Multicore Semantics and Programming

Practical Report

A. J. Taylor (*at736*), St John's College

Abstract

A written report for Tim Harris' section of the course

Fix figure labels

1 Summary of Experimental Conditions and Methods

1.1 Hardware

Experiments were carried out on a HP Spectre Laptop, which was plugged in and on maximum performance settings. The laptop has a quad core, hyperthreaded, intel i7 8550u processor for a total of 8 physical threads (two threads per core) [1].

1.2 Experimental Methods

Experiments were written in Java and run under Windows 10. The laptop was set not to sleep for the duration of each experiment and other user processes were kept to a minimum to improve reliability of the results. The construction of all objects used by the performance tests is kept in the setup code and is not timed. Furthermore, GC runs are forced between performance tests and no individual test is expected to produce enough objects to make a GC call mid-test-run.

1.3 Code Written

The code used to run experiments and process the resulting data can be found on a dedicated Github repository[2]. I created an abstract `SharedArray` class containing an array with specifiable length and an abstract `sum` (read) and `update` (write) operations. Appropriate subclasses of this class were created with the `sum` and `update` operations taking the correct locks.

2 The Experiments

2.1 Set Up and Initial Test

The supplied test code ran correctly.

2.2 Simple Multithreading

In this experiment, (Fig 2.2), repeated 100 times per number of threads, performance stays roughly equal for $n = 1, 2$, then begins to increase monotonically. As might be expected, there is a larger

[1] https://ark.intel.com/products/122589/Intel-Core-i7-8550U-Processor-8M-Cache-up-to-4-00-GHz-

[2] https://github.com/Al153/MulticoreSemantics

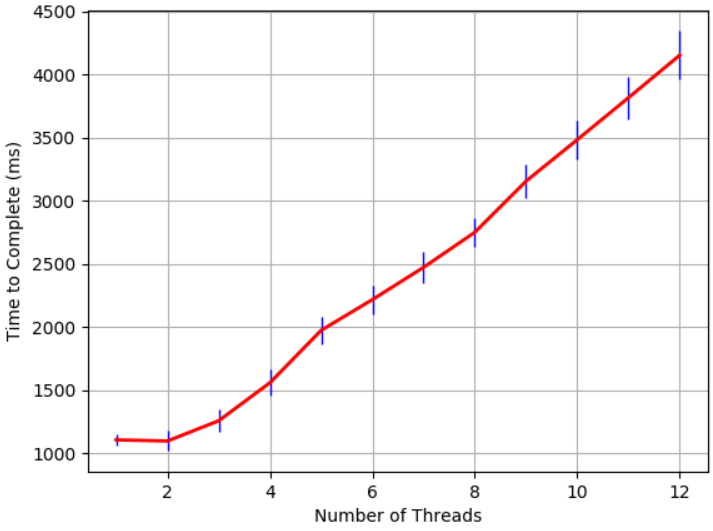

Figure 1: Time to complete for each thread running. Error bars, as is the case in the rest of this report, represent a single standard deviation on either side.

increase between $n = 4k$ and $n = 4k + 1$ than in other intervals of n. This occurs because $4k$ operations can be scheduled within k time units, whereas in the best case, $4k + 1$ requires $k + 1$ time units. In an ideal system, we might see the line be flat for $n = 4k + 14k + 2, 4k + 3, 4(k + 1)$, as each core would be utilised fully. We also see a deterioration in performance between one thread per core (1, 2, 3, 4) cores and two threads per core (5, 6, 7, 8) as the delay operation between threads are identical, tight, loops with little unpredictable memory access, meaning that simultaneous multithreading (hyperthreading) is unable to find many redundant pipeline stages to exploit.

2.3 Read Only Shared-Arrays

Unsafe Shared Array The locked array (Fig 2.3) did not perform significantly worse in the average case than the unsafe array (Fig 2.3), however the variance increases significantly as the number of threads and size of the array increases. I expect this is due to a one sided distribution, where the majority of results are scattered close to the mean with some large outliers which took much longer due to a delay in loading into local cache and taking a lock. The difference in standard deviation and means for the largest array size is shown in Fig 2.3. The fact that the mean time to complete does not increase substantially with increasing thread count suggests that the bottleneck for each loop in the common case is not taking the lock or running the sum operation but instead due to overheads like loading the array to cache to be summed and checking exit conditions for the loop.

