CCLRC

Technical Report
DL-TR-2004-003

A COMPARISON OF C, FORTRAN AND JAVA FOR MUNERICAL COMPUTING

J. V. ASHBY

5th August 2004

Council for the Central Laboratory of the Research Councils

A comparison of C, Fortran and Java for Numerical Computation

J.V. Ashby
Computational Science and Engineering Department
CCLRC Rutherford Appleton Laboratory
Chilton, Didcot, Oxon OX11 0QX
J.V.Ashby@rl.ac.uk

Abstract

Three commonly used high-level programming languages are compared for their use in numerically intensive computation. A sparse matrix multiply which contains a typical mix of floating-point arithmetic and indirect addressing has been written in each language and is run on a variety of machines under several different compilers. The results show that speed of executed code varies as much from compiler implementation as from choice of language. C and Fortran90 perform very similarly, but Java is still too slow to be a serious contender for high performance numerical programming

1. INTRODUCTION

Programming languages have evolved considerably from the early days of computing, and programming methodologies have emerged which have their roots in theoretical computer science (functional programming, object-oriented programming, etc.). Yet much of the computational science and engineering community remains wedded to Fortran in one or another of its incarnations. Often this commitment to a language which first emerged in 1954 is rationalised in one of three ways. The large corpus of legacy software, particularly in the form of libraries, would need to be re-developed in a new language. Learning a new language would (at least initially) entail a loss of productivity. Finally, the maturity of optimising compilers, it is said, provides such a performance benefit that other languages simply cannot compete. Proponents of younger languages and methodologies point to ease of development and more provably correct programs, particularly since data encapsulation should eliminate many of the unwanted side-effects that Fortran allowed.

This report looks at three languages and assesses their suitability for numerical computation. The languages are Fortran, C and Java. We shall first discuss the strengths and weaknesses of each of these languages, then present results of a comparison of their performance on a typical numerical kernel, a sparse matrix-matrix multiply.

2. THE LANGUAGES

2.1 Fortran

Fortran was one of the first high-level languages developed, and the programming model it implicitly contains is shaped by its early target market of mathematicians and scientists who wished to write programs that looked like the mathematical formulae they were used to manipulating on paper. Its very name comes from FORmula TRANslation, and the basic objects of the language, reals, integers, arrays, etc., are direct counterparts of the variables and constants which appear in any mathematical model. Fortran has evolved through the standardisation process that gave us FORTRAN66, FORTRAN77, Fortran90, Fortran95 and the Fortran2003 standard currently in its Committee Draft stage [1]. During this evolution the language became capable first of expressing algorithms more richly and more simply, then of extending the objects it handled to more abstract structures and of supporting to some extent an object oriented programming model.

Because Fortran programmers manipulate 'real life' mathematical objects, there is often a mismatch between such objects and their representation on the computer which can lead to inefficiency and errors. For example, a series of large numbers, both positive and negative, can be added together mathematically and give an answer which is much smaller than the individual terms. When implemented in the finite arithmetic of a computer, the result of such an addition can depend dramatically on the order in which the terms are added. Such truncation errors can, in the worst cases, dominate the performance of a program and give incorrect results. Similarly if a large array is accessed in a manner which causes large strides through the memory onto which it is mapped, so much time can be spent bringing portions of memory into and out of the active virtual memory that the efficiency of the calculation can be adversely affected. Modern compilers have sophisticated algorithms which can recognise many of the more obvious problems and optimise the program produced so that it is no longer a line-by-line direct translation of what the programmer wrote. Techniques which exploit particular features of the cpu hardware are also available such as loop unrolling on pipelined architectures. The maturity of Fortran, its relatively simple and safe datatypes and the importance of efficiency mean that these optimisation techniques are considerably advanced in Fortran compilers.

2.2 C

C was developed in the early 1970s, largely as a systems' programming language for Unix [2]. While it shares some of the basic objects of Fortran and is still a procedural language, it is much "closer to the machine" in that its datatypes and model of programming map more directly onto the logical representation of data in memory. In particular the use of pointers and pointer arithmetic allow much more hands-on manipulation of the low-level representation of program objects. This gives it more power, but also can prevent several of the important optimisations available to Fortran compilers as assumptions about what object is being referenced cannot be guaranteed in C. On the other hand, a good C programmer can produce hand optimised code which is comparable with machine optimised Fortran. The price paid for this is often in readability and maintainability of code. However, for the core routines in computational software which are not likely to change much or need to be read there is an argument for using C. The argument runs that one can achieve assembler-like performance while maintaining portability.

Many modern libraries of software are written in C and provide an interface to other languages such as Fortran (a language binding or Application Programming Interface (API)). For those that deal with system functionality (such as I/O, inter-process communication, user interfaces, graphics, etc.) this is clearly sensible, and C's close to machine nature is a clear advantage. Numerical libraries have been slower to migrate to C and it is not clear that such a migration is advantageous.

