Parallelism and concurrency module practical

To do this practical you will need to use a computer with at least two cores under, preferably, Linux. If you can find a machine with four cores, even better.

Please indicate in your *answers.txt* file how many cores the machine you used has.

1. [5%] Implement a sequential Java program to calculate the sum of an array of longs.

   So that it can be tested, the program should set the size of the array to the user specified positive integer passed via the command line (i.e. args[0]) and initialize each element to 1, 2, 3 ... args[0] (but you most certainly should NOT use the formula $n * (n + 1)/2$ to calculate the sum of n consecutive numbers, because your code should work for any array of longs; this is just a test case so that we can check that your code works). Call this program SumSequential.java.

2. [10%] Implement a threaded Java version of the above. See page 21 of *A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency*.

3. [10%] Time each of the above programs several times on each of the following array sizes: 500000, 1000000, 10000000, 100000000, where each element in the array is equal to 1 plus its index position. i.e. arr[0] = 1, arr[1] = 2 etc. Hint: use the Linux *time* program. See http://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1 for an explanation of real versus user time (not interested in sys time for this project). Make sure nothing cpu-intensive is running on your computer at the same time.

   List in a table the average real time taken by each version of the program.

	Sequential	Threaded
500,000		
1,000,000		
10,000,000		
100,000,000		

4. [5%] What strange and irritating result do you notice (one short sentence please)?

5. [5%] Modify your sequential program so that instead of summing the array, it sums the number of factors greater than 1 and less than the array element, for each array element.

   E.g. the number of factors of 5 is 0, the number of factors of 9 is 1 (i.e. 3) and the number of factors of 10 is 2 (i.e. 2 and 5).

   So for an array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], the answer is 0 + 0 + 0 + 1 + 0 + 2 + 0 + 2 + 1 + 2 = 8.

   Call this program FactorSequential.java.

6. [10%] Now implement a threaded version of the above.

7. [10%] List in a table the average real time taken by each version of the program.

	Sequential	Threaded
500,000		
1,000,000		
2,000,000		

8. [5%] You shouldn't see the same strange and irritating result that you got in 4. Why? (One short sentence please) [5%]

9. [5%] Try changing the value of the SEQUENTIAL_THRESHOLD to 1, 10,000 and 1,000,000 and run with an array of size 1,000,000. What difference, if any, do you see? (one or, at most two, short sentences, please)

10. [35%] Implement *Conway's Game of Life*. See the Wikipedia entry. There are an enormous number of resources on the web to help you. You are encouraged to use these, but your code must be your own. Your user interface should support, at minimum, initialising the grid to the R-Pentomino or Gosper Glider Gun (both these are described in many places on the Internet). Two working versions of the Game of Life will be provided for you to experiment with, one text-based and one a GUI. You don't have to copy the user interface, but the closer yours is to one of these versions, the easier it will likely be to mark your work. You can choose to code either a text-based or GUI interface, but a text one will be less work, albeit less spectacular. Your grid should be at least 5,000 X 5,000, but of course you should only display a tiny fraction of it (e.g. 25 rows X 50 columns if you use a plain text text interface, but it's up to you). Strictly speaking you should handle the case of the grid not being big enough, but you won't be marked on this. However, you must keep

track of the population, because this will give you an opportunity to learn how to prevent a data race.

For a working multi-threaded Game of Life that is faster than a sequential version we'll provide: **30%**

If it's slower than the sequential version we provide you can get at most 50% for this question.

For a working multi-threaded version that is demonstrably faster than the multi-threaded one we'll provide: **5%**

This is how your marker will test the speed of your game: Using either the Gosper Glider Gun or R-Pentomino, the marker will run your program with, say, 1,000 generations (the text or GUI interface should not update until all 1,000 generations are run - see the example we provide).

Hand-in instructions

Hand in your programs as well as your answers to the questions in a plain text file called *answers.txt*. If you like, you may also hand in a README file with **BRIEF AND CLEAR** instructions for your marker on how to use your Game of Life program, but it's best if the interface is simple enough that your marker doesn't need instructions. Your marker will try to compile and run your Java programs as follows, so make sure you have named them correctly:

- javac SumSequential.java
- java SumSequential
- javac SumThreaded.java
- java SumThreaded
- javac FactorSequential.java
- java FactorThreaded.java
- javac GameOfLifeSequential.java
- java GameOfLifeSequential
- javac GameOfLife.java
- java GameOfLife