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

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 1), 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 increase between $n = 4k$ and $n = 4k + 1$ than in other intervals of n. This occurs because $4k$

[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

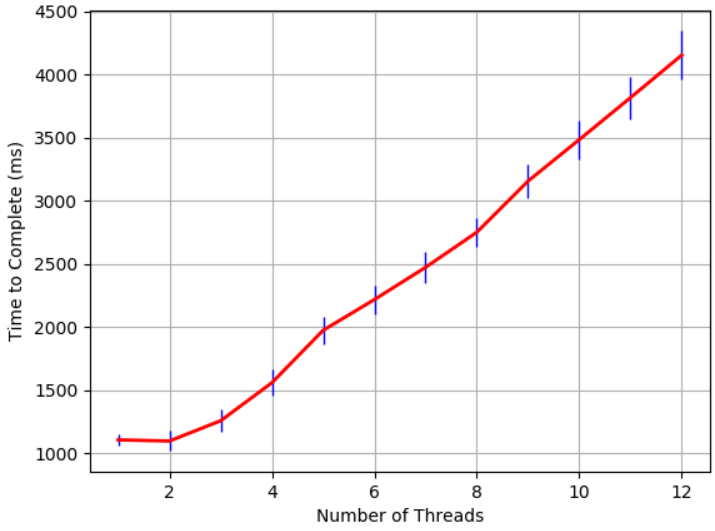

Figure 1: Time to complete for each thread running. Error bars, as is the case in the rest of this report, represent a single standard deviation on either side.

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 ($n = 1, 2, 3, 4$) cores and two threads per core ($n = 5, 6, 7, 8$). This may be cause by there being little pipeline stage redundancy in the code that was executed in each thread. Each thread's `delay` operation is an identical, tight loop with limited and predictable memory access. This means that there are few idle, usable pipeline stages for the processor to exploit. As a result, multiple threads can not be run simultaneously at full speed on a single core.

2.3 Read Only Shared-Arrays

Naive Mutex Locking The locked array (Fig 3) did not perform significantly worse in the average case than the unsafe array (Fig 2), however the variance increases significantly as the number of threads and size of the array increases. I expect this is due to the existence of a one sided distribution. The majority of results are scattered close to the mean, however a small proportion of results were delayed significantly longer due to particularly long waits to acquire locks or worst case cache transactions. The difference in standard deviation and means for the largest array size is shown in Fig 4. 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 , but as the number increased, the performance of the TaTaS lock deteriorated and the standard deviation of

