

Java Grande language comparison benchmarks on the IBM p690 system

Joachim Hein
Edinburgh Parallel Computing Centre, Edinburgh, EH9 3JZ

Abstract

In this study we compare the performance of Java, C and Fortran for a set of serial benchmarks on a p690 system. The performance of Java compares favourably with the equivalent Fortran and C versions. Some variation in run times were observed for all the benchmarks, due to different logical partitions of the p690 system not being fully independent. This variation is of the order of a few percent.

1 Java Grande benchmarking activities

EPCC leads the benchmarking activities of the *Java Grande Forum*. A subset of the sequential Java benchmarks have been written in C and Fortran to allow language comparisons to be carried out. Details on the benchmarks, and how they can be downloaded, can be obtained from:

<http://www.epcc.ed.ac.uk/javagrande/>

The versions of the language comparison benchmarks used in this study are listed in table 1.

All codes described are sequential codes running on a single processor.

2 HPCx system

The system used in this study (named HPCx) consists of 40 tightly coupled IBM p690 Regatta H nodes. Each node consists out of 4 multi chip modules (MCM), which in turn have 8 power PC processors. At the time of writing the nodes are subdivided into 4 logical partitions (Lpar). Each Lpar consists out of an MCM with 128 MB of L3 cache and 8 GB of main memory attached to it. Inside an MCM are four Power4 chips. Each chip has 2 processors and 1440 kB of L2 cache. L1 cache is 32 kB per processor. These cache sizes are summarised in table 2.

Each benchmark has been executed on a single Lpar, with no other applications running within the same Lpar.

Language	Version
Java	2.0
C	1.0
Fortran	1.0

Table 1: Versions of Java Grande language comparison benchmarks utilised

Level	Cache size
L1	32 kB
L2	1440 kB
L3	128 MB

Table 2: Summary of cache sizes on the p690 system

2.1 Environment

Details of the the operating system, compilers and run time environment are summarised in table 3.

For the Fortran version the `xlf90_r` call was used for F90 free format files. The linking was also done with a `xlf90_r` call. F77 fixed format files have been compiled with the `xlf_r` call.

3 Individual Benchmarks

With the present benchmark suite, three individual benchmarks exist for all three programming languages. The Java and C version contain additional benchmarks to the ones discussed here.

3.1 HeapSort

This benchmark measures the time taken for the heap sort algorithm to sort an array of 4 byte integers. The benchmark suite consists of three different application sizes: ‘A’, ‘B’ and ‘C’. Details of these sizes are given in table 4.

None of the three data sizes described above will fit into the p690’s L2 cache. Hence in order to test cache behaviour, we define a smaller ‘M’ size, which will fit into L2 cache. This smaller size is not part of the standard Java Grande benchmark suite.

Each benchmark has been repeated 40 times. Figure 1 shows the fastest measured run time of these 40 repeats, for each data size.

Environment	Version	Call & options
Operating System: AIX	5.1D	
Java(TM) 2 Runtime Environment	1.3.0	<code>javac -O</code>
C for AIX compiler (Visual Age C)	6.0	<code>xlc_r -O3 -qarch=pwr4</code>
XL Fortran Compiler	8.1	<code>xlf90_r -O3 -qarch=pwr4</code>
XL Fortran Compiler	8.1	<code>xlf_r -O3 -qarch=pwr4</code>

Table 3: Summary of the computing environment

Label	Array size	Memory size
A	1000000	≈ 4 MB
B	5000000	≈ 20 MB
C	25000000	≈ 100 MB
M	250000	≈ 1 MB

