Universität Hamburg

Softwareentwicklung in der Wissenschaft

Benchmarking Java against C and Fortran for scientific applications

Dmitri Bespalko

July 02, 2018

1 Abstract

Before developing software one of the most important matters is choosing the right language. When it comes to scientific applications the choice of programming language often falls on traditional languages like C and Fortran, while Java for example is looked down upon and criticized for its performance without any comparison metrics to prove those claims. To allow developers to make a better justifiable choice those metrics must be provided.
The Edinburgh Parallel Computer Centre at the University of Edinburgh developed the Java Grande benchmarking suite for Java Execution Environments and used a set of those benchmarks to compare Java to C and Fortran.
The results of those benchmarks show that Java's performance overall is still worse than that of C and Fortran, however that performance can be optimized by choosing the right JRE and is often not significant enough to make a big difference.

2 Java Grande

2.1 Description

Java Grande is a standard benchmarking suite designed by the EPCC, University of Edinburgh. Its purpose is to quantify the performance of different Java Execution Environments, making comparison between them easier and exposing critical features to encourage development in appropriate directions.

2.2 Structure

The benchmarking suite is divided into three sections:
Low-Level-Operations, **Kernels** and **Applications**.

2.3 Section I: Low-Level-Operations

As the name already suggests benchmarks in this section are used to compare low-level operations, i.e.:

- Arithmetic operations, i.e. addition, multiplication etc.
- Casting between primitive types, i.e. int to long, double to float
- Exceptions
- java.lang.Math operations
- etc.

2.4 Section II: Kernels

This section contains benchmarks with relatively short code, defining computations commonly used in scientific applications. For this exact reason this section is supposed to be reflective of actual application performance in terms of scientific applications. To compare the languages there are a total of 6 different benchmarks used: **Series**, **LUFact**, **HeapSort**, **SOR**, **FFT** and **SparseMatmult**.

2.4.1 Series

This benchmark computes the first N Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval 0,2. The following is a representation of the same function with *sin* and *cos*:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n cos(nx) + \sum_{n=1}^{\infty} b_n sin(nx)$$

With the following formulas used to compute the actual coefficients a_n and b_n:

$$a_n = \frac{1}{2} \int_0^2 f(x) cos(nx) dx$$

$$b_n = \frac{1}{2} \int_0^2 f(x) sin(nx) dx$$

2.4.2 LUFact (Lower-Upper Factorization)

Lower-Upper factorization, also Lower-Upper decomposition, is a method to represent a matrix as a product of a lower and an upper triangular matrix. The example below shows both triangular matrices for a 2 x 2 matrix

$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix}$$

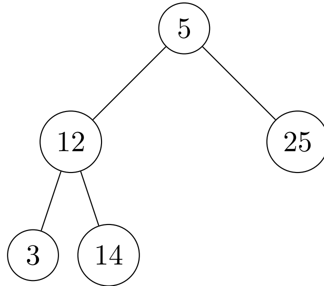
$$\begin{bmatrix} 4 & 3 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1.5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 0 & -1.5 \end{bmatrix}$$

The benchmark uses LU factorization to solve an N x N linear system.

2.4.3 HeapSort

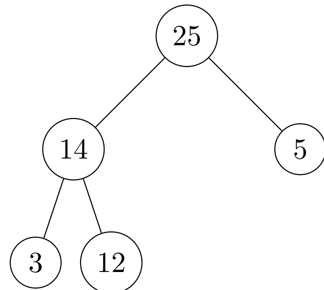
HeapSort is a common sorting algorithm that uses a heap, a tree-based data structure. To explain the algorithm shortly we try to sort the numbers 5, 12, 25, 3 and 14 represented below as an array and as a binary tree:

5	12	25	3	14



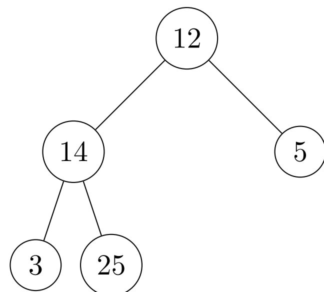
To begin with we need to transform the tree into a max heap, meaning each parent has to be larger than its children. To do that we go through each child from right to left, bottom to top and check if it is larger than its parent. If that is the case we swap them:

25	14	5	3	12

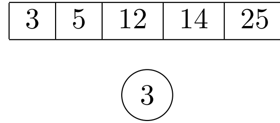


After we have built a max heap we swap the root with the last element:

12	14	5	3	25



We now know that the last element is the largest in the tree, so it is sorted. We remove it from the tree and proceed from the beginning until there is only one element left:



This benchmark sorts an array of N integers using the heap sort algorithm.

2.4.4 SOR (Successive Over-Relaxation)

Successive Over-Relaxation is an iterative method for solving linear system of equations.
This benchmark performs a total of 100 iterations on an N x N grid.

2.4.5 FFT (Fast Fourier Transform)

Fast Fourier Transform is an algorithm that samples a signal over a period of time or space and divides it into its frequency components.
The benchmark performs a one-dimensional forward transform of complex numbers.

2.4.6 SparseMatmult (Sparse Matrix Multiplication)

Sparse matrices are matrices that mostly contain zeros. Since knowing that an element is zero is not valuable information, storing each element of the array is inefficient and unnecessary. Therefore to store matrices it is much better to store only the elements that are not equal to zero.
For example we have the following matrix:

$$\begin{bmatrix} 2.0 & -1.0 & 0 & 0 \\ -1.0 & 2.0 & -1.0 & 0 \\ 0 & -1.0 & 2.0 & -1.0 \\ 0 & 0 & -1.0 & 2.0 \end{bmatrix}$$

To properly store the values of this array we have an array of 4 arrays, one for each row, which then contain tuples for each non-zero value in that row with the first value being the index of the column and the second value being the actual value of the element.

[[(0, 2.0), (1, -1.0)],
[(0, -1.0), (1, 2.0), (2, -1.0)],
[(1, -1.0), (2, 2.0), (3, -1.0)],
[(2, -1.0), (3, 2.0)]]

The benchmark performs a simple matrix-vector multiplication.

2.5 Section III: Applications

This section contains benchmarks designed with the intention of being representative of actual Grande applications. Consequently those benchmarks are of a more complex type. Therefore the differences in performance, especially in comparison between multiple languages, are very hard to explain. For this comparison there are two benchmarks used: **Euler** and **MolDyn**.

2.5.1 Euler

The benchmark is an application that solves the time-dependent Euler equations for flow in a channel with a 'bump' on one of the walls.

2.5.2 MolDyn (Molecular Dynamics)

This benchmark represents 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

2.6 Comparison

Since the Java Grande suite was mainly designed to compare Java Execution Environments, applying it to other languages has its difficulties.
While Section I may be a good indicator of the differences in performance between the different JREs, many of the low-level operations usually tested are exclusive to Java, i.e. exceptions and java.lang.Math, or are way too sensitive to compiler optimization to give any meaningful results. Therefore this section was completely excluded in the comparison.
Considering Java is the only object-oriented language out of the three, translating the different benchmarks is not an easy task. While C's syntax is still very similar to Java's, the differences between Java and Fortran are a lot more significant. Therefore all the benchmarks have been translated into C, while only LUFact and MolDyn have been translated into Fortran, with the best effort to keep the syntax as similar as possible in both cases.
Each benchmark has been run 3 times, with the results representing the best one.