2.4 TATAS Lock

The TaTaS-lock performed slightly better than the mutex-lock for small numbers of threads due to its better cache line behaviour, but as the number increased, the performance of the TaTaS lock deteriorated and the standard deviation of its operation increased greatly (Fig 2.4). This is potentially due to the overhead of having to do an extra test before test and setting, slightly delaying the operation

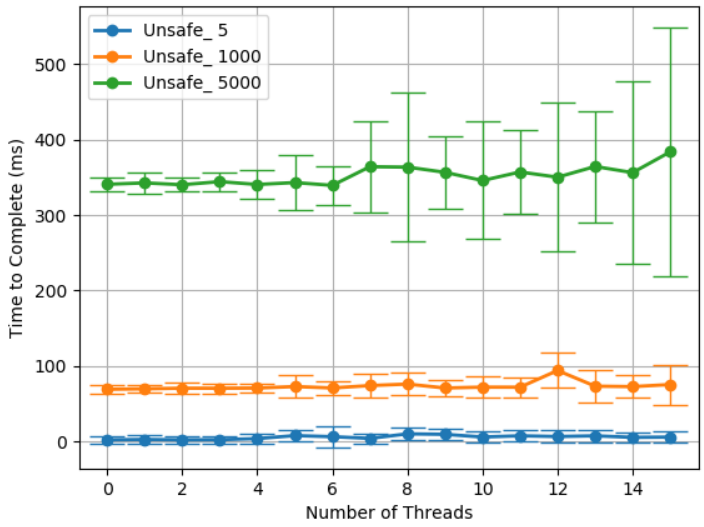

Figure 2: Speed of the unsafe shared array when instantiated with various sizes

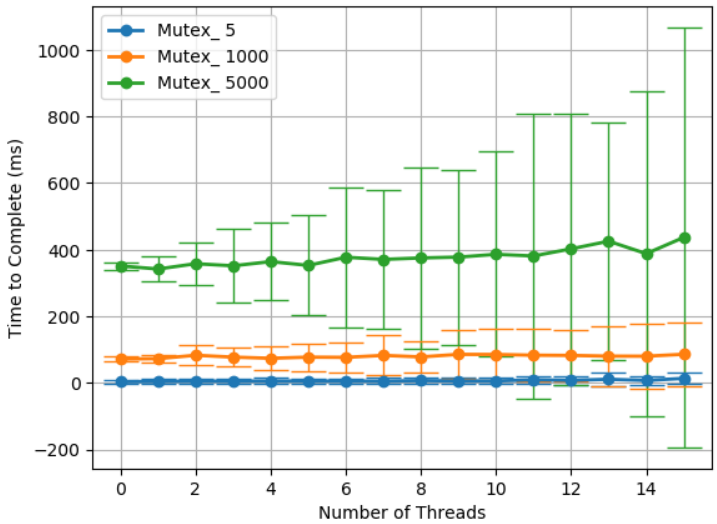

Figure 3: Speed of the safe, mutex locked, shared array when instantiated with various sizes

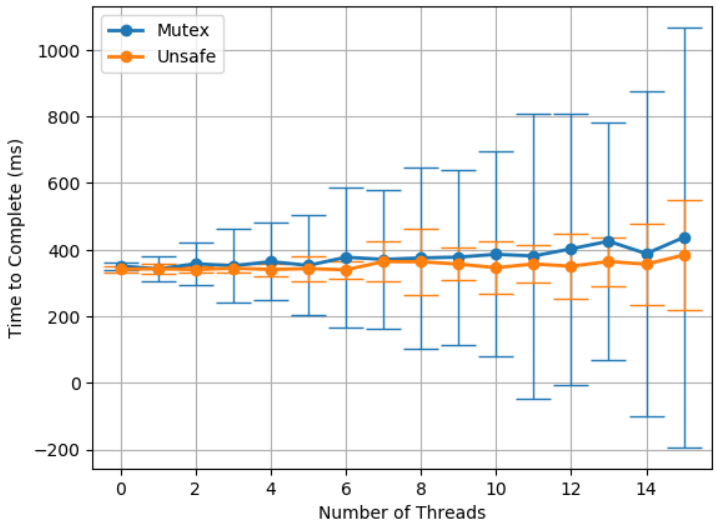

Figure 4: Comparative speed of the unsafe and mutex-locked arrays set to a size of $X = 5000$. The standard deviation is much more constant for the unsafe version.