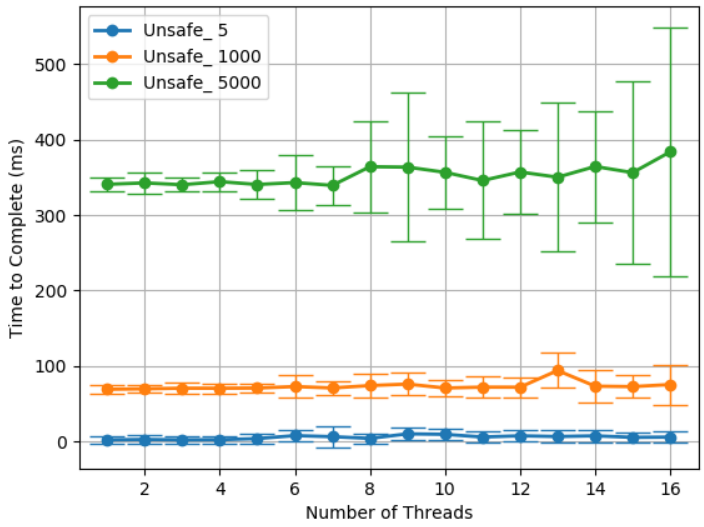

Figure 2: Speed of the unsafe shared array when instantiated with various sizes

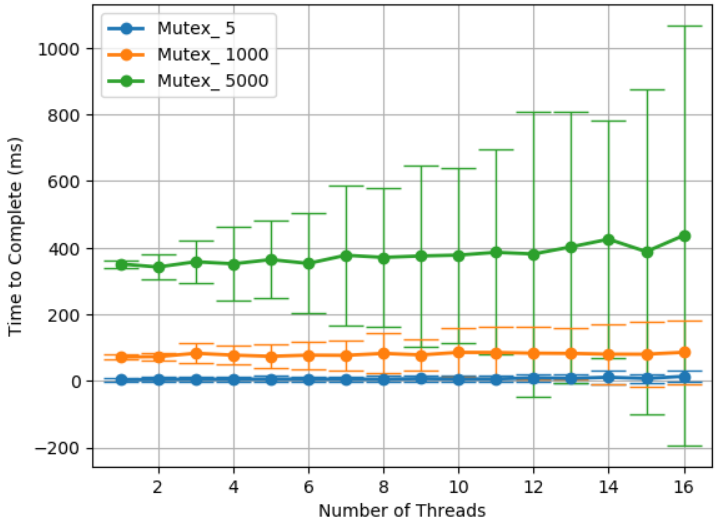

Figure 3: Speed of the safe, mutex locked, shared array when instantiated with various sizes

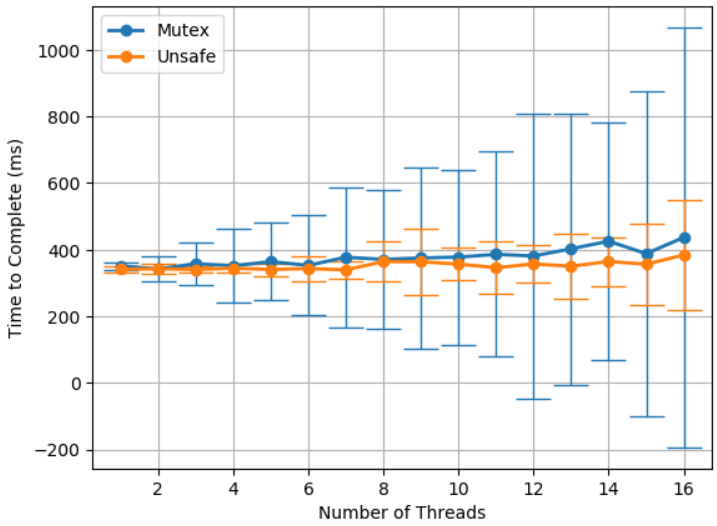

Figure 4: Comparative speed of the unsafe and mutex-locked arrays set to a size of $X = 5000$. The standard deviation is much more constant for the unsafe version.