3 Benchmark Results

3.1 Platforms/Compilers

The benchmarks were run on a total of 5 different platforms:

1. Pentium III, Windows NT
2. Pentium III, Linux 6.2
3. Sun UltraSparc II
4. Compaq ES40, Digital Unix v4.0F
5. SGI Origin 3000, Irix 6.5

With Pentium Windows and Linux being the most popular platforms and Sun being the developer of Java at the time of testing, those platforms have the largest variety of JREs, C and Fortran compilers, while the Compaq ES40 and the SGI Origin 3000 have no choice in that regard, with the latter even lacking a Fortran compiler.
In terms of compilers, the javac -o command was used for Java, with the -o flag being the standard optimization flag at the time, while only standard flags were used for C and Fortran compilers. No attempt of optimizing for individual code has been made.
Table 1 shows all Java execution environments, C and Fortran compilers and flags used.

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

Pentium III, NT	*Pentium III, Linux*
Sun JDK 1.2.2_006	Sun JDK 1.3.0 (-client)
Sun JDK 1.4.0b	Sun JDK 1.3.0 (-server)
Sun JDK 1.3.1	Blackdown JDK 1.3
Sun JDK 1.3.0 (-client)	IBM JDK 1.3.0
Sun JDK 1.3.0 (-server)	gcc 2.91.66 (-O3-funroll-loops)
IBM JDK 1.2.0	KAI C++ v4.0b (+K3-O)
IBM JDK 1.3.0	g77 2.91.66 (-O3-funroll-loops)
Microsoft SDK for Java 4.0	pg77 3.1-2 (-fast)
Borland C++ 5.5.1 (-5 -O2 -OS)	
Portland Group pgcc 3.2-3 (-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 1.4.0b, HotSpot Client	Dec C V5.9-005 (-fast -O4 -ieee -tuneev6 - archev6)
Sun JDK 1.4.0b, HotSpot Server	Compaq Fortran V5.3-1120 (-fast -O4 -ieee -tuneev6 - archev6)
Sun JDK 1.3.1, HotSpot Client	
Sun JDK 1.3.1, HotSpot Server	*SGI Origin 3000*
Sun JDK 1.3.0, HotSpot Client	SGI JDK 1.3.0
Sun JDK 1.3.0, HotSpot Server	MIPSpro V7.3.1.1m CC (-O3)
LaTTe 0.9.1	
Sun WS 6 cc 5.2 (-fast -xarch=v8plusa)	
gcc 2.95.2 (-O3 -funroll-loops)	
Apogee C 4.0	
Sun WS 6 f90 95.6.1 (-fast -xarch=v8plusa)	
g77 2.95.2 (-O3 -funroll-loops)	
Apogee f90 4.0	

3.2 Section II: Kernels

3.2.1 FFT

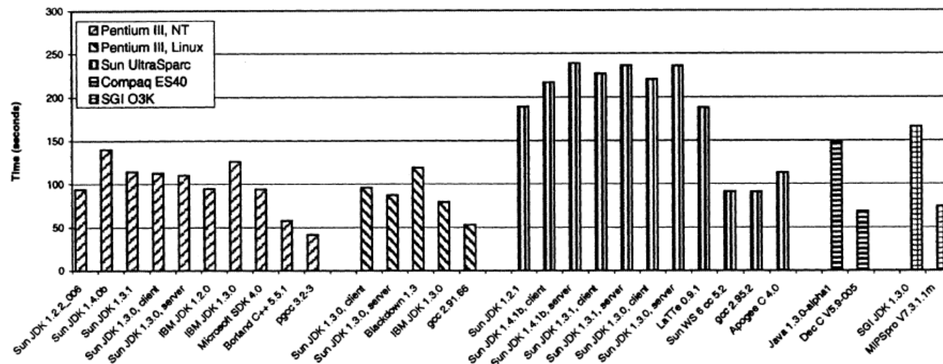


Figure 1: Execution time for the FFT benchmark.

For the Fast Fourier Transform Java is performing considerably worse on each platform. The fastest C compilers are more than twice as fast than the

fastest JREs on each platform except Pentium III, Linux, where the IBM JDK is around 50% slower.

