Benchmarking Java against C and Fortran for Scientific Applications

J. M. Bull, L. A. Smith, L. Pottage and R. Freeman
Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings,
The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
epcc-javagrande@epcc.ed.ac.uk

ABSTRACT

Increasing interest is being shown in the use of Java for scientific applications. The Java Grande benchmark suite [4] was designed with such applications primarily in mind. The perceived lack of performance of Java still deters many potential users, despite recent advances in just-in-time (JIT) and adaptive compilers. There are however few benchmark results available comparing Java to more traditional languages such as C and Fortran. To address this issue, a subset of the Java Grande Benchmarks have been re-written in C and Fortran allowing direct performance comparisons between the three languages. The performance of a range of Java execution environments, C and Fortran compilers have been tested across a number of platforms using the suite. These demonstrate that on some platforms (notably Intel Pentium) the performance gap is now quite small.

Keywords

Java, C, Fortran, performance, benchmarking, scientific applications

1. INTRODUCTION

Java has a number of important features which make it an attractive language for scientific applications. Perhaps the most important of these is portability—despite standardisation efforts, the process of creating truly portable Fortran or C programs is time consuming and requires considerable experience. Portability is especially important for high performance applications, where the hardware architecture typically has a much shorter lifespan than the application software. Java also avoids the need for complex makefiles and configure scripts. Portability may be of even greater importance in the context of the computational grid, where the target architecture could be unknown to the user.

Java is also a highly network-centric language with extensive support for, amongst others, remote method invocation, remote file access and database access. Such facilities are important for applications with remote visualisation or computational steering requirements.

Java is also generally considered to offer a better software engineering environment than C or Fortran. Features such as the absence of pointers, automatic garbage collection and strict type checking allow rapid prototyping, and lead to less buggy code and faster development times.

The nature of many scientific applications makes them well suited to Java execution environments. They typically spend a large amount of execution time in a small number of user-written methods, making them good candidates for JIT compilation, and less susceptible than other applications to poor implementations of the Java API. Scientific applications, when written in a traditional, non-objected oriented manner, often have large and persistent data structures, resulting in low garbage collection overheads.

Java is also rapidly becoming the language of choice for many mainstream and commercial applications, as well as being a very popular teaching language in many institutions. This popularity has led the major vendors to expend significant resources on developing robust and efficient Java execution environments (far more than, say, on Fortran compilers).

On the other hand, Java still suffers from some significant disadvantages compared to more traditional languages. The core Java language still lacks some features which scientific programmers find attractive and useful. These include complex numbers as a basic type, multidimensional arrays, generics, and operator overloading. The Java Grande Forum [13] is actively addressing these shortcomings, but whether they will be adopted in the language specification remains to be seen.

Another shortcoming of Java for high performance applications is the absence of the familiar parallel programming models such as MPI and OpenMP, available to C and Fortran programmers. Despite research efforts both in message passing [2] and shared memory directives [14], standardisation is still some way off.

Perhaps the most important of Java's shortcomings, at least in terms of users' perceptions, is performance. Direct comparisons are not easy to make, and have not been widely publicised. The benchmarking effort described in this paper

is intended to address precisely this issue, and may serve to dispel some of the myths about lack of performance of Java codes.

The Java Grande benchmark suite has been run on a range of Java execution environments and systems. While this provides valuable information about the relative merits of Java implementations, it cannot answer the question of how much performance (if any) will be sacrificed by abandoning C or Fortran in favour of Java. To address this issue, a subset of the benchmarks have been translated into C and Fortran, allowing direct performance comparisons to be carried out. While object-oriented design and programming holds significant potential for the future of scientific programming, this is not our principal focus. Comparing a highly object based Java implementation with non-object-oriented C or Fortran would be hard to interpret, as we cannot readily distinguish between differences due to language implementations and differences due to abstraction penalties. Since the latter have been studied elsewhere, it is more appropriate to separate the two issues.

The remainder of this paper is structured as follows: in Section 2 we review related work, while in Section 3 we describe the structure and contents of the language comparison benchmarks. In Section 4 we describe the systems and environments tested, and present and discuss the results obtained. Section 5 summarises and provides some conclusions.

2. RELATED WORK

A considerable number of benchmarks and performance tests for Java have been devised. Some of these consist of small applets with relatively light computational load, designed mainly for testing JVMs embedded in browsers—these are of little relevance to Grande applications. Of more interest are a number of benchmarks [3, 8, 9, 20, 16] which focus on determining the performance of basic operations such as arithmetic, method calls, object creation and variable accesses. These are useful for highlighting differences between Java environments, but give little useful information about the likely performance of large application codes. They are also highly vulnerable to JIT warm-up effects (it is difficult to be certain whether interpreted or compiler code is being measured) and to optimisations which make use of run-time information (for example division by one).