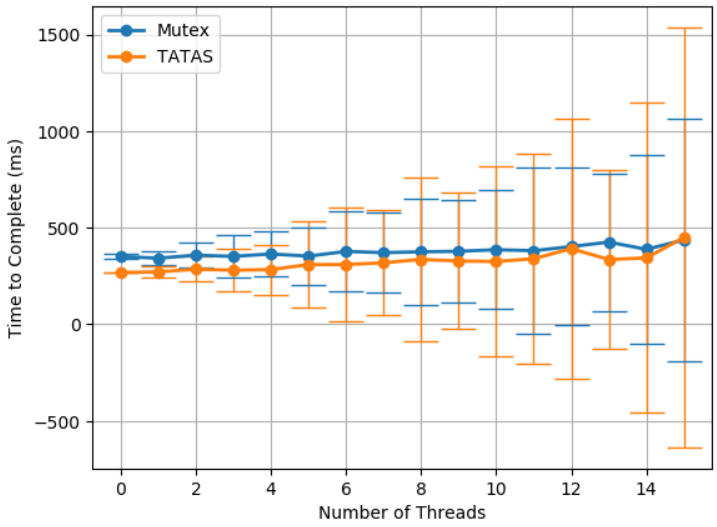

Figure 5: Speed of the mutex-locked and TaTaS shared arrays for $X = 5000$

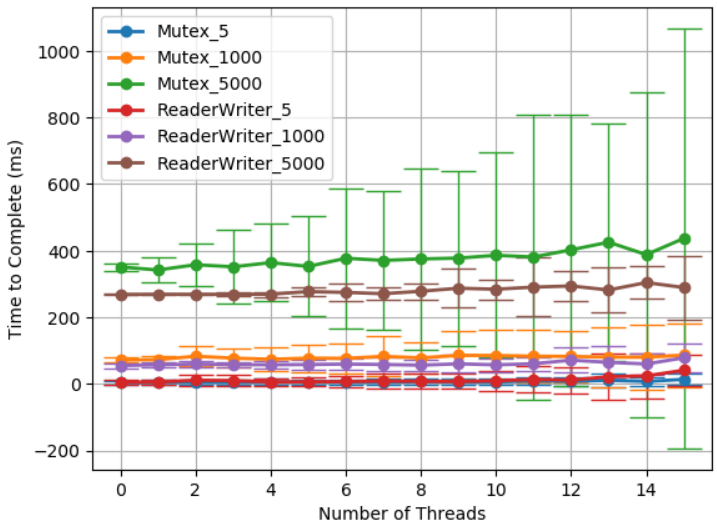

Figure 6: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes

of the code. Another potential culprit is the larger number of memory bus and cache transactions per operation.

better explanation

2.5 Reader-Writer Lock

When the array size is not negligibly small, the reader-writer lock significantly outperforms the standard mutex lock as it allows multiple summing operations to occur at once. It also has a much smaller standard deviation for each operation than seen in mutex case (Figs 2.5, 2.5). This may be due to the much shorter array access delay time due to not having to wait for a lock on the array in order to read the array. As a result, fewer coincidental pathological cases arose where the threads blocked on each other.

2.6 Flag-Based Lock

The flag based lock outperformed the mutex-lock less significantly than the reader-writer lock but with an even tighter standard deviation due to the cache lines containing flags not

2.7 Write Mode

For the write-mode experiments, I fixed the array size to be $X = 1000$. Further more, since all experiments up to this point had been run to make 100,000 calls to the `sum` function in the main thread before halting, running a writing`update` operation every 100k iterations would have resulted in barely any write operations. Hence I decided to deviate from the specification and run `updates` every 10,000 iterations in the sparse update case. In the frequent update case, I kept $K = 100$ as specified.

This is now wrong Other than some anomalous cases, all of the more advanced locks outperformed