3.2.2 HeapSort

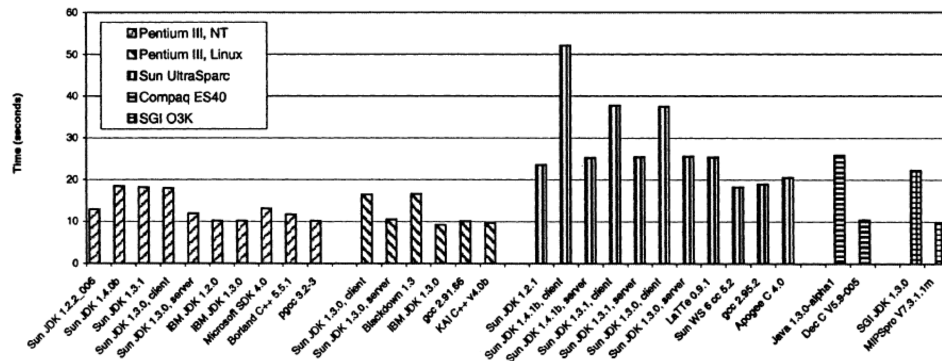


Figure 2: Execution time for the HeapSort benchmark.

The HeapSort benchmark performs very well for Java on the more popular and supported platforms. On Windows the fastest JRE's performance was equal to the fastest C compiler, while on Linux it was even faster.
On the Sun UltraSparc Java was still worse, however the difference is quiet small, although the difference between the individual JREs varies a lot.
The Compaq ES40 and SGI Origin 3000 still show a large performance loss when using Java.

3.2.3 LUFact

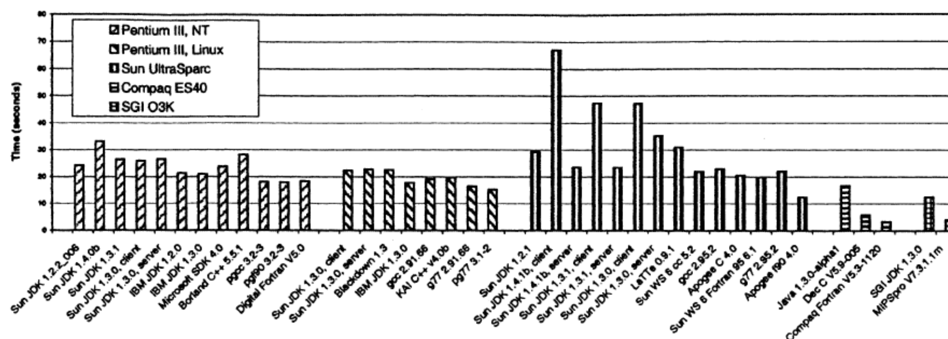


Figure 3: Execution time for the LUFact benchmark.

Lower-Upper factorization is the first benchmark that has been translated into both C and Fortran. Here Fortran has performed the best on all plat-

forms it was available on, followed very closely by C with Java being the slowest, however not by a lot.
On the Sun UltraSparc there are strong fluctuations between the different JREs once again.

3.2.4 Series

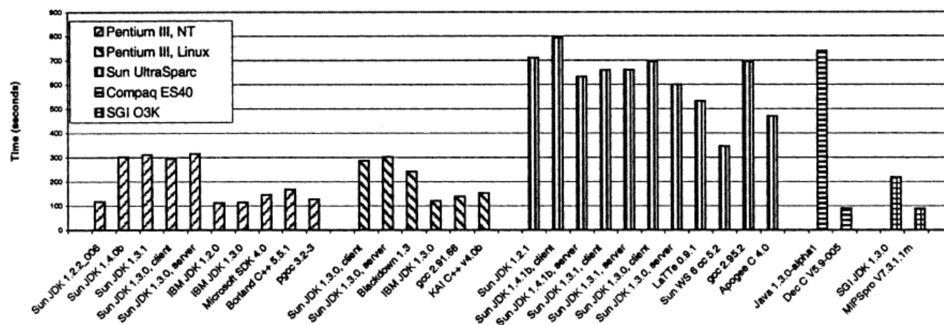


Figure 4: Execution time for the Series benchmark.