Table 4: Data sizes for the HeapSort benchmark

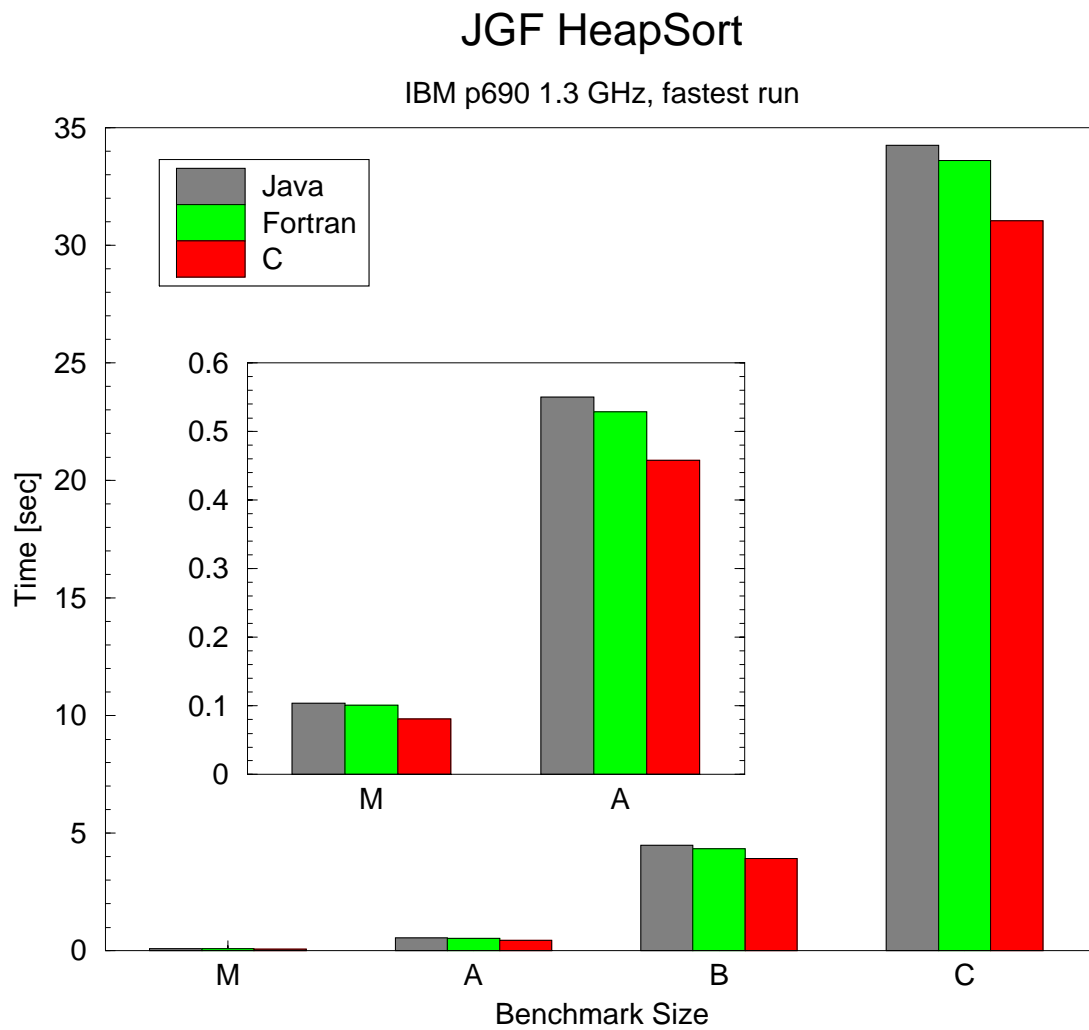


Figure 1: Execution times for the HeapSort benchmark

Label	Array size	Memory size
A	501×500	≈ 2 MB
B	1001×1000	≈ 8 MB
C	2001×2000	≈ 32 MB
M	351×350	≈ 1 MB

Table 5: Data sizes for the LUfact benchmark

Label	# particles	Memory per array	Total memory
A	2048	≈ 49 kB	≈ 147 kB
B	8788	≈ 210 kB	≈ 630 kB

Table 6: Data sizes for the LUfact benchmark

All runs have been performed on the same Lpar with the whole frame¹ reserved for the task.