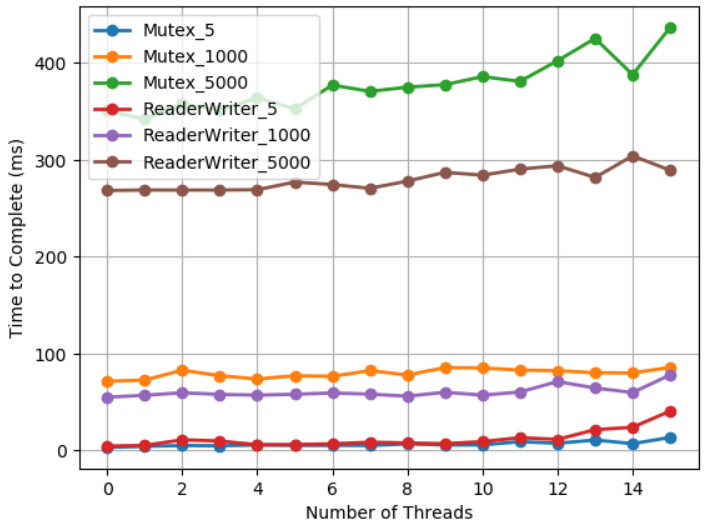

Figure 7: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

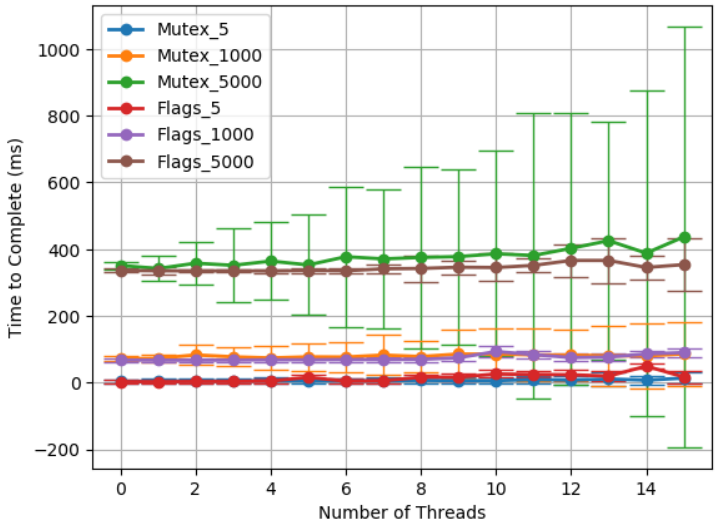

Figure 8: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes

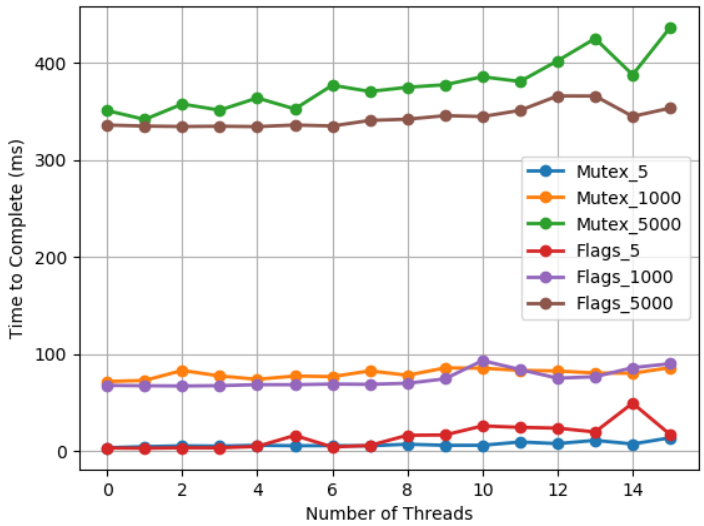

Figure 9: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

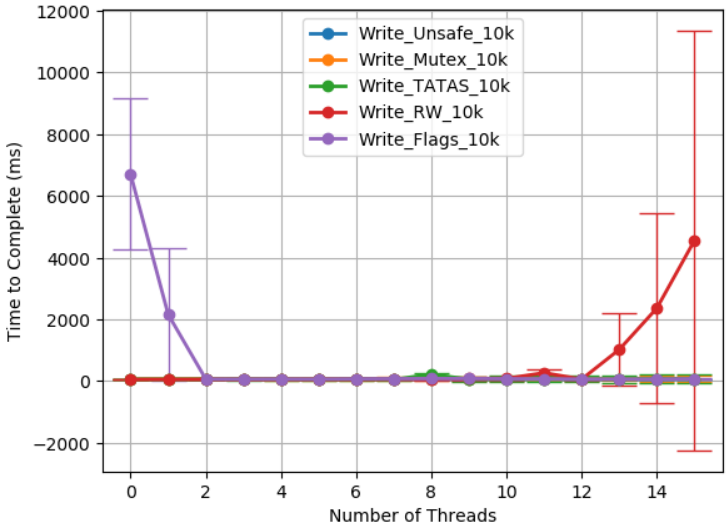

Figure 10: Performance of each type of lock when used for update operations every 10,000 operations.