Similar to HeapSort the fastest JRE performed better than the fastest C compiler on the Pentium III platforms, however there are JREs that are almost three times slower than the fastest one.
On the Sun UltraSparc the performance is overall worse for both C and Java and for the most part very similar. However the fastest C compiler is still almost two times faster than the fastest JRE.
On the SGI platform the C compiler was, similarly to other benchmarks in Section II, about twice as fast as the JRE.
The Compaq ES40 however had a performance loss of a factor of almost seven and a half when using Java.

3.2.5 SOR

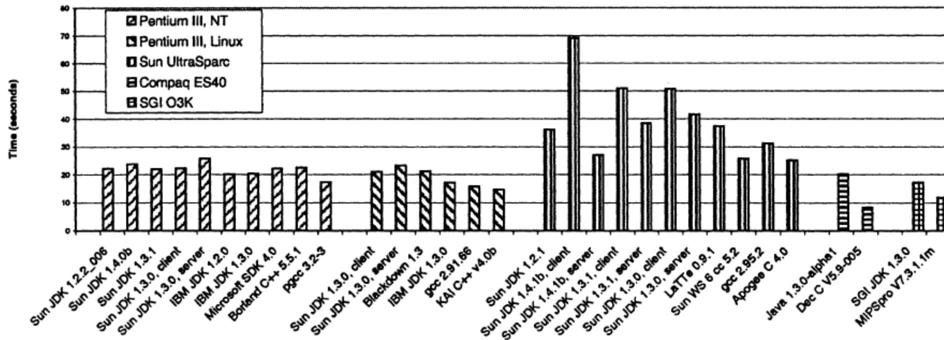


Figure 5: Execution time for the SOR benchmark.

The Successive Over-Relaxation performed similarly for Java and C on both Pentium platforms aswell as on the Sun UltraSparc, however the fluctuations between the JREs are a lot stronger on the Sun UltraSparc. Nevertheless the fastest C compiler outperformed the fastest JRE on every single platform.

3.2.6 SparseMatmult

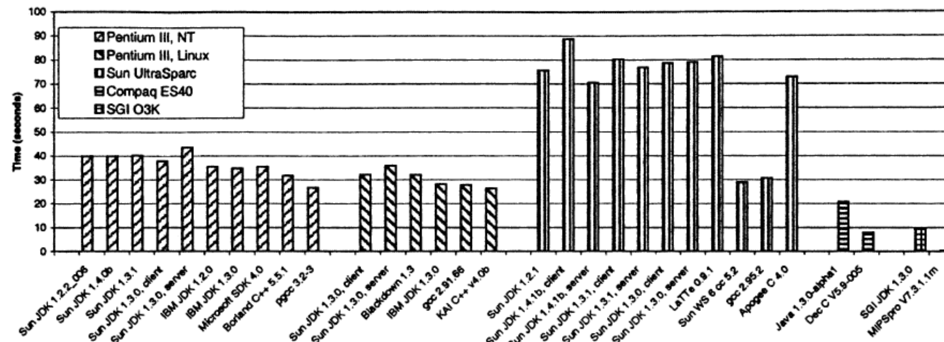


Figure 6: Execution time for the SparseMatmult benchmark.

Once again the C compiler outperformed the JRE. On Linux the difference is minimal and a little higher on Windows. On the rest of the platforms however C was much faster.
Overall the fluctuations between the JREs were very small, even on the Sun UltraSparc where they are usually queiet high.