For this benchmark, the C code shows the fastest execution time, for all problem sizes. The Java code is only marginally slower than the Fortran code, however both are considerably slower than C. These differences are largest for the smallest size ‘M’, with a difference of around 25 % between Java and C execution times. This difference falls to around 10 % for the largest problem size ‘C’.

Overall the differences are remarkably small. The fact Fortran is significantly slower than C is interesting, as the back end of the compiler is the same for C and Fortran. The fact that the size ‘M’ fits into L2 does not result in dramatic performance improvements. The reason for this is likely due to the branch prediction involved in the heapsort algorithm. Within this, there is little work to be done apart from loads and stores, with the machine spending a lot of its time waiting for the outcome of if statements.

3.2 LUFact

This benchmark transforms a matrix of double precision floating point numbers into upper triangle form. Again all sizes provided with the Java Grande suite are too large to fit into L2 cache. We define an additional ‘M’ size to fit L2. Details of the individual sizes are given in table 5.

As with the previous benchmark, the results were repeated 40 times on an empty frame. Figure 2 shows the fastest measured run time of these 40 repeats, for each data size.

For the smallest data size ‘M’, the Fortran version is fastest, with the C version around 30 % slower and the Java version around 3 times slower. However, as the problem size increases the gap in performance reduces, with the C version marginally faster (2 %) than the Fortran version for the largest problem size ‘C’. The Java version is around 10 % slower than the Fortran version on this problem size.

3.3 MolDyn

This benchmark performs a molecular dynamic simulation of particles in a Lennard-Jones potential. For each particle the position, velocity and force acting on it are required, resulting in 9 double precision numbers per particle. This Benchmark consists of two data sizes, both sizes fit into L2 cache. Details of the data sizes can be found in table 6.

