CAL and the Computer Language Benchmarks Game

Magnus Byne

Last modified: September 19, 2007

Overview

This document describes the results of implementing the Computer Language Benchmarks Games (formerly Shootout) benchmarks (http://shootout.alioth.debian.org/) in CAL. The results demonstrate that the performance of CAL is similar to that of Java for this benchmark suite. The CAL benchmarks are now included on the official website.

Micro-benchmarks

No single set of benchmarks can accurately gauge the performance of a compiler for all application domains. However, the Computer Language Benchmarks Games do cover a broad range of tasks. The benchmarks included in the set have been contributed by a number of different sources, and are not tailored to the functional or Java communities. In implementing these benchmarks we seek to demonstrate that CAL can produce very efficient programs for a variety of tasks.

Document layout

This document is divided into the following sections:

System Configuration describes the setup used for running and timing the benchmarks.

Benchmark Descriptions gives brief descriptions of the benchmarks.

The timings and comments for the benchmarks are then split into two sections: Single Threaded Benchmarks, and Multithreaded Benchmarks.

Call for feedback

Any feedback you might have to help improve this document or the benchmark implementations is very welcome. Please email any comments or suggestions to Bo Ilic (bo.ilic@businessobjects.com) or Magnus Byne (magnus.byne@businessobjects.com).

System Configuration

All of the timings in this document were recorded using the following system configuration:

Java 1.6.0 (build 1.6.0_02-b06) (using the –server option)

Open Quark 1.5.1

Microsoft Windows XP Professional, 2002, Service Pack 2.

Dual Intel® Xeon™ CPU, 2.8GHz with 2GB of RAM.

Benchmark Descriptions

The Computer Language Shootout benchmarks consists of 18 main benchmarks, these are:

fannkuch	Indexed-access to tiny integer-sequence
recursive	Naive recursive-algorithms: ack, fib, tak
fasta	Generate and write random DNA sequences
k-nucleotide	Hashtable update and k-nucleotide strings
meteor-contest (new)	Search for solutions to shape packing puzzle
partial-sums	Naive iterative summation: power sin cos
pidigits	Streaming arbitrary-precision arithmetic
nsieve	Indexed-access to boolean-sequence
regex-dna	Match DNA 8-mers and substitute nucleotides for IUB codes
nsieve-bits	Indexed-access to bit-values
mandelbrot	Generate Mandelbrot set portable bitmap file
n-body	Double-precision N-body simulation
cheap-concurrency	Send messages between linked threads
spectral-norm	Eigenvalue using the power method
chameneos	Symmetrical thread rendez-vous requests
reverse-complement	Read DNA sequences - write their reverse-complement
binary-trees	Allocate and deallocate many many binary trees
sum-file	Read lines, parse and sum integers

Full benchmark descriptions can be found on the web site:
http://shootout.alioth.debian.org/

In the following discussion we address the benchmarks in two groups: multithreaded (chameneos and cheap-concurrency) and single threaded.

Benchmark Implementations

The Java implementations used for comparison in this document are the fastest implementations that were available on The Computer Language Benchmarks Game web site at the time of writing, with the single exception of the PiDigits implementation. In this case the fastest Java implementation is based on GNU MP, a C library for efficient arbitrary precision arithmetic. We chose the fastest pure Java version, as it provides a more direct and portable comparison between CAL and Java. The only changes made to benchmarks were to repackage within the Open Quark namespace. These implementations are included with the Open Quark distribution in the `org.openquark.cal.benchmarks.shootout.java` package.

The algorithms used in the Java and CAL versions are very similar, as they are quite precisely dictated by the rules of the benchmark site. The purpose of benchmarks is not to measure the efficiency of the benchmark implementations, but rather the underlying performance of the languages.

Single Threaded Benchmarks

The table below shows the timings for the CAL and the Java implementations.

Benchmark	CAL (ms)	Java (ms)	Ratio
binarytrees	1843	1465	0.79
fannkuch	14159	10673	0.75
fasta	12862	8857	0.69
knucleotide	9871	7177	0.73
mandelbrot	2506	2147	0.86
nbody	19156	13531	0.71
nsieve	586	564	0.96
nsievebits	1453	1368	0.94
partialsums	5159	6885	1.33
pidigits	5503	5710	1.04
recursive	2109	2475	1.17
regexdna	7506	7154	0.95
spectralnorm	25647	17833	0.70
sumcol	4493	4611	1.03
revcomp	887	1050	1.18
meteor	968	1071	1.11
Geometric Mean of Ratios			0.91

How the timings were obtained

The timings for both the Java and the CAL implementations were obtained using the ICE performance timing (:pt) command. This evaluates a function a number of times (10 in this case) without restarting the JVM, and computes the average time to evaluate the function, excluding the first evaluation. This means that the timings are not affected by initial startup or class loading, and take advantage of the JVM's dynamic profiling and optimization. This corresponds with how CAL has been used at Business Objects – as part of long running server applications.