C achieved standardisation in 1989, but six years earlier C had been extended to C++ by the addition of ideas from object oriented programming. This language has been gaining popularity as a workhorse general-purpose programming language, mainly driven by computer scientists attracted by its programming model and by the commercial world who view it as providing more easily maintainable software than earlier languages. In recent years numerical libraries have been developed in C++ incorporating published class libraries for customisation [3].

2.3 Java

The new kid on the block, Java looks very similar to C and C++, and incorporates much of C++'s OO programming model [4]. It distinguishes itself from its predecessors in two major respects: the definition in the original language of a large range of class libraries, particularly for user interface activities (window management, mouse interaction, menus, etc.) and in being platform independent. This independence is achieved since Java code is either interpreted at run-time (as with BASIC) or it is compiled into machine code for a virtual machine, the Java Virtual Machine (JVM). This compiled code can then be run on any platform, the local JVM making the final translation to native machine instructions.

3. THE APPLICATION

As an example of a scientific application kernel we consider a sparse matrix-matrix multiply. Although many scientific codes solve problems represented by full matrices, the sparse multiply does contain a mixture of indirect addressing and floating point operations which is typical of even full matrix codes.

A sparse matrix is one where a significant number of entries in each row or column are identically zero. Such matrices arise naturally in finite difference and finite element discretisations of partial differential equations as well as in tight-binding Hamiltonians of large systems and in many optimisation problems (and, of course, the identity matrix is trivially sparse). With such a large number of zeroes it would clearly be pointless and wasteful to store all entries of the matrix, so schemes which use indirect addressing are employed [5]. For example, one could store the non-zero (real) entries in one array and the two integer indices, i and j, in another. This makes the multiplication of two sparse matrices more complex than that of two full ones. For full matrices the following pseudo-code is sufficient:

```
integer n, i, j, k
real matrix A(n:n), B(n:n), C(n:n)
for i=1 to n
    for j=1 to n
        C(i,j) = 0.0
        for k=1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
        next k
    next j
next i
```

For sparse matrices this could become:

```
    integer rank, n, i, j, k, l, m
    real sparse_matrix A(n), B(n), C(n)
    integer sparse_index IA(n,2), IB(n,2), IC(n,2)
    m = 1
    for i=1 to rank
        for j=1 to rank
            C(m) = 0.0
            for k=1 to n
                if (IA(k,1) == I) then
                    for l = 1 to n
                        if (ib(l,1) == IA(k,1) and IB(l,2) == j) then
                            C(m) = C(m) + A(k) * B(l)
                            IC(m,1) = i
                            IC(m,2) = j
                    next l
                    m = m + 1
                end if
            next k
        next j
    next i
```

This is by no means the most efficient algorithm, but it is typical of the extra complication sparse computations entail. In the test codes the storage system used is two integer arrays, IA and JA, and one real array, A, as above, but IA(i) contains the starting indices of row i in JA and A, while JA(IA(i):IA(i+1)-1) contains the column indices of the entries and A(IA(i):IA(i+1)-1) their value. This leads to indirect addressing of JA. Issues which affect the performance are how well the compiler can optimise this indirect addressing and how well memory can be used to minimise paging as well as how the compiler or the language can perform floating point calculations.

4. RESULTS

In order to compare the three languages under consideration we took programs written to perform a sparse matrix-matrix multiply using the same underlying algorithm in each language and ran them on a variety of matrices and on a variety of computers. The time taken for each was recorded and normalised by the cycle time of the processor. (Another possible normalisation would have been to use the SpecFP rating of the processor, but this would have been in part to beg the question we were interested in.)

The sparse kernels form the JASPA suite [6]. The matrices were taken from the Matrix Market collection (http://math.nist.gov//MatrixMarket) and comprise a wide-ranging sample of sparse matrices from computational science applications as detailed in Table 1.

Matrix name	Discipline
Af23560	CFD – aerofoil
Bcsstk30	Structural Engineering
E40r0000	CFD – driven cavity
Fidap011	Finite Element
Fidapm11	Finite Element
Memplus	Electronic circuit design
Qc2534	Quantum Chemistry
S3dkt3m2	Structural Mechanics – cylindrical shells

Table 1: Test matrices from MatrixMarket

These matrices were squared using the sparse multiply kernels. The raw and normalised speeds are given in Tables 2 and 3. Several conclusions can be drawn immediately from these tables. For example, it is clear that the detailed structure of the matrix (the sparsity pattern) has a considerable influence on the speed of this operation which can vary by a factor of up to 3.58 (Lahey Fujitsu Fortran90 on a pentium PC). The compiler also has an influence on the speed as shown by the Xeon workstation where we had three Fortran compilers available: the Lahey Fujitsu f95, Intel's f95 and the Pacific-Sierra compiler which translates a subset of Fortran90 to FORTRAN77 using VAST90 and then uses g77 to compile this. (We also had available the NAG f90 compiler, but this was unable to link the program we were using due to its non-standard use of *getarg*.) The maximum ratio of speed from the Pacific-Sierra code to that from Lahey-Fujitsu was 2.18 (af23560). However, it can also be seen from Figure 1 that for the most part the relative speeds for the various matrices are independent of language and machine. The minor exceptions are the memplus and qc2534 matrices which have anomalously slow and fast performances in some Fortran experiments.

