Exercises 11

Task 11.1. Atomic Money

Rewrite “Money” example using atomic data types found in the package `java.util.concurrent.atomic`. You will not need semaphore `M` anymore.

Task 11.2. The Cost of Synchronization

Thread safety consumes additional resources, so if you don’t need it, you shouldn’t pay for it. For example, collection classes `Vector` and `ArrayList` provide identical behavior except thread safety.

Perform measurements of `Vector` and `ArrayList` performance in a single-threaded application on your system. Test `set()` method only; add/remove operations are quite expensive regardless of synchronization. Report the results (e.g., *“ArrayList is 1.5 times faster than Vector”*) and explain how you got them.

Task 11.3. Game of Life

Use `ExecutorService` to parallelize the Game of Life (see Exercise `8.4 OpenMP Life`).

Use the following hints and suggestions:

1. Each thread should process a block no larger than `K×K` squares of the game board. You can hardcode `K` in your solution.
2. Use `Runtime.getRuntime().availableProcessors()` to identify the number of processors on your system, and thus the number of threads that can effectively work in parallel.
3. You can use the code of an `ExecutorService`-based prime-finding factory discussed in the lecture as an initial template for your solution.

Task 11.4. Forked Game of Life

Use Fork/Join framework to implement a parallel version of “Game of Life” from the previous task. The main procedure can work, for example, as follows:

1. If both sides of the game board are no larger than `K`, calculate its new state.
2. If the board is higher than `K` squares, divide it into top and bottom halves, and call the main procedure for each half.

3. If the board is wider than K squares, divide it into left and right halves, and call the main procedure for each half.

You can use the code of a Fork/Join-based prime-finding factory discussed in the lecture as an initial template for your solution.