Exercises week 4
Friday 22 September 2017

Goal of the exercises

The goal of this week's exercises is to make sure that you can conduct meaningful performance measurements of Java programs, know that measurements can vary widely between apparently similar platforms, and can discuss observed strangenesses in timing results.

Do this first

The exercises build on the lecture note *Microbenchmarks in Java and C#* and the accompanying example code. Carefully study the hints and warnings in section 7 of that note before you measure anything. **NEVER measure anything from inside an IDE or when in Debug mode**.

Download and unpack the Java example code from file benchmarks-java.zip as indicated in the *Microbenchmarks* note, section 12.

Get and unpack this week's example code in zip file pcpp-week04.zip on the course homepage.

Exercise 4.1 In this exercise you must perform, on your own hardware, some of the single-threaded measurements done in *Microbenchmarks* note.

1. Run the `Mark1` through `Mark6` measurements yourself, and save results to text files. Use the `SystemInfo` method to record basic system identification, and supplement with whatever other information you can find about the execution platform. On Linux you may use `cat /proc/cpuinfo`; on MacOS you may use Apple > About this Mac; on some versions of Windows you may use Start > Control panel > System and security > System > View amount of RAM and processor speed.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in *Microbenchmarks*.

2. Use `Mark7` to measure the execution time for the mathematical functions `pow`, `exp`, and so on, as in *Microbenchmarks* section 4.2. Record the results in a text file along with appropriate system identification. Preferably do this on at least two different platforms, eg. your own computer and a fellow student's, or some computer at the university.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in *Microbenchmarks*.

Exercise 4.2 In this exercise you must perform, on your own hardware, the measurement performed in the lecture using the example code in file TestTimeThreads.java.

1. First compile and run the timing code as is, using `Mark6`, to get a feeling for the variation and robustness of the results. Do not hand in the results but discuss any strangenesses, such as large variation in the time measurements for each case.

2. Now change all the measurements to use `Mark7`, which reports only the final result. Record the results in a text file along with appropriate system identification.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in the lecture.

Exercise 4.3 In this exercise you must use the benchmarking infrastructure to measure the performance of the prime counting example given in file TestCountPrimesThreads.java.

1. Measure the performance of the prime counting example on your own hardware, as a function of the number of threads used to determine whether a given number is a prime. Record system information as well as the measurement results for 1...32 threads in a text file. If the measurements take excessively long time on your computer, you may measure just for 1...16 threads instead.

2. Use Excel or gnuplot or Google Docs online or some other charting package to make graphs of the execution time as function of the number of threads.

3. Reflect and comment on the results; are they plausible? Is there any reasonable relation between the number of threads that gave best performance, and the number of cores in the computer you ran the benchmarks on? Any surprises?

4. Now instead of the LongCounter class, use the java.util.concurrent.atomic.AtomicLong class for the counts. Perform the measurements again as indicated above. Discuss the results: is the performance of AtomicLong better or worse than that of LongCounter? Should one in general use adequate built-in classes and methods when they exist?

5. Now change the worker thread code in the lambda expression to work like a very performance-conscious developer might have written it. Instead of calling `lc.increment()` on a shared thread-safe variable `lc` from all the threads, create a local variable `long count = 0` inside the lambda, and increment that variable in the for-loop. This local variable is thread-confined and needs no synchronization. After the for-loop, add the local variable's value to a shared AtomicLong, and at the end of the `countParallelN` method return the value of the AtomicLong.

   This reduces the number of synchronizations from several hundred thousands to at most `threadCount`, which is at most 32. In theory this might make the code faster. Measure whether this is the case on your hardware. Is it? (It is not faster on my Intel-based MacOS laptop).

   (Optional) Can you think of any possible explanations for the few-synchronizations code not being faster than the original many-synchronizations code?

Exercise 4.4 Consider again the cache implementations in TestCache.java from week 2, applied to the prime factorization problem in exercise 2.4. Use the same thread count 16 and threads numbered $t = 0 \ldots 15$ as in that exercise. To make the measurements faster, you may reduce the amount of work that each thread must perform, so that thread t computes the factors of 4 000 numbers, namely the factors of the 2 000 numbers from 10 000 000 000 to 10 000 001 999 and also of the 2 000 numbers from $10\,000\,002\,000 + t \cdot 500$ to $10\,000\,003\,999 + t \cdot 500$.

1. Use the `Mark7` function to measure and report the execution time when wrapping the Factorizer class as a Memoizer1 instance.

2. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer2 instance.

3. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer3 instance.

4. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer4 instance.

5. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer5 instance.

6. Similarly, measure and report the execution time when wrapping the Factorizer class as a Memoizer0 instance.

7. Reflect on the results of the measurements you made. Which cache implementation performs best in this particular application: factorization of relatively large numbers, 16 threads working on partially overlapping ranges of such numbers? Does this result agree with the lecture's and Goetz's development of the cache classes?

8. What experiment would you set up to compare the scalability of the different cache implementations? You do not have to actually make that experiment, just describe it.