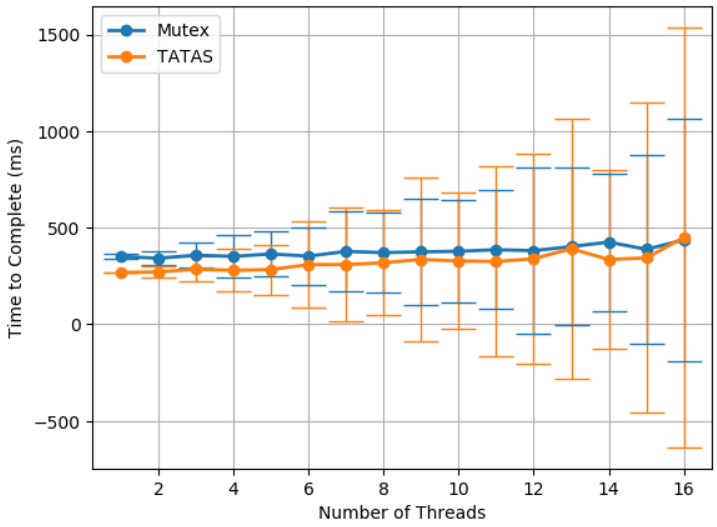

Figure 5: Speed of the mutex-locked and TaTaS shared arrays for $X = 5000$

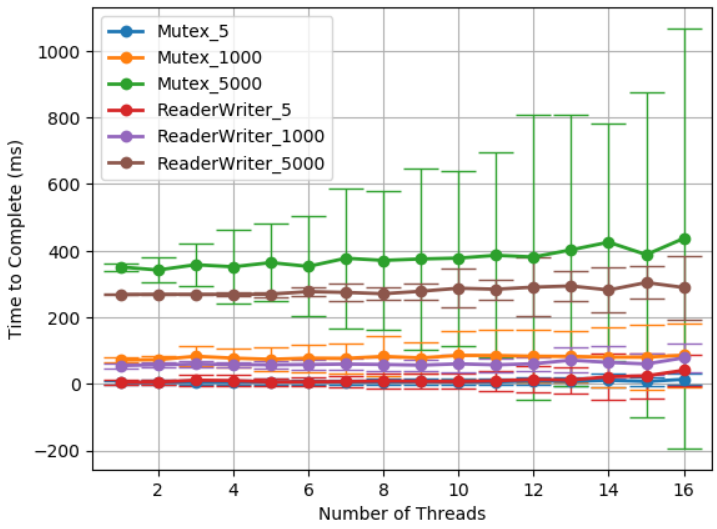

Figure 6: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes

its operation increased greatly (Fig 5). An explanation might be that the TaTaS's forecasted better cache-line behaviour works when relatively few threads are trying to take the mutual exclusion lock, but as the number of threads increases, so does the contention on the cache-coherence protocols, since test-and-test-and-set requires more cache coherency transactions than test-and-set. Furthermore, since the naive mutex lock simply uses Java's built-in `synchronized` primitive while the TaTaS was built using an `AtomicBoolean` and compare-and-set. The primitive must be able to lock over the whole object and hence may have some overhead in low-contention cases, but by being built-in, it may be able to take advantage of more low level JVM optimisations in higher contention cases. The high-level TaTaS construction is built with more general constructs, making it harder for the JVM to optimise away.

2.5 Reader-Writer Lock

When the array size is not negligibly small, the reader-writer lock significantly outperforms the standard mutex lock. It also has a much smaller standard deviation for each operation than seen in mutex case (Figs 6, 7). Firstly, the reader-writer lock allows as many writers as possible to access the array. Secondly, the maximum wait time for a thread to access the array would be much shorter due to fewer pathological cases where lots of threads are blocked on each other. This means that the distribution of run-times would be much smaller.

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. This may be due to the fact that the reader flags occupy separate cache lines, so there is even less delay in taking a read lock, as there is single memory location for there to be contention over (a single counter in the reader-writer case). This means that no threads have to compete for exclusive cache-line ownership of the lock, so the locking-time is improved.