3.3 Section III: Applications

3.3.1 Euler

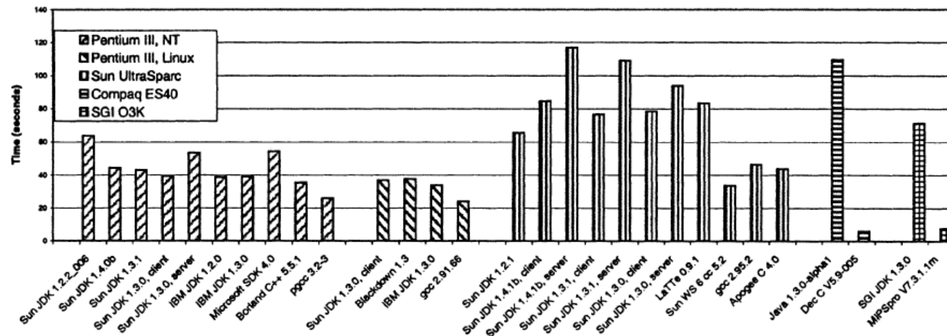


Figure 7: Execution time for the Euler benchmark.

On every platform the fastest C compiler outperformed all JREs. On the Pentium platforms the difference is relatively small, while on the Sun UltraSparc the fastest JRE is almost two times slower than the fastest C compiler. On the Compaq ES40 and the SGI Origin 3000 however the difference is immense. Even though the Sun UltraSparc had JREs with worse performances than the JREs used on these platforms, the C compilers were significantly faster than on any other platform, therefore making the performance of the JREs look a lot worse.

3.3.2 MolDyn

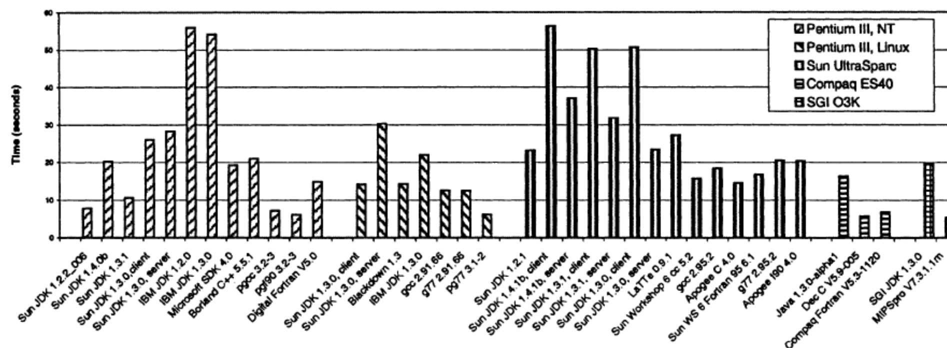


Figure 8: Execution time for the MolDyn benchmark.

For the Molecular Dynamics application the benchmark results look different compared to other benchmarks. The reason being the fluctuations in performance, especially between the different JREs. For Instance on the

Windows platform the fastest JRE is about five times faster than the slowest one.
Nevertheless the difference between the performance of the fastest compilers and JRE is for the most part small, the Fortran and C compilers each being the fastest on different platforms and Java being the slowest on every single one.

3.4 Summary

Table 2: 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					
	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 1.4.0b	1.38 (8)	2.01 (8)	2.47 (2)	1.57 (2)		
Sun JDK 1.3.1	1.19 (8)	1.73 (8)	1.59 (2)	1.01 (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 UltraSpace II					
	Sun WS6 cc 5.2	gcc 2.95.2	Apogee C	Sun WS6 f90 95.6.1	g77 2.95.2	Apogee f90
Sun JDK 1.2.1	1.72 (8)	1.43 (8)	1.40 (8)	1.43 (2)	1.23 (2)	1.64
Sun JDK 1.4.0b, client	2.78 (8)	2.30 (8)	2.26 (8)	3.37 (2)	2.89 (2)	3.86
Sun JDK 1.4.0b, server	1.87 (8)	1.55 (8)	1.52 (8)	1.62 (2)	1.39 (2)	1.86
Sun JDK 1.3.1, client	2.32 (8)	1.54 (8)	1.89 (8)	2.68 (2)	2.30 (2)	3.07
Sun JDK 1.3.1, server	1.93 (8)	1.93 (8)	1.57 (8)	1.50 (2)	1.29 (2)	1.72
Sun JDK 1.3.0, client	2.33 (8)	1.93 (8)	1.89 (8)	2.68 (2)	2.30 (2)	3.08
Sun JDK 1.3.0, server	1.92 (8)	1.59 (8)	1.56 (8)	1.58 (2)	1.35 (2)	1.81
LaTTe 0.9.1	1.80 (8)	1.49 (8)	1.46 (8)	1.59 (2)	1.37 (2)	1.83
	Compaq ES40					
	Dec C V5.9-005	Compaq Fortran V5.3-1120				
Compaq Java 1.3.0-alpha1	3.77 (8)	3.47 (2)				
	SGI Origin 3000					
	MIPSpro V7.3.1.1m CC					
SGI JDK 1.3.0	3.88 (8)					

