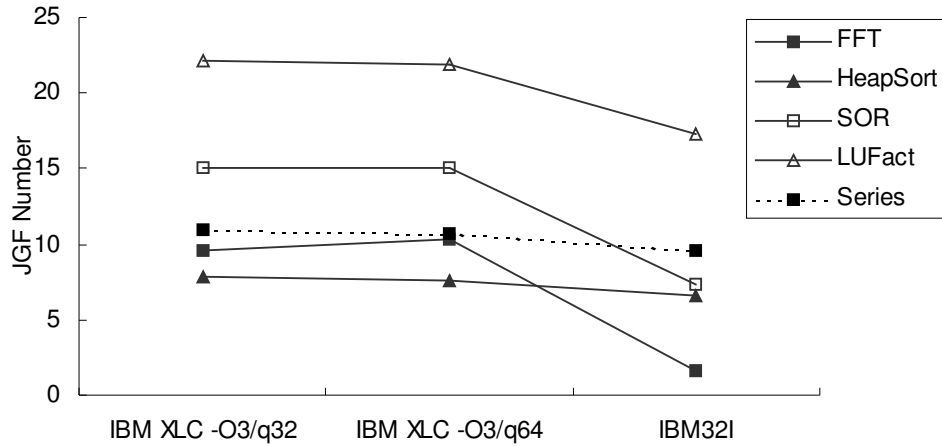


Figure 10 Performance comparison of IBM execution environments for Section II (Size B) benchmarks: view 1.

There is a “performance contrast” between the Euler and MolDyn benchmarks, shown in Figure 11. We see the performance drop for the MolDyn by less than a factor of 1 for C to Java, while the Euler benchmark by more than a factor of 5.

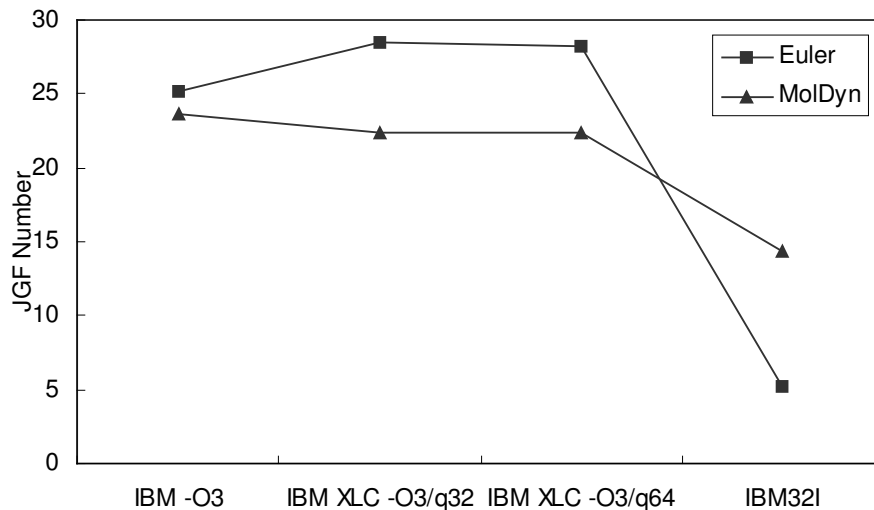


Figure 11 Performance comparison of IBM execution environments for Section III (Size A) benchmarks: View 1.

2.3 Language Comparison on the PC Platform

In the previous sections, the SOR, LUFact, and Sparse benchmarks were important in gaining meaningful insight into the performance characteristics for language comparison. Here we observe another type of performance characteristic from the Series benchmark. Performance figures for the Series benchmark rise from around 4 to 13.75 for the IBM SDK 1.3.0 Java execution environment, which is quite unusual.

In addition, we observe multiple performance reversals between C and Java executions, as shown in Figure 12. Java shows a diverse behavior across different compilers and VMs for the given benchmarks.

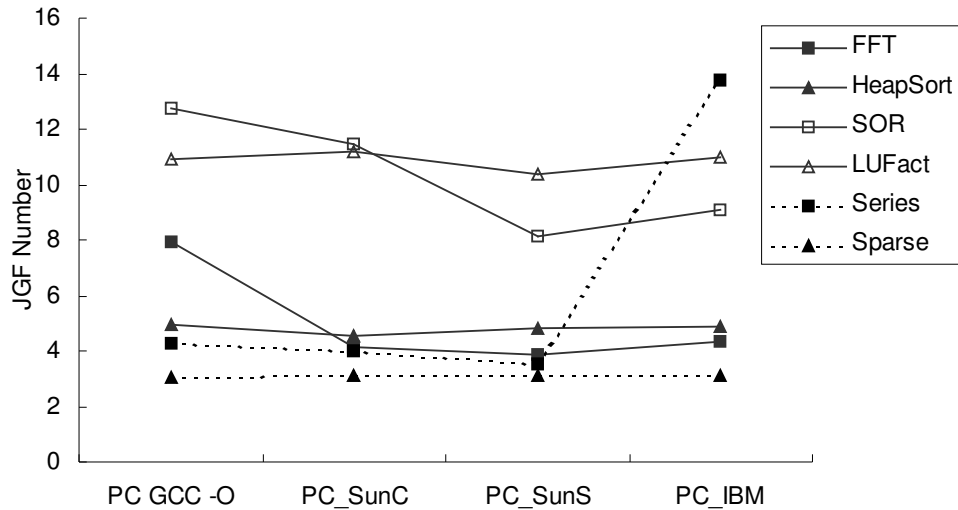


Figure 12 Performance comparison of Java and C on PC for Section II (Size B): view 1.