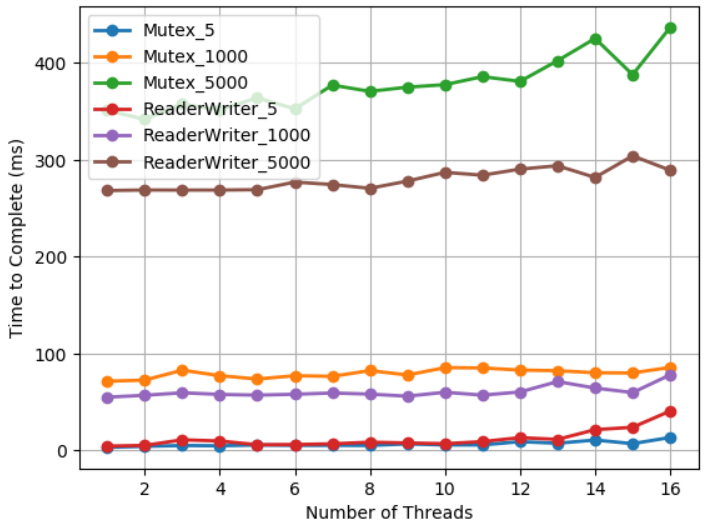

Figure 7: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

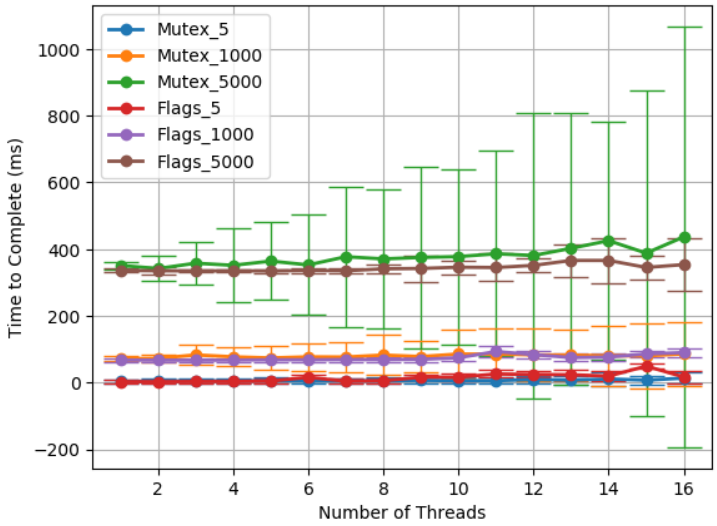

Figure 8: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes

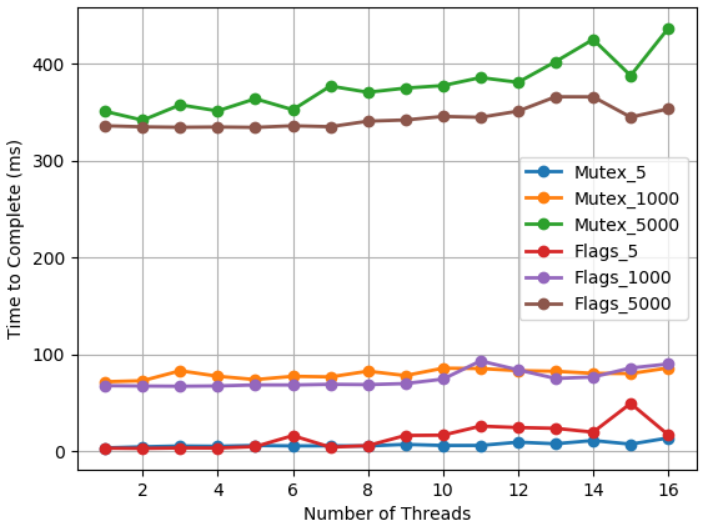

Figure 9: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

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. Despite iterating several times over reformulating the tests to remove experimental error, experimental results were still somewhat unstable from test run to test run. One factor appeared to be individual tests being greatly slowed due to environment processes, leading to anomalous spikes. The write tests appeared to be much more numerically unstable than the read-only tests.

In the sparse update case ($k = 10,000$, Figs 10, 11), the best performer, as might be expected, was the flag based lock. This provided the least contention when switching between read and write locking. All five locking techniques performed well when the number of program threads was less than the number of hardware threads. Around and above 8 threads, performance deteriorated for most of the schemes. The unsafe case slowed to a very consistent speed over 8 threads. This may have been due to poor cache behaviour where threads were allowed to modify cache lines that were shared between other processors' caches. The basic mutex slowed more greatly and less predictably, likely due to its overzealous locking of read threads. The fact that we see a more obvious progression of runs getting slower as the number of threads increases compared with the read-only tests may be due to the JVM or bare metal processor itself enforcing a total store order on the write operations to the same address. This means that fewer runtime optimisations such as memory instruction reordering available, making performance more predictable. The reader-writer lock performed poorly and inconsistently. This may be due to a high level of contention on the atomic integer that represents the lock. This lock is continuously atomically `Compare-And-Set`ed by multiple threads at once. The TaTaS lock performed well due to its better cache behaviour over the mutex. This presents questions as one would expect it to have similar cache behaviour to the reader-writer lock.

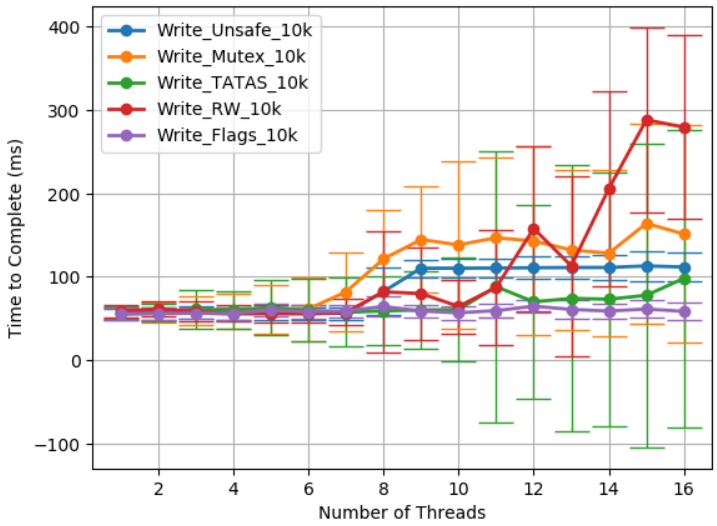

Figure 10: Performance of each type of lock when used for update operations every 10,000 operations.