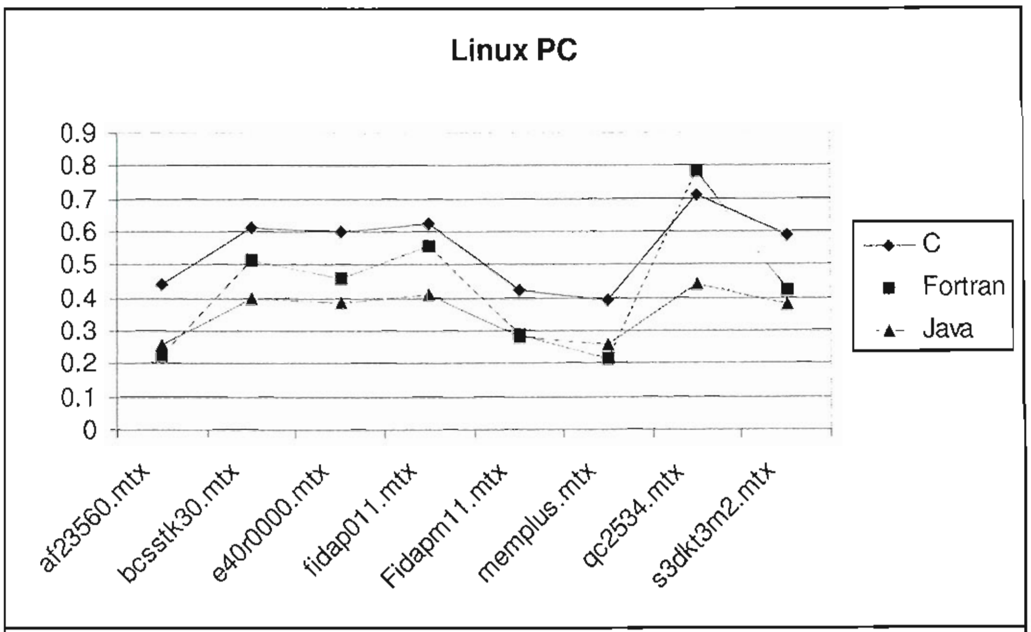
Two points can be drawn from this. Firstly that the differences we see between languages are similar in magnitude to the differences between different implementations of the same language. Secondly that nevertheless we can use the consistency of the relative speeds to draw (albeit with some caveats) some conclusions about the performance of the languages.

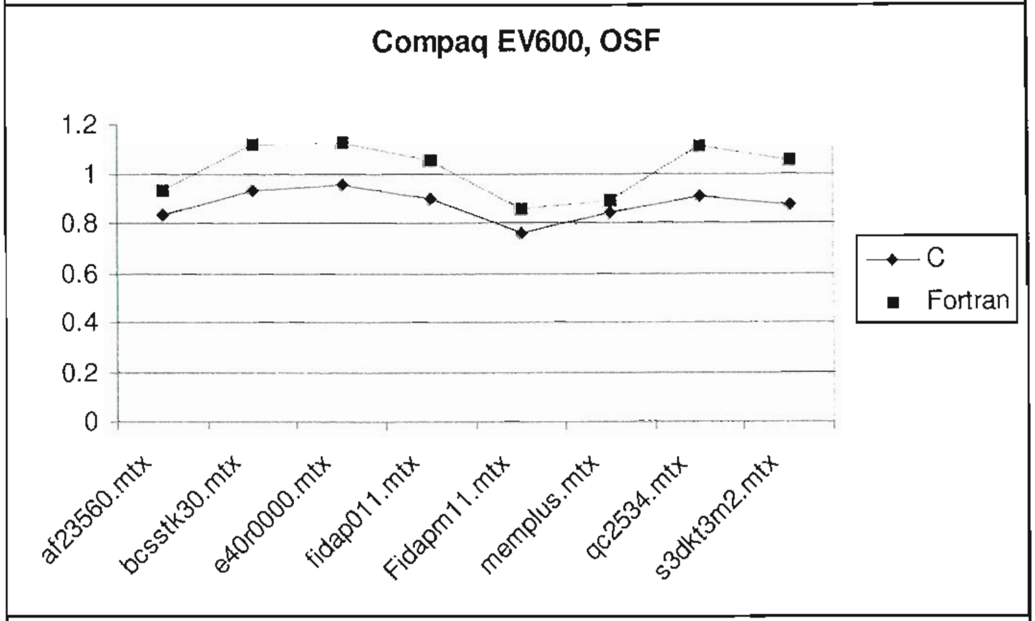
machine	speed	Language	Compiler	Af23560	Bcsstk30	E40r000	Fidap011	Fidapm11	Memplus	Qc2534	S3dkt3m2
Pentium Linux PC	1.5GHz	C	Gnu	66.21	91.81	90.11	93.67	63.04	58.83	106.46	87.61
		Fortran90	Lahey-Fujitsu	34.19	76.74	68.55	89.92	42.93	32.40	116.83	62.81
		Java		38.46	59.65	57.68	61.69	42.07	38.60	65.64	56.39
Compaq EV600	833MH	C	Compaq C	69.35	77.51	79.31	74.53	63.09	69.80	75.25	72.94
		Fortran90	Compaq f90	77.25	93.06	94.12	87.87	71.48	74.02	92.79	87.56
		Java									
Compaq EV6	500MH	C	Compaq C	47.17	53.25	55.25	51.43	42.72	48.20	49.43	48.86
		Fortran90	Compaq f90	54.51	66.31	68.49	61.81	49.23	52.17	60.45	60.89
		Java									
Origin 300 R14000	500MH	C	MIPSpro	45.82	55.13	56.90	55.23	44.84	42.56	57.61	52.17
		Fortran90	MIPSpro	50.04	62.41	64.22	58.69	46.82	38.49	64.06	60.18
		Java		17.71	20.92	20.23	21.43	17.57	16.07	22.38	20.39
Xeon Linux Workstation	3020	C	gcc	66.21	91.81	90.11	93.67	63.04	58.83	106.46	87.61
		Fortran90	Lahey	70.07	123.44	117.80	139.30	82.39	61.60	178.46	104.72
		Fortran90	Intel	139.17	209.90	205.51	230.07	138.08	111.40	275.54	191.12
		Fortran90	VAST90/g77	172.81	225.03	215.63	227.10	150.50	139.13	262.12	220.43
		Java		38.46	59.65	57.68	61.69	42.07	38.60	65.64	56.39