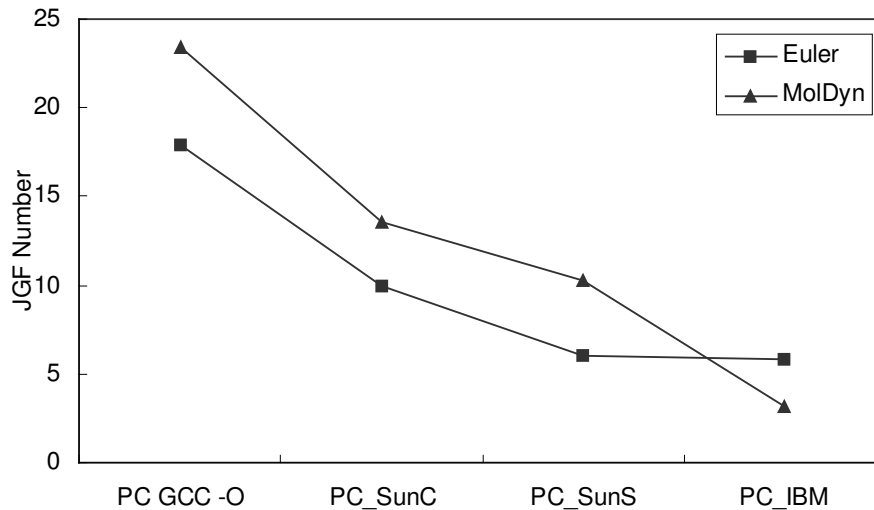


Figure 13 Performance comparison of Java and C on PC for Section III (Size A): view 1.

Figure 13 shows that performance reversal occurs only for the IBM Java execution environment, which is quite unusual if we consider Java performance characteristics on the Sun native environment. Also, the IBM Java execution environment on the PC shows higher performance on the Euler benchmark rather than on the MolDyn benchmark.

2.4 Summary

For the larger application codes, C shows substantially better performance than Java, by a factor of 3 to 5. However comparing and contrasting the performance of the kernel benchmarks shows that Java performance is, at the most, only a factor of 2 slower than C performance, with the exception of the FFT benchmark on the IBM platform.

3. Conclusions

As mentioned previously, Java offers a number of benefits as a language for High Performance Computing (HPC), especially in the context of the Computational Grid. There are however a number of issues surrounding the use of Java for HPC, one of these being performance. In this document we have evaluated the performance of different Java execution environments and compared and contrasted this performance with equivalent benchmarks in C. While the performance of Java is still poorer than that of C, the gap has closed substantially in recent years and in a minority of cases this performance gap is small enough to allow Java to be considered as a real alternative to C for HPC.

4. Bibliography

- [1] J. M. Bull, L. A. Smith, C. Ball, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. *Concurrency and Computation: Practice and Experience*, Volume 15, Issue 3-5, 2003, pages 417-430
- [2] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12, 2000, pages 375-388.
- [3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. Benchmarking Java Grande Applications. In *Proceedings of the Second International Conference on the Practical Applications of Java*, pages 63-73, Manchester, U.K., April 2000.
- [4] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 106-115, Palo Alto, CA, USA, June 2001.

- [5] JGF, The Java Grande Forum. See: <http://www.javagrande.org>
- [6] The Java HotSpot Virtual Machine, Technical White Paper, September 2002. See: <http://java.sun.com/products/hotspot/docs/whitepaper/>
- [7] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and Development of Java Grande Benchmarks. In Proceedings of the ACM 1999 Conference on Java Grande, pages 72-80, Palo Alto, CA, June 1999.
- [8] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, 2nd Edition. Morgan Kaufmann, 1998.
- [9] R. H. Saavedra and A. J. Smith. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. In ACM Transactions on Computer Systems (TOCS), Volume 14 Issue 4, November 1996, pages 344-384
- [10] X. Zhang and M. Seltzer. HBench:Java: - An Application-Specific Benchmarking Framework for Evaluating Java Virtual Machines. In Proceedings of the ACM 2000 conference on Java Grande, pages 62-70, San Francisco, CA, June 2
- [11] J. Hein, Java Grande language comparison benchmarks on the IBM p690 system, UKHEC Technical Report, See: http://www.ukhec.ac.uk/publications/reports/hpc_java_comp.pdf