Exercises week 8

Last update 2024/10/10

These exercises aim to give you practical experience with performance measurements of:

- Java elements such as threads and volatile;
- algorithms exploiting concurrency to improve performance e.g. sorting and searching

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 week 8: (week08). The code for the exercises is in:
`.../week08/code-exercises/week08exercises/....`

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 8.1 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.
3. Use the same reasoning as in the lecture (slide "Thread create + start") to estimate the cost of creating a thread on your own computer. Include your result in the text files with answers for this exercise.

Exercise 8.2 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 8.3 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?

   Challenging

3. 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 `int 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 8.4 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.In the guest lecture (Week 7) you heard about 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 Week 8.

```
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.

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:

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