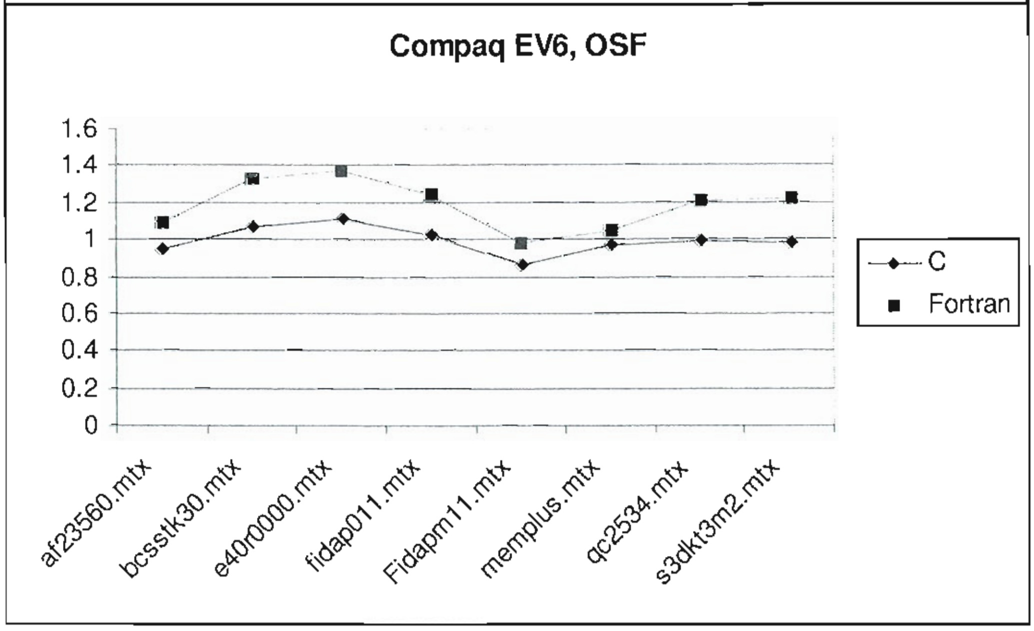
Table 2: Speed in Mflops of sparse matrix-matrix multiply in various languages on various computers

Machine	speed	Language	Compiler	Af23560	Bcsstk30	E40r000	Fidap011	Fidapm11	Memplus	Qc2534	S3dkt3m2
Pentium Linux	1.5GHz	C	Gnu	.044	.061	.069	.063	.042	.039	.071	.059
		Fortran90	Lahey-Fuiitsu	.023	.051	.046	.055	.029	.o22	.078	.042
		Java		.026	.040	.039	.041	.028	.026	.044	.038
Compaq EV600	833MHz	C		.083	.093	,095	.089	.076	.084	.090	.088
		Fortran90		.093	.112	.113	.105	.086	.089	.111	.105
		Java									
Compaq EV6	500MHz	C		.094	.107	.111	.103	.085	.096	.099	.098
		Fortran90		.109	.133	.137	.124	.098	.104	.121	.122
		Java									
Origin 300	500MHz	C	MIPSpro	.092	.110	.114	.110	.090	/085	.115	.104
		Fortran90	MIPSpro	.100	.125	.128	.117	.094	.077	.128	.120
		Java		.035	.042	.040	.043	.035	.032	.045	.041
Xeon Linux Workstation	3.02GHz	C		.022	.030	.030	.031	.021	.019	.035	.029
		Fortran90	Lahey Fujitsu	.026	.041	.039	.046	.027	.020	.059	.035
		Fortran90	Intel	.046	.070	.068	.076	.046	.037	.091	.063
		Fortran90	VAST90/g77	.057	.075	.071	.075	,050	.046	.087	.073
		Java		.013	.020	.019	.020	.014	.013	.022	.019

Table 3: Speed of sparse matrix-matrix multiplication relative to processor clock speed.