Other sets of benchmarks, from both academic [7, 18] and commercial [17, 21] sources, consist primarily of computational kernels, both numeric and non-numeric. This type of benchmark is more reflective of application performance, though many of the kernels in these benchmarks are on the small side, both in terms of execution time and memory requirements. Finally there are some benchmarks [6, 12, 22] which consist of a single, near full-scale, application. These are useful in that they can be representative of real codes, but it is virtually impossible to say why performance differs from one environment to another, only that it does.

Few benchmark codes attempt inter-language comparison. In those that do, (for example [20, 23]) the second language is usually C++, and the intention is principally to compare the object oriented features, rather than basic memory and arithmetic operations. The Scimark 2.0 suite citeBull:pozo (from which some of the Java Grande benchmarks are derived) has a C version, though no results have been published.

3. THE BENCHMARK SUITE

The aim of the Java Grande benchmark suite is to provide a standard benchmark suite that can be used to:

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

We adopt the structure of the GENESIS Benchmark suite [1], providing three types of benchmark: low-level operations (which we refer to as Section I of the suite), simple kernels (Section II) and applications (Section III) within a single suite. A subset of these benchmarks have been rewritten in C and Fortran to allow inter-language comparisons. These benchmarks are described here. For further details of the complete set of benchmarks and the rational for their design see [4] or `www.epcc.ed.ac.uk/javagrande`.

A range of data sizes for each benchmark in Sections II and III is provided to avoid dependence on particular data sizes (sizes A, B and C for Section II and sizes A and B for Section III). To remove any ambiguity in the question of what is being tested the source code for all the benchmarks is distributed.

For the language comparison benchmarks, we omit Section I for the following reasons: many do not have suitable direct translations into C or Fortran, while those that do (such as arithmetic operations) have been found to be too sensitive to compiler optimisation to give meaningful results. The language comparison benchmark suite is therefore structured as follows:

Section II: Kernels

Series Computes the first N Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval 0,2.

LUFact Solves an $N \times N$ linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark [7].

HeapSort Sorts an array of N integers using a heap sort algorithm.

SOR The SOR benchmark performs 100 iterations of successive over-relaxation on an $N \times N$ grid.

FFT This performs a one-dimensional forward transform of N complex numbers.

SparseMatmult Performs matrix-vector multiplication using an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure.

Section III: Applications

Euler Solves the time-dependent Euler equations for flow in a channel with a "bump" on one of the walls.

MolDyn A simple N-body code modelling the behaviour of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions.

Of these, all have C versions while LUFact and Moldyn have also been translated into Fortran. For the C versions, we have attempted to keep as close as possible in terms of syntax to the Java code: this is possible thanks to the strong similarities between much of the basic syntax of the two languages. Indeed in many cases the computationally intensive loops are syntactically identical. The Fortran versions are of necessity somewhat less closely related to the Java source code. For example, in the Java version of MolDyn, each particle is represented by an object. In the C version, it is represented by a struct, but in the Fortran version the data for each particle is simply a series of entries (at the same index) in a number of arrays.

Performance metrics for the benchmarks are provided in three forms: execution time, temporal performance and relative performance (see [4]). However, in this paper we simply report execution time, the wall clock time required to execute the benchmark, excluding initialisation, validation and clean-up phases. For the Java versions we use the `System.currentTimeMillis()` method. For the Fortran and C versions there is no fully portable timing routine, so we use system specific high-resolution timers.

4. RESULTS

The benchmark suite has been tested on a range of execution environments on the following platforms: a 700MHz Pentium III with 256 Mb of RAM, running Windows NT 4.0; a 700MHz Pentium III with 256 Mb of RAM, running Linux 6.2; a 300 MHz Sun UltraSparc II with 1Gb RAM; a Compaq ES40, (500 MHz Alpha EV6) with 4Gb of RAM, running Digital UNIX V4.0F. The Java execution environments, C and Fortran compilers studied are summarised in table 1. The flags shown are compile time flags for C and Fortran and runtime flags for Java. All Java compilation was performed using `javac -O`. For the C and Fortran compilers, a standard set of optimisation flags was chosen for all the benchmarks—no attempt was made to tune the flags for individual codes. The results reported are, in all cases, the best time obtained over three runs of the code. Data size B was used for Section II codes, and data size A for Section III.

It is worth noting that Sun significantly altered the structure of their JDK between versions 1.2 and 1.3. The production 1.2 JDK contained the Classic Virtual Machine, while the HotSpot Engine was available as a plug-in to the reference version. In the 1.3 JDK the Hotspot Engine has replaced the Classic VM, and has two modes available: Client and Server.

