Exercises week 5

Last update 2022/09/24

Goals

The goals of this week are related to the material in the *Microbenchmarks note* by Peter Sestoft: *Microbenchmarks in Java and C sharp* that can be found in the folder (CodeForBenchmarkNote) with course material for week 5 of the Fall 2022 course Practical and Concurrent and Parallel Programming.

The goals are:

- Identify 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
- Apply benchmarking to the students own algorithms/methods written in Java
- Recognize the statistics of benchmarking (normal distribution, mean and variance).
- Recognize floating point representation and loss of precision.

On-line algorithm for computing variance

On page 6 of the *Microbenchmarks note* are two formulas defining the average and variance μ and σ:

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

These can be converted to an algorithm for computing the average and variance, by first having a loop computing the average μ followed by a second loop computing the variance σ. However, the code for `Mark4` (and all the following Mark X) uses a slightly different formula for computing variance σ having only one loop. It is an example of an on-line algorithm in which both the average and variance μ and σ are computed in a single loop. 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 (week05). The code for the exercises is in `.../week05/code-exercises/...`.

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

Mandatory

1. Use `Benchmark.java` to 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*.

2. 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 5.2 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 5.3 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 (defining the computation of the thread), 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 5.4 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:

```
public class TestVolatile {
  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?

Exercise 5.5 In this exercise you must write code searching for a string in a (large) text. Such a search is the core of any web-crawling service such as Google, Bing, Duck-Go-Go etc. Later in the semester, there will be a guest lecture from a Danish company providing a very specialized web-crawling solution that provides search results in real-time.

In this exercise you will work with the nonsense text found in:
`src/main/resources/long-text-file.txt` (together with the other exercise code). You may read the file with this code:

```
final String filename = "src/main/resources/long-text-file.txt";
...
public static String[] readWords(String filename) {
  try {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    return reader.lines().toArray(String[]::new);//will be explained in Week 7
  } catch (IOException exn) { return null;}
}
```

`readWords` will give you an array of lines, each of which is a string of (nonsense) words.

The purpose of the code you are asked to write is to find all occurences of a particular word in the text. This skeleton is based on sequentially searching the text (i.e. one thread). You may find it in the `code-exercises` directory for Week05.

```
public class TestTimeSearch {
  public static void main(String[] args) { new TestTimeSearch(); }
  public TestTimeSearch() {
    final String filename = "src/main/resources/long-text-file.txt";
    final String target= "ipsum";

    final LongCounter lc= new LongCounter();
    String[] lineArray= readWords(filename);

    System.out.println("Array Size: "+ lineArray.length);
    System.out.println("# Occurences of "+target+ " :"
           +search(target, lineArray, 0, lineArray.length, lc));
  }

  static long search(String x, String[] lineArray, int from,
                                        int to, LongCounter lc){
    //Search each line of file
    for (int i=from; i<to; i++ ) lc.add(linearSearch(x, lineArray[i]));
    return lc.get();
  }
  static long linearSearch(String x, String line) {
    //Search for occurences of c in line
    String[] arr= line.split(" ");
    long count= 0;
    for (int i=0; i<arr.length; i++ ) if ( (arr[i].equals(x)) ) count++;
    return count;
  }
}
```

Mandatory

1. `TestTimeSearch` uses a slightly extended version of the `LongCounter` where two methods have been added `void add(long c)` that increments the counter by `c` and `void reset()` that sets the counter to 0.

   Extend `LongCounter` with these two methods in such a way that the counter can still be shared safely by several threads.

2. How many occurencies of "ipsum" is there in `long-text-file.txt`. Record the number in your solution.

3. Use Mark7 to benchmark the search function. Record the result in your solution.

4. Extend the code in `TestTimeSearch` with a new method

   ```
   private static long countParallelN(String target,
                       String[] lineArray, int N, LongCounter lc) {
   // uses N threads to search lineArray
     ...
   }
   ```

Fill in the body of `countParallelN` in such a way that the method uses N threads to search the lineArray. Provide a few test results that make i plausible that your code works correctly.

5. Use Mark7 to benchmark `countParallelN`. Record the result in your solution and provide a small discussion of the timing results.

Exercises week 6

Last update 2022/10/06

Goal of the exercises

The goals of the exercises for this week are:

- Gaining practical experience in using threads to improve the scalability and performance of concurrent programs.
- Using Java Executors and Futures.
- Using lock striping.

Exercise 6.1 This exercise is based on the program `AccountExperiments.java` (in the exercises directory for week 6). It generates a number of transactions to move money between accounts. Each transaction simulate transaction time by sleeping 50 milliseconds. The transactions are randomly generated, but ensures that the source and target accounts are not the same.

Mandatory

1. Use Mark7 (from Benchmark.java in the `benchmarking` package) to measure the execution time and verify that the time it takes to run the program is proportional to the transaction time.
2. Now consider the version in `ThreadsAccountExperimentsMany.java` (in the directory `exercise61`).

   The first four lines of the `transfer` method are:

   ```
   Account min = accounts[Math.min(source.id, target.id)];
   Account max = accounts[Math.max(source.id, target.id)];
   synchronized(min){
     synchronized(max){
   ```

   Explain why the calculation of `min` and `max` are necessary? Eg. what could happen if the code was written like this:

   ```
   Account s= accounts[source.id];
   Account t = accounts[target.id];
   synchronized(s){
     synchronized(t){
   ```

   Run the program with both versions of the code shown above and explain the results of doing this.
3. Change the program in `ThreadsAccountExperimentsMany.java` to use a the executor framework instead of raw threads. Make it use a fixed size thread pool. For now do not worry about terminating the main thread, but insert a print statement in the `doTransaction` method, so you can see that all executors are active.
4. Ensure that the executor shuts down after all tasks has been executed.

   Hint: See slides for suggestions on how to wait until all tasks are finished.

   .

Challenging

5. Use Mark8Setup to measure the execution time of the solution that ensures termination.

   Hint: Be inspired by `QuicksortExecutor.java` (in the `code-lecture` directory for week 6)

Exercise 6.2 Use the code in file `TestCountPrimesThreads.java` (in the exercises directory for week 6) to count prime numbers using threads.

Mandatory

1. Report and comment on the results you get from running `TestCountPrimesThreads.java`.
2. Rewrite `TestCountPrimesthreads.java` using Futures for the tasks of each of the threads in part 1. Run your solutions and report results. How do they compare with the results from the version using threads?

Exercise 6.3 A histogram is a collection of bins, each of which is an integer count. The span of the histogram is the number of bins. In the problems below a span of 30 will be sufficient; in that case the bins are numbered 0. . . 29.

Consider this Histogram interface for creating histograms:

```
interface Histogram {
  public void increment(int bin);
  public int getCount(int bin);
  public float getPercentage(int bin);
  public int getSpan();
  public int getTotal();
}
```

Method call `increment(7)` will add one to bin 7; method call `getCount(7)` will return the current count in bin 7;method call `getPercentage(7)` will return the current percentage of total in bin 7; method `getSpan()` will return the number of bins; method call `getTotal()` will return the current total of all bins.

There is a non-thread-safe implementation of Histogram1 in file SimpleHistogram.java. You may assume that the `dump` method given there is called only when no other thread manipulates the histogram and therefore does not require locking, and that the span is fixed (immutable) for any given Histogram object.

Mandatory

1. Make a thread-safe implementation, class `Histogram2` implementing the interface `Histogram`. Use suitable modifiers (`final` and `synchronized`) in a copy of the `Histogram1` class. This class must use at most one lock to ensure mutual exclusion.

   Explain what fields and methods need modifiers and why. Does the `getSpan` method need to be synchronized?
2. Now create a new class, `Histogram3` (implementing the `Histogram` interface) that uses lock striping. You can start with a copy of `Histogram2`. Then, the constructor of `Histogram3` must take an additional parameter `nrLocks` which indicates the number of locks that the histogram uses. You will have to associate a lock to each bin. Note that, if the number of locks is less than the number of bins, you may use the same lock for more than one bin. Try to distribute locks evenly among bins; consider the modulo operation `%` for this task.
3. Now consider again counting the number of prime factors in a number `p`. Use the `Histogram2` class to write a program with multiple threads that counts how many numbers in the range 0. . . 4 999 999 have 0 prime factors, how many have 1 prime factor, how many have 2 prime factors, and so on. You may draw inspiration from the `TestCountPrimesThreads.java`.

   The correct result should look like this:

```
0:         2
1:    348513
2:    979274
3:   1232881
4:   1015979
5:    660254
```

```
   6:    374791
   7:    197039
   8:     98949
   9:     48400
... and so on
```

showing that 348 513 numbers in 0. . . 4 999 999 have 1 prime factor (those are the prime numbers), 979 274 numbers have 2 prime factors, and so on. (The 2 numbers that have 0 prime factors are 0 and 1). And of course the numbers in the second column should add up to 5 000 000.

Hint: There is a class `HistogramPrimesThread.java` which you can use a starting point for this exercise. That class contains a method `countFactors(int p)` which returns the number of prime factors of `p`. This might be handy for the exercise.

.

4. Finally, evaluate the effect of lock striping on the performance of part 3. Create a new class where you use `Mark7` to measure the performance of `Histogram3` with increasing number of locks to compute the number of prime factors in 0. . . 4 999 999. Report your results and comment on them. Is there a increase or not? Why?

Challenging

3. Define a thread-safe class Histogram3 that uses an array of java.util.concurrent.atomic.AtomicInteger objects instead of an array of integers to hold the counts.

   In principle this solution might perform better, because there is no need to lock the entire histogram object when two threads update distinct bins. Only when two threads call `increment(7)` at the same time do they need to make sure the increments of bin 7 are atomic.

   Can you now remove `synchronized` from all methods? Why? Run your prime factor counter and check that the results are correct.