¹a frame consists of 4 Lpars, and 32 processors

JGF LUFact

IBM p690 1.3 GHz, fastest run

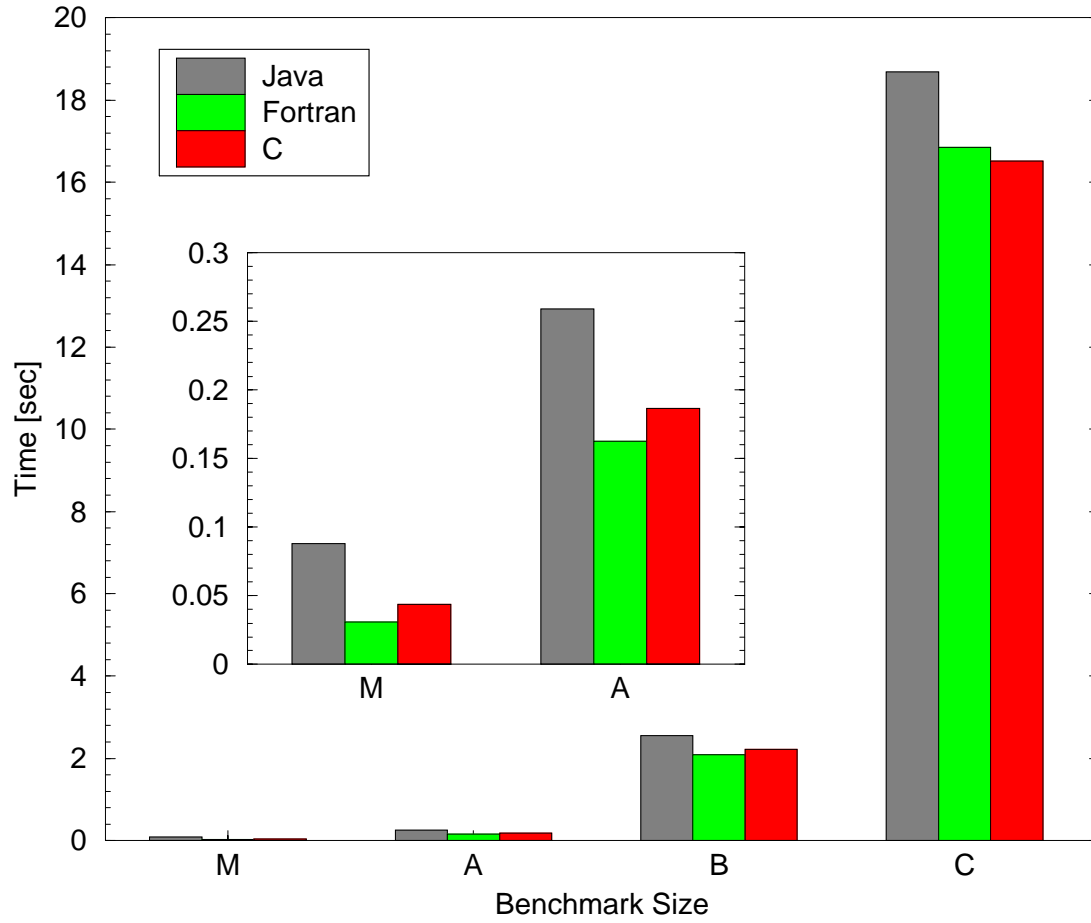


Figure 2: Execution times for the LUFact benchmark

JGF MolDyn

IBM p690 1.3 GHz, fastest run

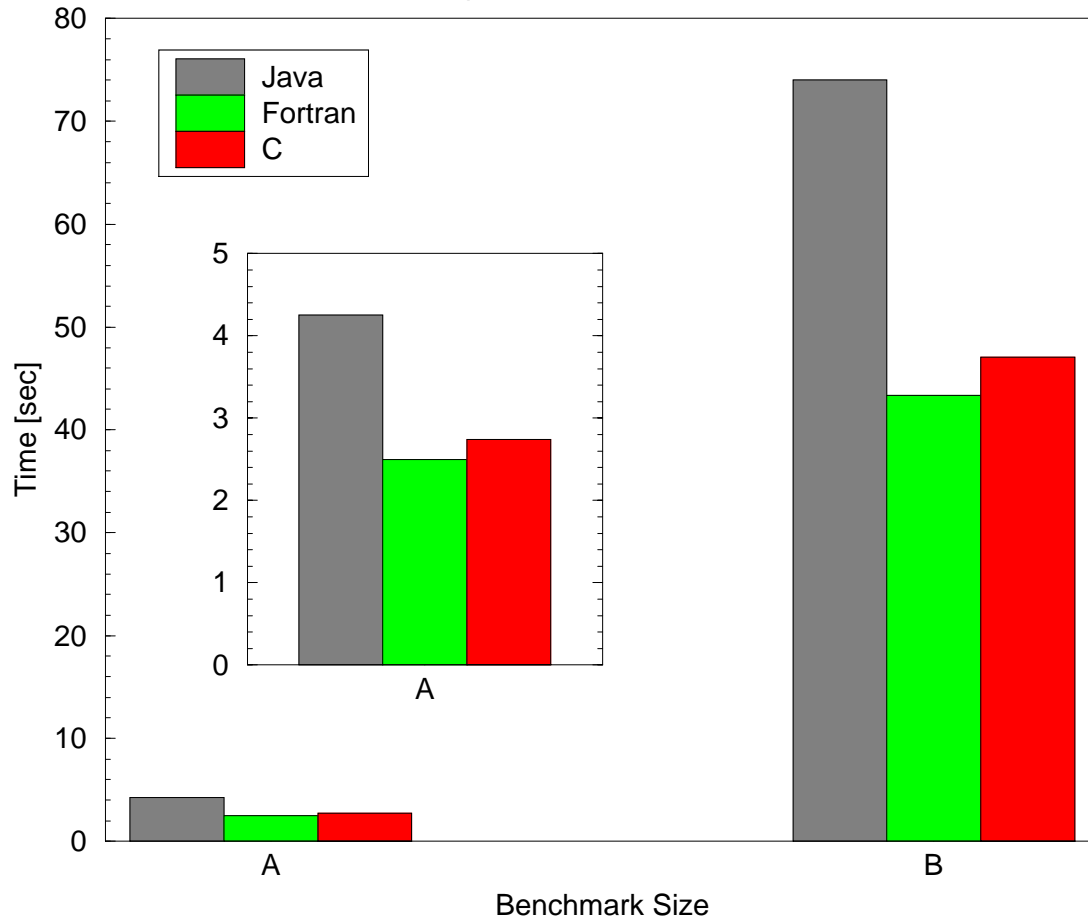


Figure 3: Execution times for the MolDyn benchmark

In this case the benchmark was run on a single Lpar, with the other Lpars of the frame running user processes. The results were repeated 40 times.

Figure 3 shows the fastest measured run time of these 40 repeats, for each data size.

For this benchmark, the relative performance of Java is poor. The benchmark holds little data but is quite intense with respect to computation. For both data sizes, the Fortran benchmark is around 10 % faster than the C. The Java performance is considerably worse, requiring around 70 % longer than Fortran.

4 Runtime variation

4.1 Scattering of run times

When doing the comparison it became apparent that the fluctuations of the run time for the Mol-Dyn Benchmark were about a factor 10 smaller than for the other two benchmarks. The obvious

