# Exercises week 3

Last update 2021/09/09

## 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, a sorting algorithm and an algorithm for computing prime factors
- Are able to benchmark your own algorithms/methods written in Java
- Understand the statistics of benchmarking (normal distribution, mean and variance).
- Are aware off 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 2021 course Practical and Concurrent and Parallel Programming.

## On-line algorithm for computing variance

On page 6 of the *Microbenchmarks note* is two formulas defining the average and variance $\mu$ and $\sigma$:

$$\begin{aligned}\mu &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} t_j \\ \sigma &= \sqrt{\tfrac{1}{n-1}\textstyle\sum_{j=1}^{n}(t_j-\mu)^2}\end{aligned}$$

These can be converted to an algorithm for computing the average and variance, by first having a loop computing the average $\mu$ followed by a second loop computing the variance $\sigma$. However, the code for `Mark4` (and all the following Mark X) uses a slightly different formula for computing variance $\sigma$ having only one loop. It is an example of an on-line algorithm The following derivation shows that the two formulas are equivalent:

$$\begin{aligned}
\mu &= \tfrac{1}{n}\textstyle\sum_{j=1}^{n} t_j \\
\sigma &= \sqrt{\tfrac{1}{n-1}\textstyle\sum_{j=1}^{n}(t_j-\mu)^2} \\
\sigma &= \sqrt{\tfrac{1}{n-1}\textstyle\sum_{j=1}^{n}(t_j^2+\mu^2-2t_j\mu)} \\
\sigma^2 &= \tfrac{1}{n-1}\textstyle\sum_{j=1}^{n}(t_j^2+\mu^2-2t_j\mu) \\
\sigma^2 &= \tfrac{1}{n-1}(\textstyle\sum_{j=1}^{n} t_j^2+\sum_{j=1}^{n}(\mu^2-2t_j\mu)) \\
\sigma^2 &= \tfrac{1}{n-1}(\textstyle\sum_{j=1}^{n} t_j^2+n\mu^2-2\mu\sum_{j=1}^{n} t_j) \\
\sigma^2 &= \tfrac{1}{n-1}(\textstyle\sum_{j=1}^{n} t_j^2+n\mu^2-2\mu n\mu) \\
\sigma^2 &= \tfrac{1}{n-1}(\textstyle\sum_{j=1}^{n} t_j^2-n\mu^2) \\
\sigma^2 &= \tfrac{1}{n(n-1)}(n\textstyle\sum_{j=1}^{n} t_j^2-\mu^2) \\
\sigma^2 &= \tfrac{1}{n(n-1)}(n\textstyle\sum_{j=1}^{n} t_j^2-(\tfrac{1}{n}\sum_{j=1}^{n} t_j)^2)
\end{aligned}$$

## Do this first

The exercises build on the lecture, the *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/jst/PCPP2021-public) in the directory week03.

You will run some 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 3.1** In this exercise you must perform, on your own hardware, some of the measurements done in the *Microbenchmarks note*.

*Mandatory*
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/friends 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 3.2** In this exercise you must perform, on your own hardware, the measurement performed in the lecture using the example code in file TestTimeThreads.java.

*Mandatory*

1. 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. 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 3.3** In this exercise you must use the benchmarking infrastructure to measure the performance of the prime counting example given in file TestCountPrimesThreads.java.

*Mandatory*

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. 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. 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?

*Challenging*

4. 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 3.4** In this exercise you should estimate whether there is a performance gain by declaring a shared variable as volatile. Consider this simple class that has both a volatile `int` and another `int` that is not declared volatile:

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

  public void vInc () {
      vCtr++;
  }

  public void inc () {
      ctr++;
  }
}
```

*Mandatory*

Use Mark7 (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?