Multithreaded Performance and Correctness

### Participants

* [Javier Matos](mailto:jmatosdenizac22@amherst.edu)
* Emmanuelle Sohn
* Chloe Wohlgemuth

### Background

In the first lab assignment, you were asked to write a multithreaded program to estimate pi using a Monte Carlo simulation. Each thread in the program generates many random points in a square, and reports back the number of sampled points that fall inside a disk. In order to “return” the number of such points each thread produces, the threads access a shared array. In this activity, you will examine some undesirable behavior that can occur when threads share objects if we are not careful.

### Part I: Bad Counter

To get started download, compile, and run the following files:

* [BadCounter.java](https://willrosenbaum.com/assets/java/mt-performace-correctness/BadCounter.java)
* [Counter.java](https://willrosenbaum.com/assets/java/mt-performace-correctness/Counter.java)
* [ThreadCount.java](https://willrosenbaum.com/assets/java/mt-performace-correctness/ThreadCount.java)

The main method is in BadCounter. The basic program creates a simple Counter object and creates a thread that increments the object 100 million times, reports the count, and how long it took to produce the count.

How long does it take the program to run (in milliseconds)?

|  |
| --- |
| 151 ms, 54 ms, 32 ms |

Now increase the number of threads to 2 by setting NUM\_THREADS = 2 in BadCounter.java. What output is produced? Is it what you’d expect?

|  |
| --- |
| 116 ms, 32 ms, 23 ms, |

Play around with different values of NUM\_THREADS (consider a range of values: 4, 8, 16, 32, 100, 1000…) and notice what happens with the reported count and the runtime. (Run the program multiple times for each choice of values.) Do you notice a pattern? Do others in your group observe the same pattern on their computers?

|  |
| --- |
| It is a near-negligible amount of time less going from 2 threads to 4 threads. Going from 4 to 8 and onwards, it starts taking longer to run. For the slowest computer, the increase in time was not negligible (116 to 75 ms), but then it began to increase again with 8 threads. |

Is this the behavior you’d expect from the program? How might you explain it? And how might you fix it?

|  |
| --- |
| We would expect more threads being helpful to an extent, but then eventually the computer doesn’t have enough threads. This leads to it conducting it just as sequentially as with the slightly fewer threads. The time to create and then terminate the non-needed threads also grows. |

### Part II: Slow Samples

Now download and run the following files:

* [RandomTester.java](https://willrosenbaum.com/assets/java/mt-performace-correctness/RandomTester.java)
* [RandomWorker.java](https://willrosenbaum.com/assets/java/mt-performace-correctness/RandomWorker.java)

This program generates many random numbers using a shared Random object and reports the elapsed time. (The threads don’t do anything with the random numbers, just compute them.)

With NUM\_THREADS set to 1, how long does your program take to execute?

|  |
| --- |
| 124 ms, |

Now run the program with several values of NUM\_THREADS as before. What pattern do you notice?

|  |
| --- |
| **2 threads**: 144 ms, 516 ms, 83 ms  **4 threads**: 1004 ms, 667ms  **8 threads**: 1274 ms, 882ms  **32 threads**: 1253 ms, 1035 |

Finally, modify RandomWorker.java so that it uses a thread local random number generator within the run method instead of the shared Random object. That is, replace the line

r.nextInt();

with

ThreadLocalRandom.current().nextInt();

The thread local random number generator creates a separate, independent number generator for each thread, rather than accessing the shared number generator passed to the constructor of the RandomWorker class.

Now run the program with various values of NUM\_THREADS. What pattern do you see? Is it a similar pattern to before? How might you explain the difference (if any)?

|  |
| --- |
| **1 thread**: 29 ms,  **2 threads**: 23 ms,  **4 threads**: 24 ms,  **8 threads**: 41 ms  **32 threads**: 110 ms, |