Table 2 gives the execution timings for all the benchmarks. The execution times are represented graphically in Figures 1- 8. Table 3 shows the mean ratio (over benchmarks) of execution times for every Java environment to every C or Fortran compiler. Since some runs failed, the number of benchmarks used to compute the mean is also shown. Table 4 shows, for each benchmark, the ratio of the fastest Java execution time to the fastest C or Fortran execution time for the given hardware platform. This comparison seems reasonable, since there is very little effort or expense involved in downloading a number of Java environments for a given platform, and choosing the best for a given application.

4.1 Intel Pentium, Windows NT

On the Pentium III NT platform we tested six Java environments. No one of these was a clear winner on all the benchmarks. The IBM 1.2 and 1.3 JDK's give very similar performance, with one or other giving the fastest time on all the codes except FFT and MolDyn. There is also little to chose between the HotSpot Client and HotSpot Server modes of the Sun 1.3 JDK. Although there are differences on individual benchmarks, neither version can be considered better than the other. In general, the Sun 1.2 JDK (with the Classic VM) performs better than the Sun 1.3 (in either mode), so for scientific applications, it seems that the move to HotSpot has been a retrograde step. The Microsoft JDK performs moderately well across all the codes, being rarely either the fastest or the slowest Java environment.

Comparisons with C are very reasonable on this system. For example, for the Sun 1.2, Microsoft and IBM JDKs the overall performance loss is less than 10% compared to Borland C++ and less than 60% compared to Portland Group cc. The ratio of the best Java to best C execution time has a mean of 1.23 and exceeds 2.0 only in the case of the FFT benchmark.

Java also fares well compared to Fortran on this system, especially with respected to the (somewhat dated) version of Digital Fortran. Comparing the best Java to best Fortran execution time shows an increase of less than 25%.

4.2 Intel Pentium, Linux

We tested four Java environments under Linux. Of these, the IBM 1.3 JDK gave the best performance on all benchmarks except MolDyn. In almost all cases the execution times are less than for the NT version of the same JDK. The Blackdown 1.3 and the two versions of Sun 1.3 were roughly comparable overall, though there are significant differences on individual benchmarks.

Comparisons with C compilers on the system are very favourable. The IBM 1.3 JDK is on average slightly faster than KAI C++, and only 15% slower than gcc. The mean ratio of fastest Java to fastest C execution times is only 1.07. At this level of difference there is no case for preferring C to Java on grounds of performance.

Comparisons with Fortran are not quite as impressive, as none of the Java environments comes close to the perfor-

Pentium III, NT	**Pentium III, Linux**
Sun JDK 1.2.2_006	Sun JDK 1.3.0 (-client)
Sun JDK 1.3.0 (-client)	Sun JDK 1.3.0 (-server)
Sun JDK 1.3.0 (-server)	Blackdown JDK 1.3
IBM JDK 1.2.0	IBM JDK 1.3.0
IBM JDK 1.3.0	gcc 2.91.66 (-O3 -funroll-loops)
Microsoft SDK for Java 4.0	KAI C++ v4.0b (+K3 -O)
Borland C++ 5.5.1 (-5 -O2 -OS)	g77 2.91.66 (-O3 -funroll-loops)
Portland Group pgcc 3.2-3 (-fast)	pg77 3.1-2 (-fast)
Portland Group pgf90 3.2-3 (-fast)	
Digital Fortran V5.0 (-fast)	
Sun UltraSparc II	**Compaq ES40**
Sun JDK 1.2.1 (-Xoptimize)	Compaq Java 1.3.0-alpha1
Sun JDK 1.3.0, HotSpot Client	Dec C V5.9-005 (-fast -O4 -ieee -tuneev6 -archev6)
Sun JDK 1.3.0, HotSpot Server	Compaq Fortran V5.3-1120 (-fast -O4 -ieee -tuneev6 -archev6)
LaTTe 0.9.1	
Sun WS 6 cc 5.2 (-fast -xarch=v8plusa)	
gcc 2.95.2 (-O3 -funroll-loops)	
Sun WS 6 f90 95.6.1 (-fast -xarch=v8plusa)	
g77 2.95.2 (-O3 -funroll-loops)	

Table 1: Tested Java execution environments, C and Fortran compilers (with flags)

