CS 131 Homework 3 Report

Paul Talma
UCLA
paultalma@ucla.edu

1 Approach

In order to automate testing and data collection, we defined a new class, `PerformanceTest.java`. This class defines the variables we wish to vary before running tests on all combinations of the variables. The results of the tests are automatically logged to a `.csv` file, for ease of analysis.

We considered the following values for each of the variables:

Variable	Values
`array_size`	5, 10, 20, 40, 60, 80, 100, 130
`num_threads`	1, 8, 16, 32, 40
`state_types`	Null, Synchronized, Unsynchronized
`thread_type`	Platform, Virtual

`nTransitions` was fixed at $100,000,000$, as per the spec. In addition, for each combination of variables, we ran the test harness three times and averaged the results to smooth out some of the noise. We carried out our experiments on `lnxsrv11` and on our personal machine (Apple M2), obtaining qualitatively similar results. The results reported here were obtained on the Linux machine.

1.1 AcmeSafe Implementation

We chose to implement our `AcmeSafe` class using the `java.util.concurrent.atomic` package. This package provides classes that support atomic read and write operations, ensuring thread-safe access to variables without resorting to locks. To implements `AcmeSafe`, we use the `AtomicLongArray` class, calling `getAndDecrement/getAndIncrement` to update the array.

2 Results

To decrease clutter, we omit results for the `Null` class from the graphs. In each case, the `Null` class took the least amount of time and produced an array sum of 0, as expected.

2.1 Performance

As we can see in Figure 1, the `Unsynchronized` class performs much better than the `Synchronized` class. On small arrays (size 5), the `AcmeSafe` class performs worse than the other two, while on larger arrays (size 100), its performance is close to that of the `Unsynchronized` class. The performance profile of the synchronized approach is unsurprising: as the number of threads increases, the opportunities for lock contention increase, and hence the time taken does too. By contrast, increasing the number of threads does not significantly impact the performance of the `Unsynchronized` class. The AcmeSafe approach sits somewhere in between: on a small array, it must deal with more lock contention and hence takes more time as the number of threads increases. On a longer array, the atomic operations are less likely to collide, taking greater advantage of the parallelism. Note that the performance of the virtual threads is generally better than that of the platform threads.

Figure 2 provides more insight into the performance of the different approaches. We see that increasing the number of threads results in a modest increase in average time-per-swap for the `Unsynchronized` class and a slightly greater increase for the `AcmeSafe` class. By contrast, increasing the thread count results in a large increase in average time-per-swap for the `Synchronized` class. This is to be expected, since a greater number of threads increases the chances of a lock contention, and hence of a slowdown, in the synchronized implementation. This is also true for the AcmeSafe approach, but since collisions are less costly (the CPU spins instead of blocking a whole thread), the effect is less pronounced.

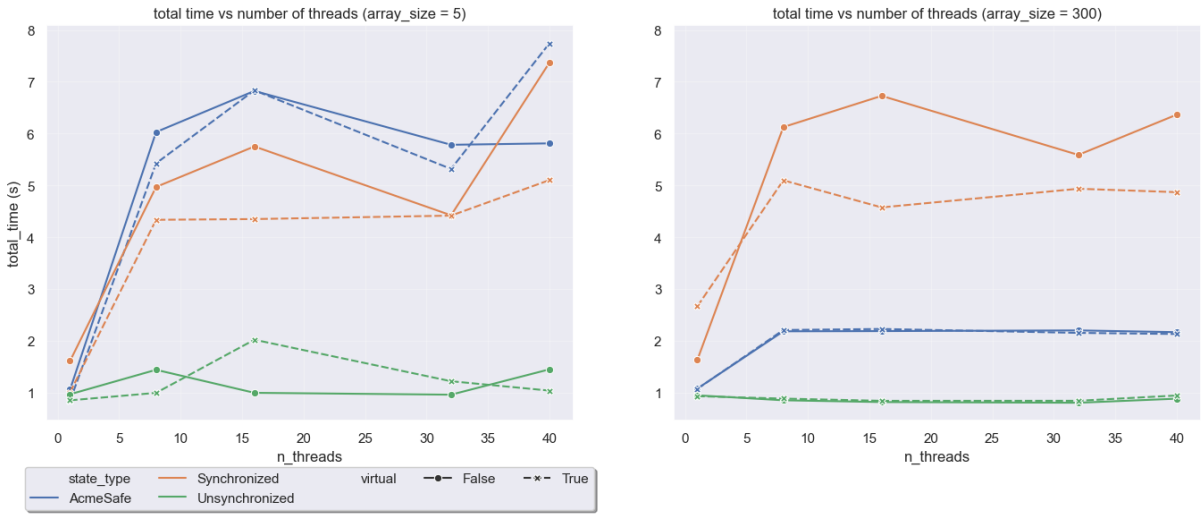

Figure 1: Total time vs. number of threads, with an array size of 300. Solid lines indicate platform threads; dashed lines, virtual threads.