However, it is different to the approach taken by the Computer Language Benchmarks Game web site, were the timings given are the minimum of three separate runs with a 'cold' JVM.

Comments

The CAL language is tightly linked to Java and the JVM. In many of the CAL benchmark implementations Java classes and primitives are imported and used, for example for input/output, mutable arrays, and mutable hash maps. The use of mutable arrays and maps is essentially required by the benchmark specifications. The implementations serve to show how these constructs can be used effectively within a CAL program. The timings show that the CAL code is compiled to very efficient Java, which in some cases even exceeds the performance of the hand coded Java implementations.

Multithreaded Benchmarks

Open Quark has the ability to evaluate CAL expressions concurrently on a single execution context. This feature is enabled using the VM argument org.openquark.cal.machine.lecc.concurrent_runtime. The parallel library defines primitives for evaluating expressions concurrently, e.g. parallelMap applies the supplied function to elements in a list concurrently. This library, combined with the principles of functional programming, provides a very clear and concise way to represent concurrent algorithms. This has been used in the implementation of the two multithreaded benchmarks.

The table below shows the timings obtained for the two multithreaded benchmarks:

Benchmark	Quark (ms)	Java (ms)	Ratio
Cheap-concurrency	2974	5641	1.90
Chameneos	4448	9531	2.14
Geometric Mean			2.02

How the timings were obtained

The timings for these benchmarks were obtained by running the programs 10 times and averaging the time taken. In the java version of cheap-concurrency, the fastest implementation on the benchmark website explicitly calls exit to shutdown all background threads, so the JVM had to be restarted after each run of the program.

Comments

These timings convincingly show the effectiveness of concurrent CAL – the CAL programs are roughly twice as quick as the Java versions. It may be that there is scope for improvement to the Java implementations, or that the implementations do not perform well on the dual processor hyper-threaded machine used for testing. However, these timings are for the fastest accepted versions on the website at time of writing.

Conclusions

The timings show that CAL provides a very fast functional language implementation; on average the performance is within about 10% of Java's on this set of benchmarks. As CAL is implemented on the JVM, it provides a fast and portable platform for functional programming, and it will benefit from ongoing improvements to the JVM. The concurrent benchmark implementations demonstrate how effective CAL is for expressing concurrent programs.

Appendix

In this section we include the previous results for Open Quark 1.5.0, and describe the changes since then.

Since 1.5.0 was released several of the Java benchmarks have been improved, most notably Mandelbrot, Revcomp , Nsievebits (e.g. Nsievebits now uses a custom bitset implementation, which avoids the time penalty for error checking included in the Java library implementation). This is reflected in an improvement between the new Java timings given above and those measured for the Open Quark 1.5.0 release.

The corresponding CAL benchmarks have been updated to incorporate similar improvements. This helps to keep the algorithms similar and allow the benchmarks to highlight differences in languages, rather than the benchmark implementations. The CAL benchmarks have also been updated to make use of a new console library which has been included with Open Quark 1.5.1. The relative changes between the performance of CAL and Java remain small.

The 1.5.1 release has a number of enhancements that improve the startup time and memory footprint of applications deployed as standalone JARs. However, the startup time is excluded in our benchmarking methodology, so this does not affect the results presented in this document.

Benchmark	CAL (ms)	Java (ms)	% Difference
binarytrees	1,878	1,466	-28.10
fannkuch	13,137	10,458	-25.62
fasta	13,477	8,777	-53.55
knucleotide	11,270	6,923	-62.79
mandelbrot	2,638	3,833	+31.18
nbody	21,925	13,526	-62.10
nsieve	574	553	-3.80
nsievebits	3,487	3,663	+4.80
partialsums	5,121	5,359	+4.44
pidigits	5,590	5,928	+5.70
recursive	2,093	2,418	+13.44
regexdna	7,262	6,908	-5.12
spectralnorm	25,437	17,654	-44.09
sumcol	4,654	4,583	-1.55
revcomp	1,293	1,652	+21.73
meteor	836	1,038	+19.46
Average			-11.62

Quark 1.5.0 Results