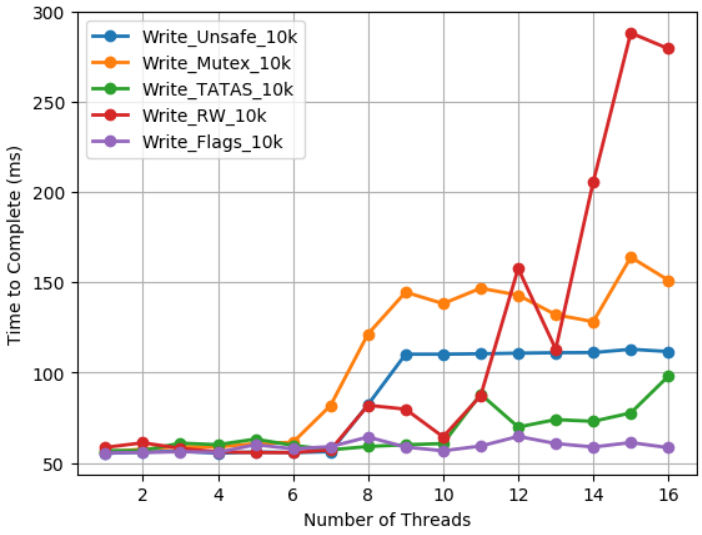

Figure 11: Performance of each type of lock when used for update operations every 10,000 operations, drawn without error bars to more clearly show the mean-behaviour.

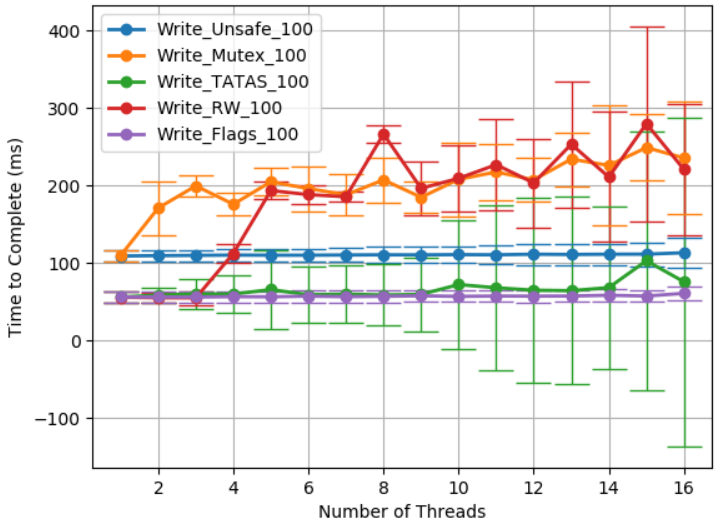

Figure 12: Performance of each type of lock when used for update operations every 100 operations

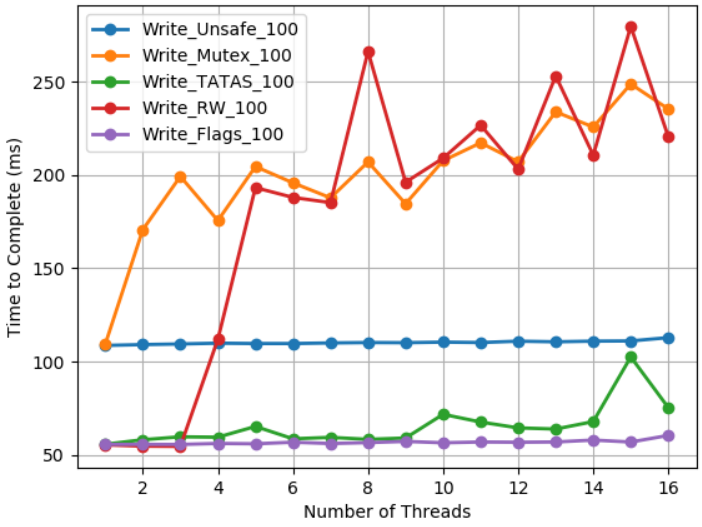

Figure 13: Performance of each type of lock when used for update operations every 100 operations, drawn without error bars to more clearly show the mean-behaviour.

These trends continue for the more frequent update case. The Reader-Writer and simple mutex locks continued to be the least performant, showing poor run-times from a lower thread count in this case. The unsecured case continued to provide a consistent level of performance that was worse than the best locked cases, and the TaTaS and Flags based locks continued to run fastest, with the Flag-Lock also having a very low variance.

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, most obviously in running the `update` operation tests. 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. Further more, since Java is a higher level language and there a large number of techniques used to make the JVM run fast, such as JIT compilation, performance can be hard to predict. Perhaps using a lower level language such as C or C++ with compiler optimisations turned off might cause trends to be more obvious.