Table 2 summarizes the overall performance of all used JREs in comparison to the C and Fortran compilers over all benchmarks.
The results show that for the most part the ratios stay under 2, which in most cases is not significant enough to choose a language solely for its performance. However for situations where each performance increase is very important the choice of the JRE can make a big difference, since the ratios vary between them.

Table 3: Ratios of fastest Java execution times to fastest C/Fortran execution times.

		Pentium III, NT	Pentium III, Linux	Sun UltraSparc II	Compaq ES40	SGI O3000
C	FFT	2.26	1.49	2.07	2.15	2.25
	Heapsort	1.00	0.95	1.29	2.48	2.28
	SOR	1.17	1.17	1.44	2.45	1.45
	LUFact	1.17	0.93	1.43	2.84	3.12
	Series	0.87	0.87	1.54	8.29	2.47
	Sparse	1.30	1.07	2.61	2.64	26.0
	Euler	1.48	1.41	1.94	17.6	9.31
	MolDyn	1.09	1.12	1.61	2.84	3.67
	Mean	1.23	1.07	1.74	3.78	3.88
Fortran	LUFact	1.18	1.16	2.36	5.07	
	MolDyn	1.27	2.27	1.39	2.38	
	Mean	1.22	1.62	1.88	3.47	

Considering we always choose the best JRE for each situation Java's performance might still be worse then C's and Fortran's, however the difference can be kept minimal.
Table 3 shows the ratios of the fastest Java execution times to fastest C and Fortran execution times. As can be seen for the Series benchmark on the Pentium platforms aswell as for the LUFact benchmark on Linux, Java can outperform the traditional languages. Even for the MolDyn benchmark, which is supposed to represent a real application, the performance loss of Java is very small.

4 Conclusion

Overall, looking at the results, it is clear that Java's performance in general is still worse than C's and Fortran's. However the differences vary a lot with the platform and benchmark used.
It is also apparent that a newer JRE does not necessarily mean a better performance. Among all of the tested JREs there is none that has been the best across all benchmarks. That means that with a proper understanding of the platform used and the requirements of the software the right Execution Environment can be chosen to minimize the performance loss by choosing Java.
In a lot of cases however the performance difference is not significant enough for the developers to worry about anyway, making it more important to look at other aspects like comfort.

5 References

- Bull J.M., Smith L.A., Ball C., Pottage L. and Freeman R. Benchmarking Java against C and Fortran for scientific applications. *Concurrency and Computation: Practice and Experience* 2003;**15**(3-5):417-430.

- Bull J.M., Smith L.A., Westhead M.D., Henty D.S. and Davey R.A. Benchmarking Java Grande Applications. [https://pdfs.semanticscholar.org/db16/efd09640224c890bb188092bd2a0cac48055.pdf]. Accessed July 02, 2018.

- Bull J.M., Smith L.A., Westhead M.D., Henty D.S. and Davey R.A. A Methodology for Benchmarking Java Grande Applications. [http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.5439&rep=rep1&type=pdf]. Accessed July 02, 2018.