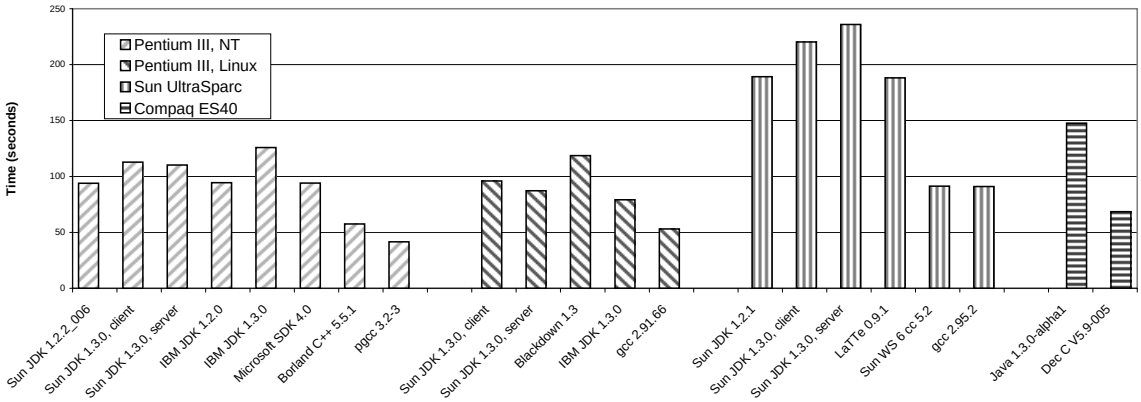

Figure 1: Execution time for the FFT benchmark

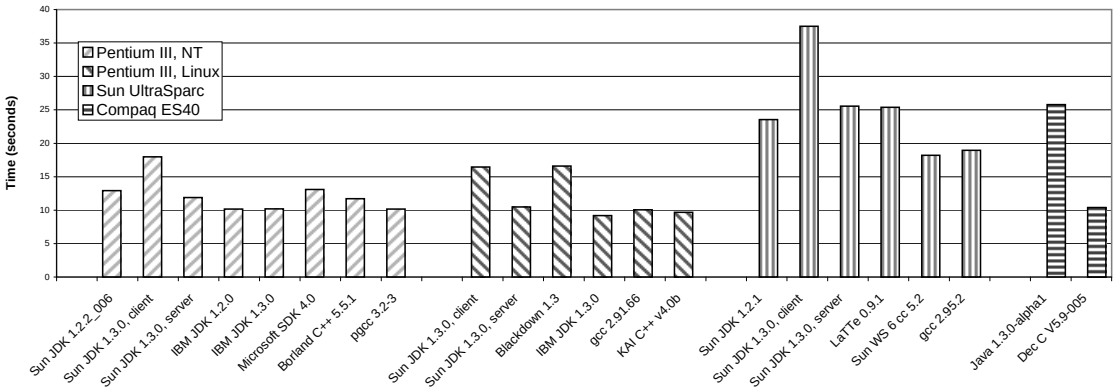

Figure 2: Execution time for the HeapSort benchmark

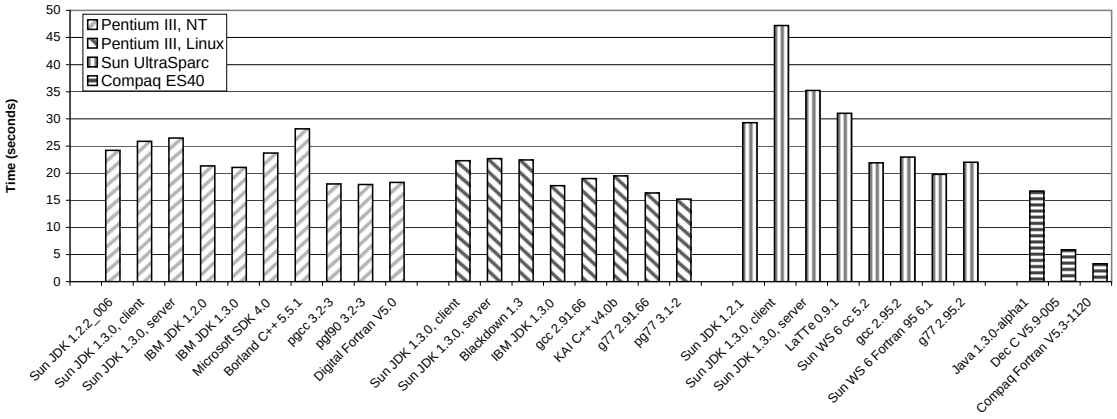

Figure 3: Execution time for the LUFact benchmark

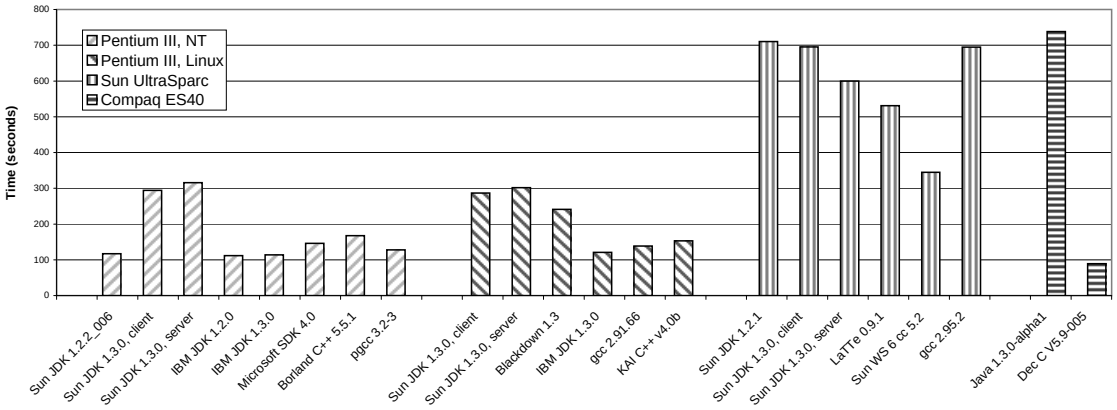

Figure 4: Execution time for the Series benchmark

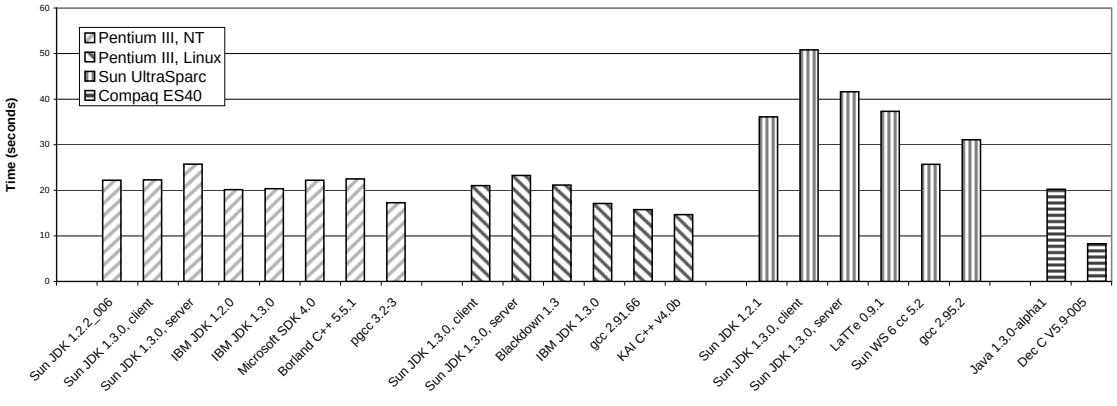

Figure 5: Execution time for the SOR benchmark

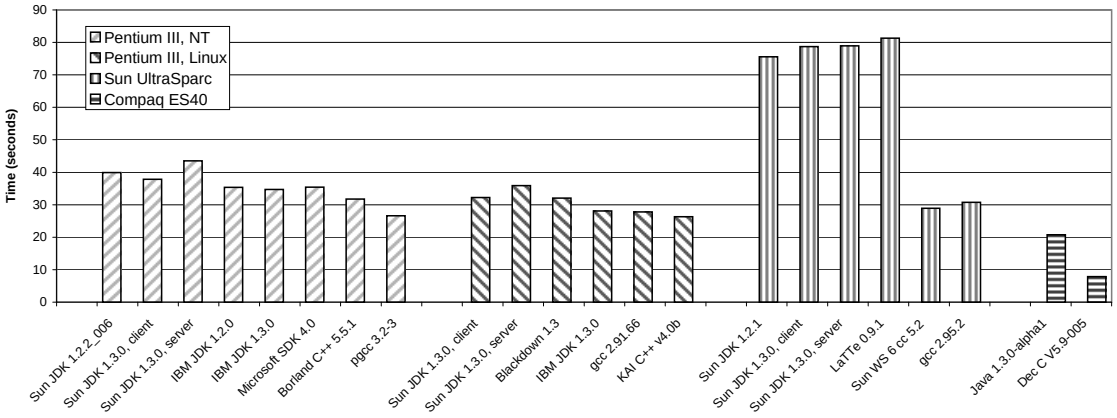

Figure 6: Execution time for the SparseMatmult benchmark

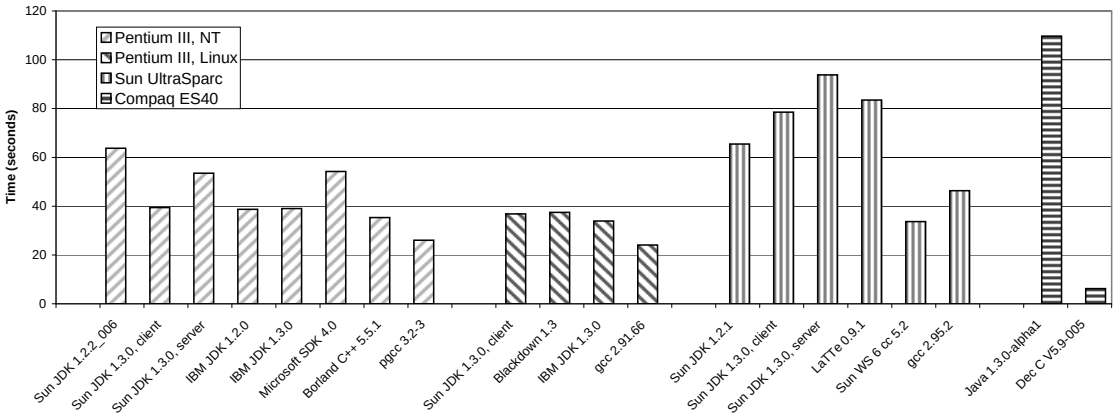

Figure 7: Execution time for the Euler benchmark

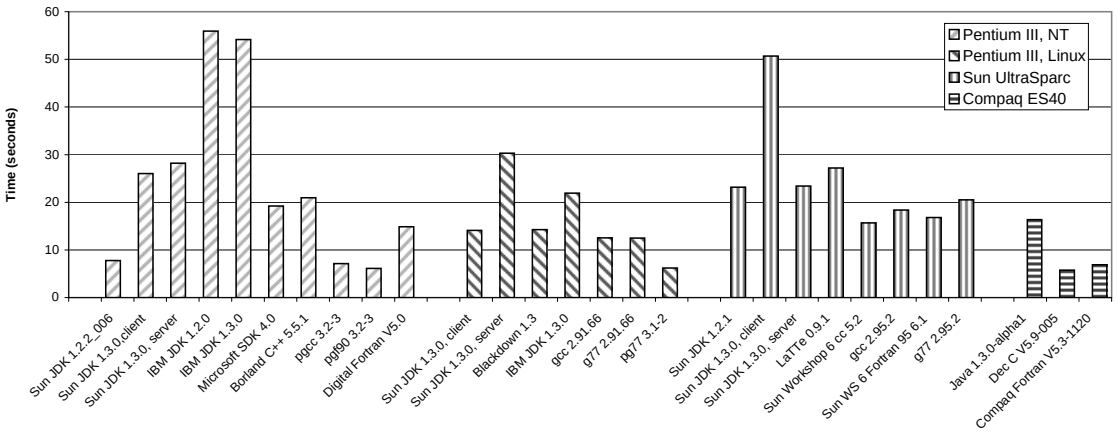

Figure 8: Execution time for the MolDyn benchmark

Environment	Benchmarks							
	FFT	HeapSort	SOR	LUFact	Series	SparseMatmult	Euler	MolDyn