Run time variation

IBM p690 1.3 GHz

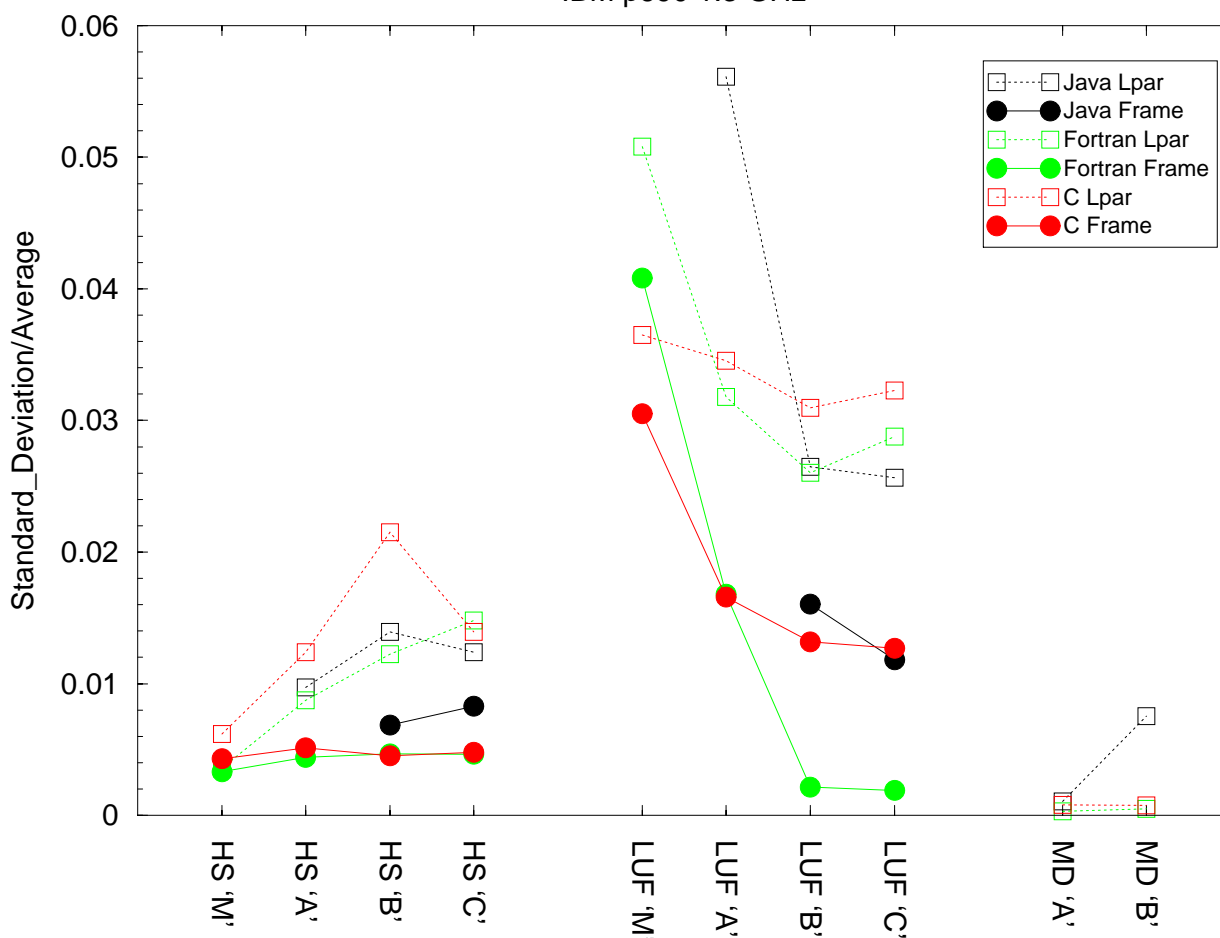


Figure 4: Runtime variations

question is: *Why is that?* The above analysis reveals that all the MolDyn sizes fit into L2, where as the standard sizes 'A', 'B' and 'C' for the other two benchmarks do not fit L2 and have to use L3 as well. To check whether the access to L3 cache is indeed causing runtime variation, we introduced the 'M' sizes for these benchmarks. Another question is about the source of the variation.² Looking into the details of the architecture reveals that the four Lpars of a frame are not as independent as one hopes. In fact, things like cache snooping and checking for cache coherency between the Lpars of a frame are still active. In an attempt to understand these runtime variations, we have carried out a detailed study. This compares the runtime variation when using one processor of an Lpar and leaving the other Lpar to other users, against reserving a whole frame and using a single node for the benchmark, which should cause as minimal disturbance by the other Lpars as possible. The results are shown in figure 4.

Figure 4 gives the results from 5 separate run sequences. Each sequence consisted of a series of runs for a single benchmark, beginning with the three Java data size (smallest - largest) before

²Please note that these serial benchmarks make no use of the interconnect. Hence, the interconnect can be ruled out as a source of variation.

repeating the progress with the Fortran and C benchmarks. These runs were then repeated at least 40 times before the job step was finished. The aim of this procedure was to provide meaningful comparisons between the sizes and the languages for a single benchmark. Because of the small variation of the MolDyn benchmark, we did execute these on a single frame. Because of the inaccurate timer used for the Java versions (1 ms), we did not measure meaningful variations for the small Java benchmarks. Those numbers are excluded from the figure.

4.2 Observations

Let us start with the observation that the single frame runs (full lines and circles) are indeed less noisy than the runs with other user codes running on other Lpar of the same frame. Please note that reproducing the single Lpar numbers is tricky, since we have no control what was running on the other Lpars.

The second observation is that, for 'M' size benchmarks, which fit into L2, the relative increase in run time variation when allowing other codes on the Lpars of the same frame is smaller than for the large 'B' and 'C' versions. It seems that access to L3 is indeed causing greater runtime variation.

5 Conclusion

We present a comprehensive comparison of the performance of Fortran, C and Java on a p690 system. For problems which are severely limited by the bandwidth from L3 to the MCM the Java runtime environment proved to generate efficient code, which is quite comparable in performance with that achievable with Fortran or C. For the Molecular dynamics code, the performance of Java is worse, being around 70% slower.

When carrying out this study we observed large variations in run times. It is clear that these variations depend on the usage of the other Lpars in the same frame. Applications accessing L3 cache are more susceptible to these variations than those only accessing L2.