Linux PC
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
af23560.mtx
bcsstk30.mtx
e40r0000.mtx
fidap011.mtx
Fidapm11.mtx
memplus.mtx
qc2534.mtx
s3dkt3m2.mtx
C
Fortran
Java


Compaq EV600, OSF
1.2
1
0.8
0.6
0.4
0.2
0
af23560.mtx
bcsstk30.mtx
e40r0000.mtx
fidap011.mtx
Fidapm11.mtx
memplus.mtx
qc2534.mtx
s3dkt3m2.mtx
C
Fortran


Compaq EV6, OSF
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
af23560.mtx
bcsstk30.mtx
e40r0000.mtx
fidap011.mtx
Fidapm11.mtx
memplus.mtx
qc2534.mtx
s3dkt3m2.mtx
C
Fortran

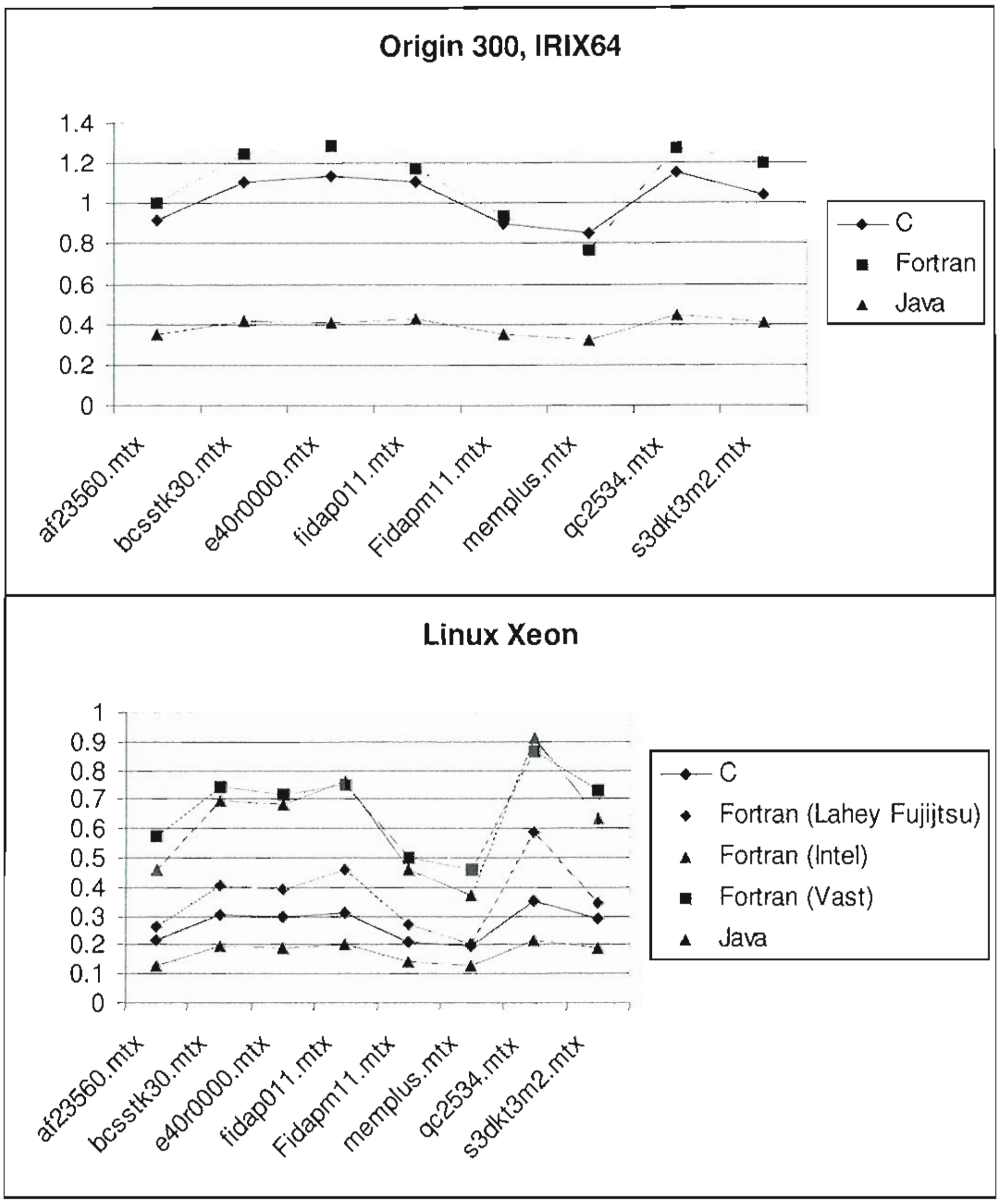


Figure 1: Normalised speeds for sparse matrix multiply in various languages on various machines.