Pentium III, 700 MHz, NT								
Sun JDK 1.2.2_006	93.9	12.9	22.2	24.2	117	39.9	63.7	7.80
Sun JDK 1.3.0, client	113	18.0	22.3	25.9	294	37.9	39.4	26.0
Sun JDK 1.3.0, server	110	11.9	25.7	26.5	315	43.5	53.5	28.2
IBM JDK 1.2.0	94.4	10.2	20.2	21.3	111	35.4	38.7	55.9
IBM JDK 1.3.0	126	10.2	20.3	21.1	114	34.7	39.1	54.2
Microsoft SDK for Java 4.0	94.1	13.1	22.2	23.7	146	35.4	54.2	19.2
Borland C++ 5.5.1 for Win32	57.5	11.7	22.5	28.2	167	31.7	35.4	21.0
Portland Group cc	41.6	10.2	17.3	18.0	127	26.6	26.1	7.13
pgf90 3.2-3				17.9				6.13
pgf77 3.2-3				19.0				6.00
Digital Fortran V5.0				18.3				14.9
Pentium III, 700 MHz, Linux								
Sun JDK 1.3.0, client	96.0	16.5	21.0	22.3	287	32.2	36.8	14.1
Sun JDK 1.3.0, server	87.3	10.5	23.3	22.7	301	35.9	F	30.3
Blackdown 1.3	119	16.6	21.1	22.5	241	32.1	37.5	14.3
