Exercises week 4

Last update 2020-09-14

Goals

The goals of this week is that you:

- Understand the challenges and pitfalls in benchmarking Java code
- Have benchmarked Java code for a number of Java concepts including: object creation, threads, locks and a sorting algorithm.
- Are able to benchmark your own algorithms/methods written in Java
- Understand the statistics of benchmarking (normal distribution and mean)
- Understand floating point representation and loss of precision

In these exercises *Microbenchmarks note* is the note by Peter Sestoft: *Microbenchmarks in Java and C sharp*. that can be found in the folder (CodeForBenchmarkNote) with course material for week 4 of the Fall 2020 course Practical and Concurrent and Parallel Programming.

Do this first

The exercises build on the lecture note *Microbenchmarks note* 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.

All the Java code listed in the *Microbenchmarks note*, the lecture and these exercises can be found on the Github page for the course (https://github.itu.dk/kasper/PCPP-Public) under week 4 in the directory: `CodeForBenchmarkNote` . - Download these.

You will run some (or all of) the measurements discussed in the *Microbenchmarks note* 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 your execution platform.

- on Linux you may use `cat /proc/cpuinfo`;
- on MacOS you may use Apple > About this Mac;
- on Windows 10 look in the System Information

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

Green
Run the `Mark1` through `Mark4` measurements. 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 *Microbenchmarks note*.

Yellow
Run the `Mark1` through `Mark5` measurements. 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 *Microbenchmarks note*.

Red
Run the `Mark1` through `Mark6` measurements. 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 *Microbenchmarks note*. Use `Mark7` to measure the execution time for the mathematical functions `pow`,

exp, and so on, as in *Microbenchmarks note* 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/frinds computer.

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 *Microbenchmarks*.

Exercise 4.2 In this exercise you will write code for calculating means and standard variation.

Green
In all the examples in Benchmark.java there are some lines like these:

```
      ...
      st += time;
      sst += time * time;
   }
   double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
   ...
```

Explain what is calculated here and relate to the formulas on the "slide" entitled *Variance* from the lecture.

Complete this primitive code for calculating the mean and standard variaton of the numbers in the array a:

```
class MeanVar {
  public static void main(String[] args) {
    double[] a=
      { 30.7, 30.3, 30.1, 30.7, 50.2, 30.4, 30.9, 30.3, 30.5, 30.8 };
    ...
    for (int j= 0; j<n; j++) {
      ...
    }
    System.out.printf("%6.1f ns +/- %6.3f%n", mean, sdev);
  }
}
```

Your hand-in must contain the results of running the completed program.

Now change the values in a to be: 30.7, 100.2, 30.1, 30.7, 20.2, 30.4, 2, 30.3, 30.5, 5.4 . Run the program with these numbers and include the results in your hand in.

Yellow
Assume that you include one more observation 25 in the sample (contents of array a). Is 25 an outlier in the first set of numbers? In the second?

Red
Write a more general Java program for calculating the mean and standard deviation of a large set of data in a file.

Exercise 4.3 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. *Green* First compile and run the thread 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. *Yellow* 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 4.4 In this exercise you must use the benchmarking infrastructure to measure the performance of the prime counting example given in file TestCountPrimesThreads.java.

1. *Green* 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. 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?
3. *Yellow* 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?
4. *Red* 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.

Exercise 4.5 In this exercise, we will analyze the performance of locks ("synchronized"). This lambda can be used in the benchmarking code (`Benchmark.java`)

```
"Uncontended lock",
      i -> {
        synchronized (obj) {
          return i;
        }
      }
```

Green
Use Mark6 to estimate the cost of getting an uncontended lock on your own hardware. Include the results in your hand-in and comment on them: Are they plausible? Any surprises?

Exercise 4.6 *Green*
This exercise is a follow up on the question asked in the PCPP forum: https://learnit.itu.dk/mod/forum/discuss.php?d=24089 . The question arose from measurement on this class:

```
class Exp {
    public long volExp (int N) {
        PerfTest pt = new PerfTest();
        Long start = System.nanoTime();
        for (int i = 0; i < N; i++) {
            pt.vInc();
        }
        return System.nanoTime()-start;
    }
    public long nonVolExp (int N) {
        PerfTest pt = new PerfTest();
        Long start = System.nanoTime();
        for (int i = 0; i < N; i++) {
            pt.inc();
        }
        return System.nanoTime()-start;
    }
}

public class PerfTest {
    private volatile int vCtr;
    private int ctr;

    public void vInc () {
        vCtr++;
    }

    public void inc () {
        ctr++;
    }
    public static void main(String[] args) {
        int[] input = {100_000_00,1000_000_00,Integer.MAX_VALUE/10};
        System.out.println("***VOLATILE EXPERIMENTS***");
        for (int n : input) {
            System.out.println("N=" + n);
            long vol = new Exp().volExp(n);
            System.out.println("Volatile time: " + vol);
            System.out.println("Volatile time per iteration: " + (float)
               vol / n);
        }
        System.out.println("\n");
        System.out.println("***NON-VOLATILE EXPERIMENTS***");
        for (int n : input) {
            System.out.println("N=" + n);
            long nonVol = new Exp().nonVolExp(n);
            System.out.println("Non volatile time: " + nonVol);
            System.out.println("Non Volatile time per iteration: " +
               (float) nonVol / n);
        }
        System.out.println("\n");
    }
}
```

The code attempts to measure the performance difference between a `volatile int` and a normal `int`.

Green

Use Mark6 (from `Bendchmark.java`) to compare the performance of incrementing a `volatile int` and a normal `int`. Include the results in your hand-in and comment on them: Are they plausible? Any surprises?

Red

Compile and run the PerfTest program (shown above). Compare the results of this program with the results from running Mark6. Are they consistent? If not, can you find a plausible explanation.