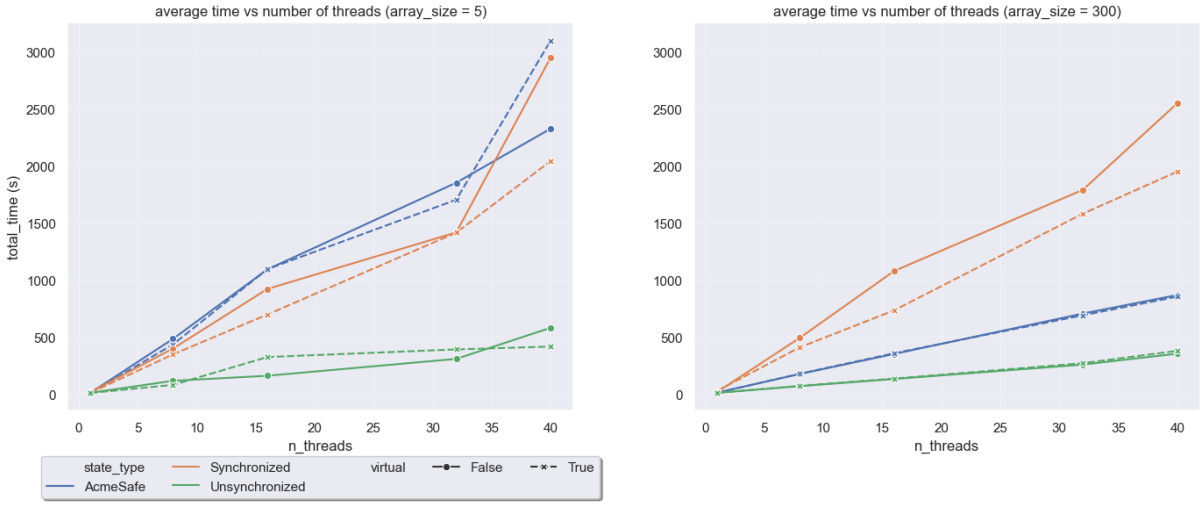

Figure 2: Average time per swap vs. number of threads, with an array size of 300.

2.2 Correctness

Figure 3 displays the array sum results for arrays of size 5 and 300. As expected, the AcmeSafe and the synchronized approaches result in an array sum of 0, reflecting correctness (on this particular task; the atomic approach could fail to be correct on more complex tasks). By contrast, the unsynchronized approach yields incorrect results. As expected, the results are far worse in the short array case, since race conditions, and hence incorrectness, are much more likely on a small array.

3 Summary

The best approach will depend on the relative importance of performance and correctness, as well as on the nature of the data being processed. In almost every case, virtual threads provide a performance advantage over platform threads. While the data is not very conclusive, it seems like they do not incur a large correctness cost, as compared to platform threads (and we cannot think of a reason why they would). Thus, virtual threads should be preferred in every case.

If correctness is paramount and the arrays are small, then then synchronized approach is best, as it offers better performance than the atomic approach on small arrays. The reverse is the case for large arrays.

If performance is paramount, then the unsynchronized approach is best, although it suffers large correctness issues on small arrays.

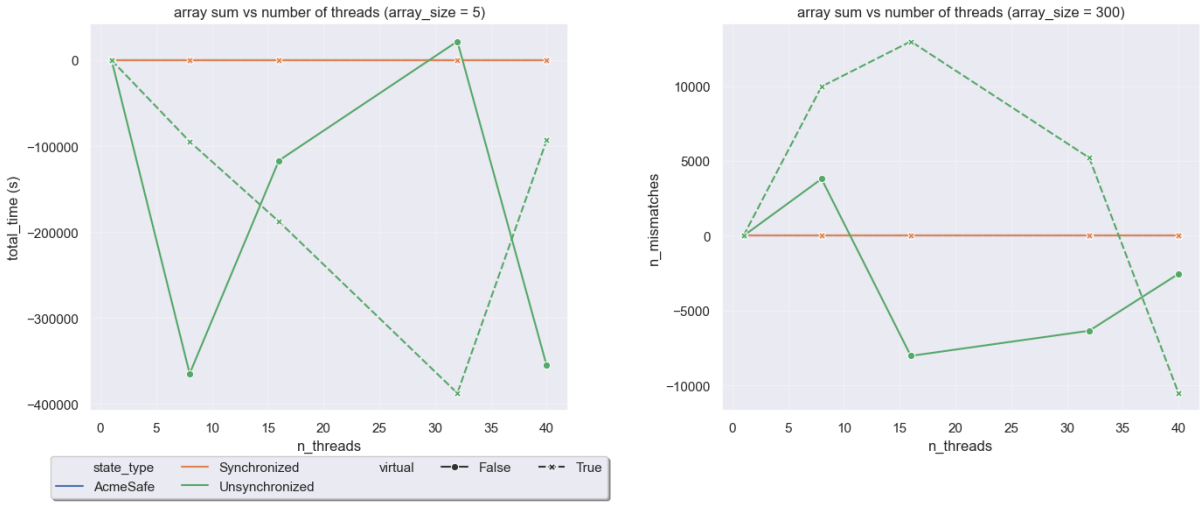

Figure 3: Array sum vs. number of threads, with arrays of size 5 and 300.