IBM JDK 1.3.0	79.1	9.18	17.1	17.7	121	28.1	33.9	21.9
gcc 2.91.66	53.0	10.1	15.8	19.0	139	27.8	24.1	12.5
KAI C++ v4.0b	F	9.68	14.7	19.5	153	26.4	F	F
g77 2.91.66				16.3				12.5
pg77 3.1-2				15.2				6.21
Sun UltraSparc, 300MHz								
Sun JDK 1.2.1	189	23.5	36.1	29.3	710	75.6	65.5	23.2
Sun JDK 1.3.0, client	220	37.5	50.8	47.2	695	78.7	78.5	50.7
Sun JDK 1.3.0, server	236	25.6	41.6	35.2	600	79.0	93.8	23.4
LaTTe 0.9.1	188	25.4	37.4	31.0	531	81.3	83.5	27.2
Sun Workshop 6 cc 5.2	91.3	18.2	25.7	21.9	345	28.9	33.7	15.7
gcc 2.95.2	91.0	18.9	31.1	23.0	695	30.7	46.4	18.4
Sun Workshop 6 Fortran 95 6.1				19.8				16.8
g77 2.95.2				22.0				20.5
Compaq ES40, 500MHz Alpha EV6								
Compaq Java 1.3.0-alpha1	148	25.8	20.2	16.7	738	20.8	110	16.3
Dec C V5.9-005	68.6	10.4	8.26	5.88	89.0	7.86	6.24	5.75
Compaq Fortran V5.3-1120				3.29				6.88