Looking at table 3 and at Figure 1 it is clear that in the main Fortran90 is faster than C (though not by much and this is very dependent upon the compiler) and that Java is quite considerably slower. It is interesting to consider the Java results on the Pentium PC and on the Origin300. The Origin did not have a Java compiler available, so we used the compiled (JVM) code produced by the compiler on the PC. The speeds relative to processor rate are virtually identical, which may simply be a function of the Java Virtual Machine or it may mean that there is still scope for tuning the runtime translation from JVM to native instructions to make better use of the processor architecture. If the latter is the case then we may see the speed of Java code increase in the next few years as more Java implementations come onto the market, targeted at specific machines.

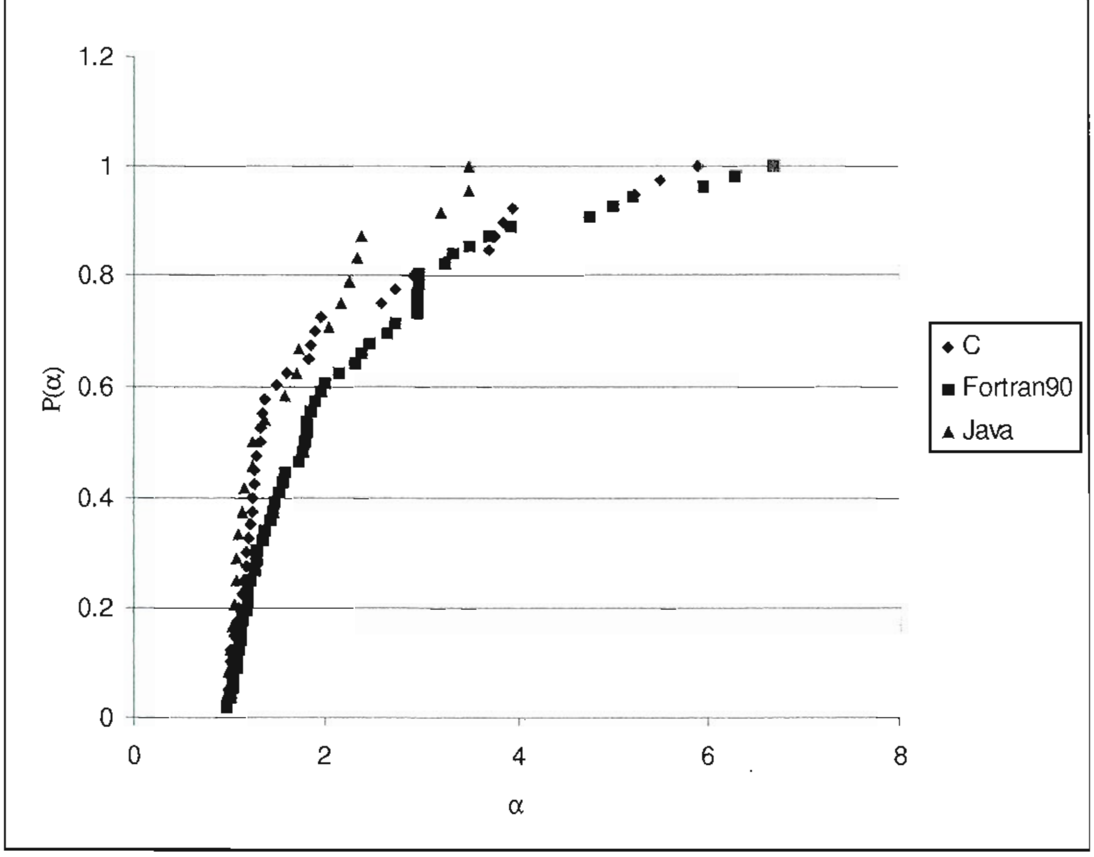
There are no speeds given for Java on the two Compaq chips. This is because although there was a Java compiler on both these machines, the runtime environment was lacking and the compiled code failed to run, issuing the error message: Unable to initialise threads: cannot find class java/lang/Thread. Attempts to circumvent this problem, for example through the use of compiler flags to use either native or green threads or through environment variables to hint where the class might be found were to no avail.

To analyse the data further we have taken several approaches which aim to measure the performance of a language across a range of problems. The simplest method is to average the normalised times for each language over the set of matrices and computers. In this case Fortran90 is the fastest with an average of 0.0772, then comes C with 0.0742 and Java is less than half as fast with 0.0306.

Another measure is to take the speeds for each matrix on each machine and to assign three points to the fastest code, two to the middle and one point to the slowest code. In this case the total number of points is Fortran90: 110, C: 88 and Java: 42. The difference between this measure and the simple average is that the relative difference between Fortran90 and C in the two measures shows the fact that Fortran90 is faster than C in many more cases, although the factor by which it is faster may not be that large. It may also be the case that the previous result for C can be skewed by a few very fast results.