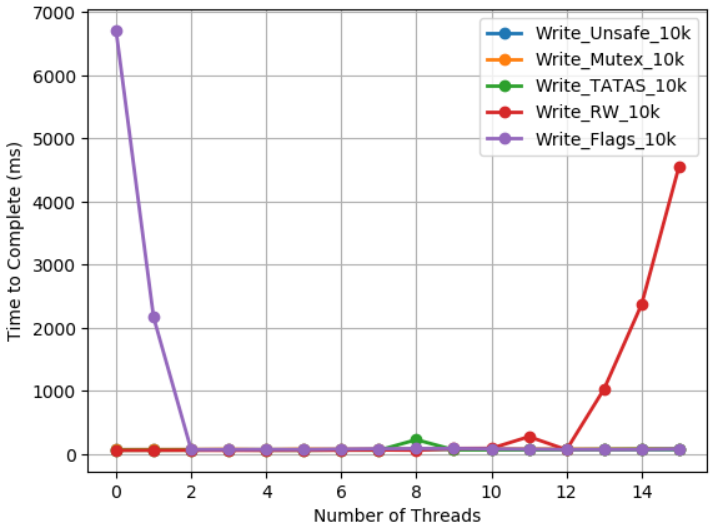

Figure 11: Performance of each type of lock when used for update operations every 10,000 operations, drawn without error bars to more clearly show the mean-behaviour.

the naive mutex lock in the sparse update case (Figs 2.7, 2.7). The flags based lock appeared to be the best performer in the higher thread count cases, though the variance of results outweighed the difference in means.

More interesting trends emerge when one looks at higher frequency write operations. In the $K = 100$ case (Figs 2.7, 2.7), it can clearly be seen that the naive mutex case performs as poorly as the other worst other locks. However, the flags based locks and TaTaS locks performed well across all thread counts.

3 Summary

One issue with these experiments has been the non-reproducibility of the tests. Despite the high number of repeats (200 tests per batch) and length of tests in terms of operations (100k operations per test), and reviews of my experimental methods, I have seen poor reproducibility. Another anomaly is the general lack of performance degradation as the number of threads increased. In each test case, we would expect the amount of time to complete to increase by around a factor of two between 8 threads and 16, however this has not occurred. This suggests there is an additional overhead in the tests brought about by the operation of the JVM. Perhaps using a lower level language such as C or C++ with compiler optimisations turned off might cause trends to be more obvious.

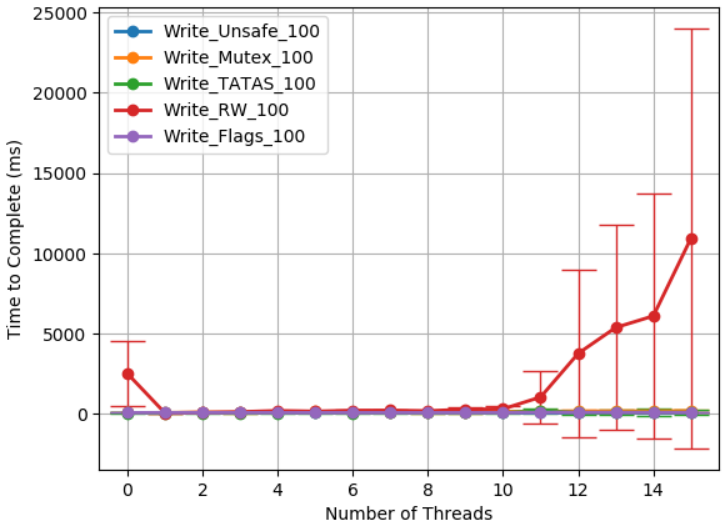

Figure 12: Performance of each type of lock when used for update operations every 100 operations

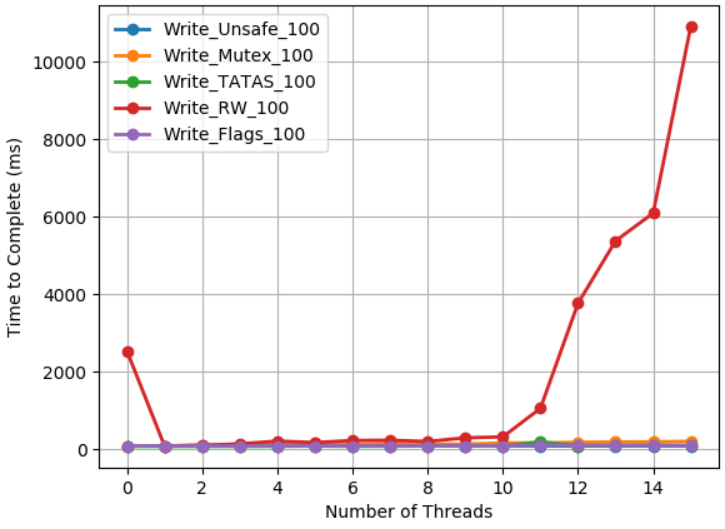

Figure 13: Performance of each type of lock when used for update operations every 100 operations, drawn without error bars to more clearly show the mean-behaviour.