Table 2: Benchmark results (seconds) for various Java execution environments. Benchmarks which failed to execute correctly are denoted by F.

mance of the Portland Group compiler on MolDyn (unlike the Sun 1.2 JDK under NT!). On LUFact, however the best Java execution time is within 20% of the best Fortran.

4.3 Sun Ultrasparc

On this platform we also tested four Java environments. We observe that the Sun 1.3 JDK is generally (but not always) faster in Server mode than in Client mode, but that performance in either mode is often worse than the Sun 1.2 JDK. The overall performance of the LaTTe VM lies between that of the Sun 1.2 and 1.3 versions.

Comparison with the C compilers shows a wider gap on the Ultrasparc platform than on the Pentium. The Sun 1.2 JDK is, on average, 1.43 times slower than gcc and 1.72 times slower than the Sun WorkShop 6 C compiler. Taking the best Java execution time and comparing to the fastest C execution time, we observe a mean ratio of 1.61, with a range from 1.29 (HeapSort) to 2.61 (SparseMatmult).

The differences between the C and Fortran execution times on the Sun are small, so very similar observations to the above apply when comparing Java and Fortran.

4.4 Compaq Alpha

On this platform we tested only the vendor supplied Java, C and Fortran environments. We observe a significant gap between Java and C performance, with execution time ratios varying from 2.15 (FFT) to 17.6 (Euler), with a mean of just over 4. The comparison with Fortran yields similar observations.

5. CONCLUSIONS AND FUTURE WORK

The results of Section 4 demonstrate that the performance gap between Java and more traditional scientific programming languages is no longer a wide gulf. Although for each platform there are differences between the benchmark codes in terms of Java/C and Java/Fortran performance ratios, the variance is small enough to give some confidence that the benchmark suite is representative of a class of applications.

On Intel Pentium hardware, especially with Linux, the performance gap is small enough to be of little or no concern to programmers. On the Sun Ultrasparc platform the gap is a little wider, but generally less than a factor of two. On the Compaq Alpha platform, the gap is around a factor of four. These differences between platforms probably reflect the relative effort expended by vendors on developing Java environments compared to that expended on C and Fortran compilers, rather than intrinsic properties of the hard-

Pentium III, NT				
	Borland C++ 5.5.1	pgcc	pgf90 3.2-3	Digital F V5.0
Sun JDK 1.2.2_006	0.99 (8)	1.44 (8)	1.31 (2)	0.83 (2)
Sun JDK 1.3.0, client	1.30 (8)	1.87 (8)	2.48 (2)	1.57 (2)
Sun JDK 1.3.0, server	1.35 (8)	1.97 (8)	2.61 (2)	1.66 (2)
IBM JDK 1.2.0	1.10 (8)	1.60 (8)	3.30 (2)	2.10 (2)
IBM JDK 1.3.0	1.13 (8)	1.65 (8)	3.22 (2)	2.05 (2)
Microsoft SDK for Java 4.0	1.10 (8)	1.60 (8)	2.04 (2)	1.29 (2)
Pentium III, Linux				
	gcc 2.91.66	KAI C++ v4.0b	g77 2.91.66	pg77 3.1-2
Sun JDK 1.3.0, client	1.45 (8)	1.45 (5)	1.24 (2)	1.82 (2)
Sun JDK 1.3.0, server	1.54 (7)	1.40 (5)	1.83 (2)	2.70 (2)
Blackdown 1.3	1.46 (8)	1.40 (5)	1.25 (2)	1.84 (2)
IBM JDK 1.3.0	1.15 (8)	0.97 (5)	1.38 (2)	2.03 (2)
Sun UltraSparc II				
	Sun WS 6 cc 5.2	gcc 2.95.2	Sun WS 6 f90 95.6.1	g77 2.95.2
Sun JDK 1.2.1	1.72 (8)	1.43 (8)	1.43 (2)	1.23 (2)
Sun JDK 1.3.0, client	2.33 (8)	1.93 (8)	2.68 (2)	2.30 (2)
Sun JDK 1.3.0, server	1.92 (8)	1.59 (8)	1.58 (2)	1.35 (2)
LaTTe 0.9.1	1.80 (8)	1.49 (8)	1.59 (2)	1.37 (2)
Compaq ES40				
	Dec C V5.9-005		Compaq Fortran V5.3-1120	
Compaq Java 1.3.0-alpha1	3.77		3.47	