Finally we introduce a measure which attempts to indicate the value of each language as a general-purpose tool [7]. Consider the set of timings (normalised by both size of problem and processor clock rate) for each matrix i and language j $\{x_i^j\}$. Let $x_{\min}^j$ be the value of the shortest time for a given language. Then define $P^j(\alpha)$ as the fraction of matrices for which the speed in language j is within a factor α of the fastest (i.e. for which $x_i^j < \alpha x_{\min}^j$). $P^j(\alpha)$ runs from 0 to 1 as α runs from 1^- to infinity. The faster $P^j(\alpha)$ rises, the more generally useful the language, since this means that for a given acceptable time more matrices will be multiplied within that time. Of course, this says nothing about whether or not a different language might perform faster. If all experiments in language A ran at the same speed, but this speed was 10 times slower than the slowest experiments in language B (whose spread of speeds was perhaps 10%), then one might still conclude that B was more useful than A. In figure 2 we show $P^j(\alpha)$ as defined above.

Figure 2. Performance Criterion (against fastest per language)

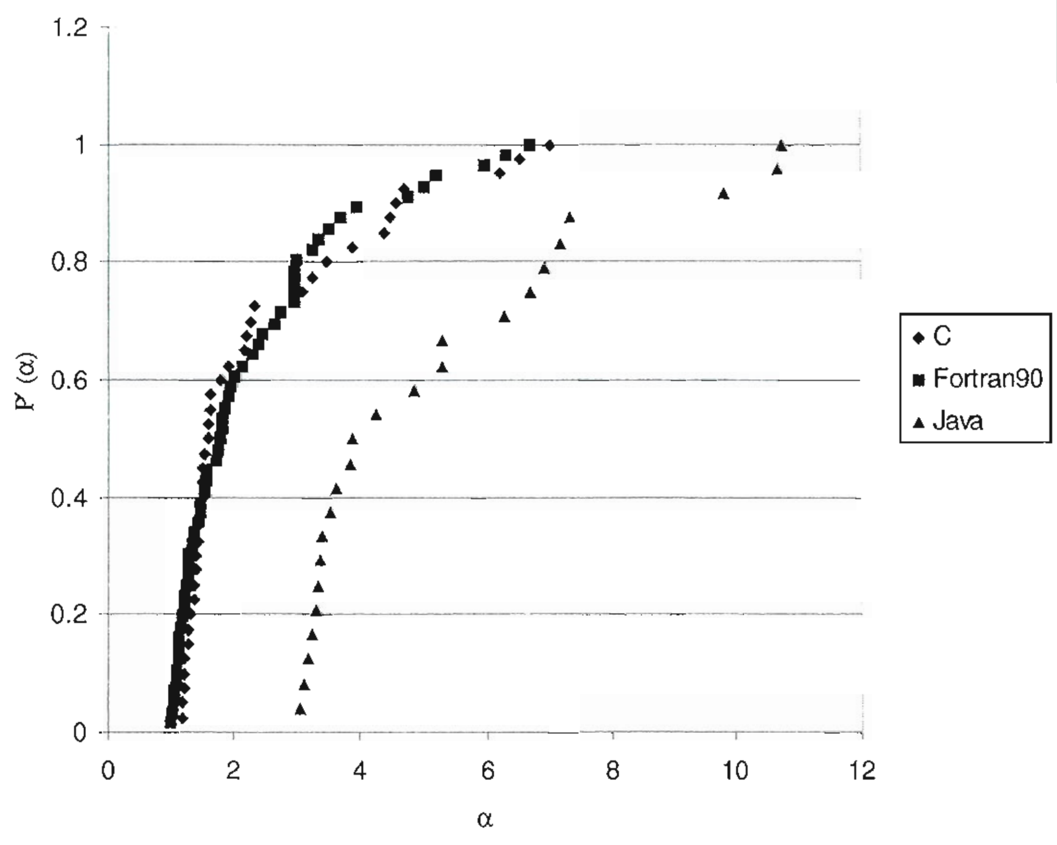


When languages are compared within themselves it is clear that Java and C are the most consistent, with Java having the edge over C at low and high α. To an extent this reflects the uniformity of compilers – all the Java code is compiled with a Sun or Sun-derived compiler and only the virtual machines are tailored to the individual architectures. Similarly the poor behaviour of Fortran at the high α end is an artefact of the relatively poor performance of the Lahey-Fujitsu compiler.

Comparing against the overall fastest execution speed, on the other hand, shows clearly how slow (roughly a factor of three) Java is. Figure 3 shows $P'^{j}(\alpha)$, where the definition of $P'^{j}(\alpha)$ uses the shortest normalised time over both experiments and languages. Again C performs slightly better than Fortran in the mid-range and it would be hard to make a clear recommendation that one language is clearly better than the other on the basis of these results.