Table 3: Mean execution time ratios for various Java execution environments. Figures in brackets represent the number of benchmarks used to calculate the ratios.

		Pentium III, NT	Pentium III, Linux	Sun UltraSparc II	Compaq ES40
C	FFT	2.26	1.49	2.07	2.15
	HeapSort	1.00	0.95	1.29	2.48
	SOR	1.17	1.17	1.40	2.45
	LUFact	1.17	0.93	1.34	2.84
	Series	0.87	0.87	1.54	8.29
	Sparse	1.30	1.07	2.61	2.64
	Euler	1.48	1.41	1.94	17.6
	MolDyn	1.09	1.12	1.49	2.84
	Mean	1.23	1.07	1.61	4.08
Fortran	LUFact	1.18	1.16	1.48	5.07
	MolDyn	1.27	2.27	1.39	2.38
	Mean	1.22	1.62	1.43	3.47

Table 4: Ratios of fastest Java execution times to fastest C/Fortran execution time.

ware. However, the possibility cannot be discounted that the highly super-scalar nature of Pentium III micro-architecture has some influence.

Future work will extend the coverage of the language comparison benchmarks (particularly in the Fortran version). We will continue to monitor latest software releases, and broaden the coverage to more platforms. A further possibility is to include some parallel versions of the benchmarks (for both shared and distributed memory) in the language comparison suite.

6. ACKNOWLEDGEMENTS

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

We would also like to thank the University of St. Andrews and PPARC for access to the Compaq ES40 system.

7. REFERENCES

[1] Addison, C.A., Getov, V.S., Hey, A.J.G., Hockney, R.W. and Walton, I.C. (1991) *The GENESIS Distributed-memory Benchmarks*, Computer Benchmarks, J.J. Dongarra and W. Gentzsch (Eds), Advances in Parallel Computing, Vol 8, pp. 257–271.

[2] M. Baker, B. Carpenter, G. Fox, S.-H. Ko, and S Lim. mpiJava: An Object-Oriented Java interface to MPI. In *Proc. International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999.

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

[4] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. (2000) *A Benchmark Suite for High Performance Java*, Concurrency, Practice and Experience, vol. 12, pp. 375–388.

[5] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. (1999) *A Benchmark Suite for High Performance Java*, Proceedings of ACM 1999 Java Grande Conference, June 1999, ACM Press, pp. 81–88.

[6] Caromel, D., Doyon, F., Klauser, W. and Vayssiere, J. (1998) *A distributed raytracer for benchmarking Java RMI and Serialization*, `www.inria.fr/sloop/C3D/`

[7] Dongarra, J.J. (1998) *Performance of Various Computers Using Standard Linear Equations Software (Linpack Benchmark Report)*, University of Tennessee Computer Science Technical Report, CS-89-85 and Dongarra, J.J. and Wade, R. (1996) *Linpack benchmark: Java version*, `www.netlib.org/benchmark/linpackjava/`.

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

[9] Griswold, W. and Phillips, P. (1997) *UCSD Benchmarks for Java*, `www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html`

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

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

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

[13] Java Grande Forum, `www.javagrande.org`.

[14] M. E. Kambites, J. Obdrzalek and J. M. Bull (2001) An OpenMP-like Interface for Parallel Programming in Java, to appear in Concurrency and Computation: Practice and Experience.

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

[16] Mathew, J.A., Coddington, P.D., Hawick, K.A. (1999) *Analysis and development of Java Grande Benchmarks*, Proceedings of ACM 1999 Java Grande Conference, June 1999, ACM Press, pp. 72–80.

[17] Pendragon Software Corp. *Caffeine Mark 3.0*, `www.webfayre.com/pendragon/cm3/`

[18] Pozo, R. *Java SciMark benchmark for scientific computing*, `math.nist.gov/scimark2`

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

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

[21] SPEC, *SPEC JVM98 Benchmarks*, `www.spec.org/osg/jvm98/`

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

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