Since Fortran90 has a number of new language features that are used by this code, it is worth estimating the effect using these has on the performance of the Fortran90 code. The two main features used are derived types (akin to structs in C) and pointer indirection. Indeed in Fortran 90 and 95 the use of derived types of any complexity almost invariably forces the use of pointers since elements of a derived type cannot be ALLOCATABLE – this is remedied in Fortran2003. We have produced two alternative versions of the Fortran implementation of the sparse matrix multiply. In

Figure 3. Performance Criterion (against overall fastest)

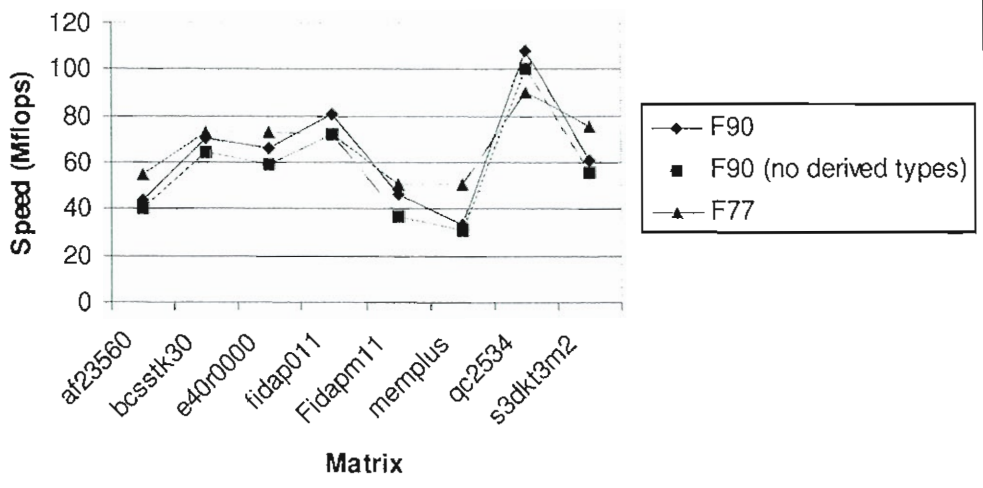


the first we expand the sparse matrix derived type into its components and pass those explicitly. Thus one argument of type sparse matrix to a subprogram unit is replaced by two integers, two integer pointer arrays, one real pointer array and a logical. In Figure 4 this is referred to as F90 (no derived types). In the second variant we make the pointer arrays static and large enough to hold the largest matrix in our test set. This is referred to as F77 in Figure 4, although strictly speaking there are still a few F90 remnants in the code such as the use of KIND= and INTERFACE. However, the passage of arguments and use of memory follows the Fortran77 model.

Figure 4 shows that there is a performance gain from using derived types (presumably achieved by reducing the overheads in calling subprograms) of between 7% and 20%. Fortran77, on the other hand, also represents a performance gain in most cases, sometimes of as much as 25%, though it can also lead to loss of performance (17% in the worst case). This loss is likely to be due to the way in which the matrix fits within the statically declared memory leading to increased paging.

It would be informative to perform a similar experiment with migrating the C code to C++.

Figure 4. Effects of F90 language features



5. SUMMARY

The results given above suggest that (at least for the limited range of problems considered) Java has still a long way to go before it can be considered a serious contender for numerically intensive computing. On the other hand, the performance of C and Fortran90 is now so close as to make any distinction between the languages on performance grounds specious. Programmers will, and indeed, should, use the language they are most comfortable with to develop their code. To this end the emerging Fortran2003 standard which deals, *inter alia*, with the interoperation of C and Fortran is a welcome advance.

6. REFERENCES

[1] Metcalf, M., Reid, J. Fortran 90/95 Explained, OUP (1996), Metcalf, M., Reid, J. and Cohen, M.A. Fortran 95/2003 Explained, OUP (In Press)

[2] The Development of the C Programming Language, Dennis Ritchie *History of Programming Languages-II* ed. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. ACM Press (New York) and Addison-Wesley (Reading, Mass), 1996. Also <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>

[3] Applied Numerical Libraries for Parallel Software, Antonioletti, M., Darling , G., Ashby, J.V. and Allan, R.J. <http://www.ukhec.ac.uk/publications/reports/numlib.pdf>

[4] History of the World Wide Web, Shahrooz Feizabadi, in World Wide Web, Beyond the Basics, Marc Abrams (Ed) Prentice Hall 1998 <http://ei.cs.vt.edu/~wwwbtb/book/chap1/java_hist.html>

[5] Numerical Recipes in Fortran, Press, W.H., Flannery, B.P. and Teukolsky, S.A., CUP (1992) <http://www.library.cornell.edu/nr/bookfpdf/f2-7.pdf>

[6] Comparing the performance of JAVA with Fortran and C for numerical computing, Hu, Y.F, Allan, R.J. and Maguire, K.C.F., <http://www.cse.clrc.ac.uk/arc/JASPA/bench.shtml>

[7] I am grateful to Jennifer Scott and Nick Gould for suggesting this approach

ISSN 1362-0207