Multicore Semantics and Programming

Pratical 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.

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 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 $k + 1$

[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

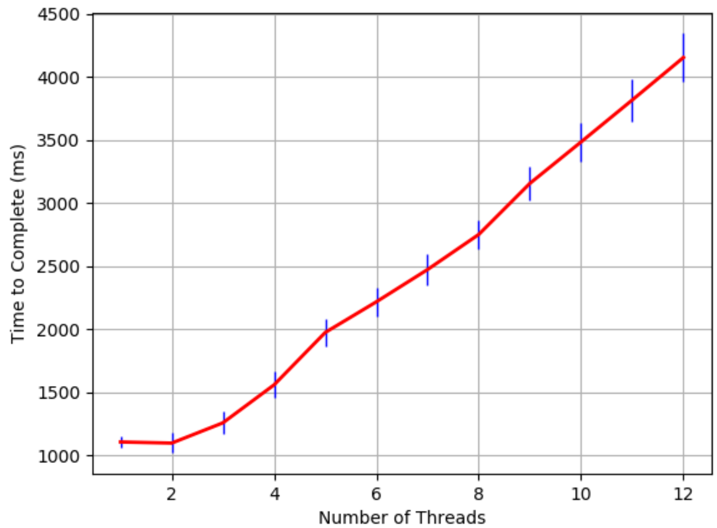

Figure 1: Time to complete for each thread running. Error bars, as is the case in the rest of this report, represent a single standard deviation on either side.

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.

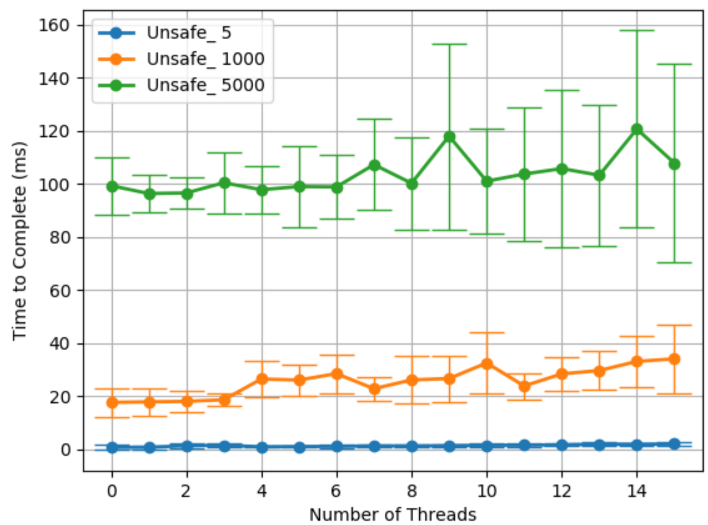

Figure 2: Speed of the unsafe shared array when instantiated with various sizes

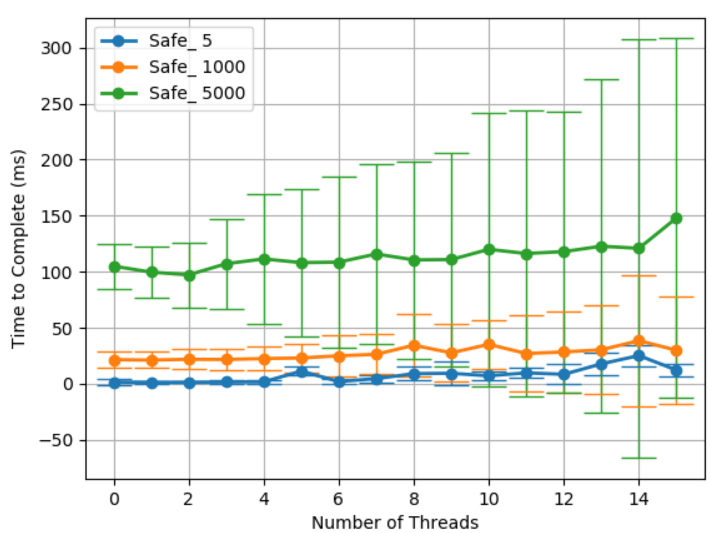

Figure 3: Speed of the safe, mutex locked, shared array when instantiated with various sizes

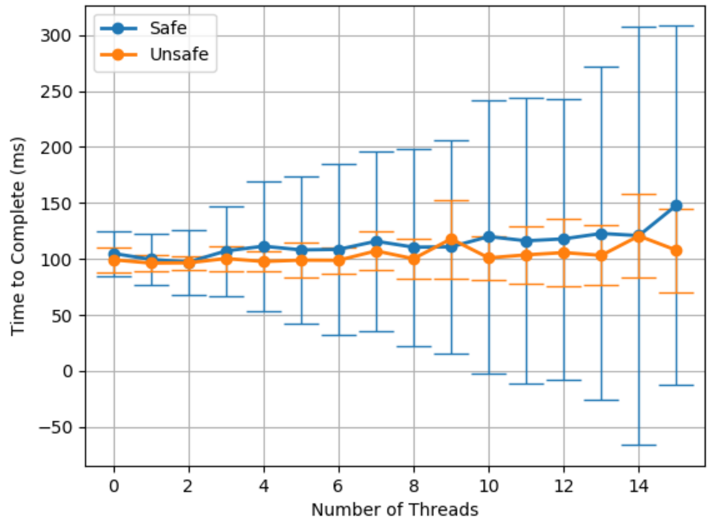

Figure 4: Comparative speed of the unsafe and mutex-locked arrays set to a size of $X = 5000$. The standard deviation is much more constant for the unsafe version.

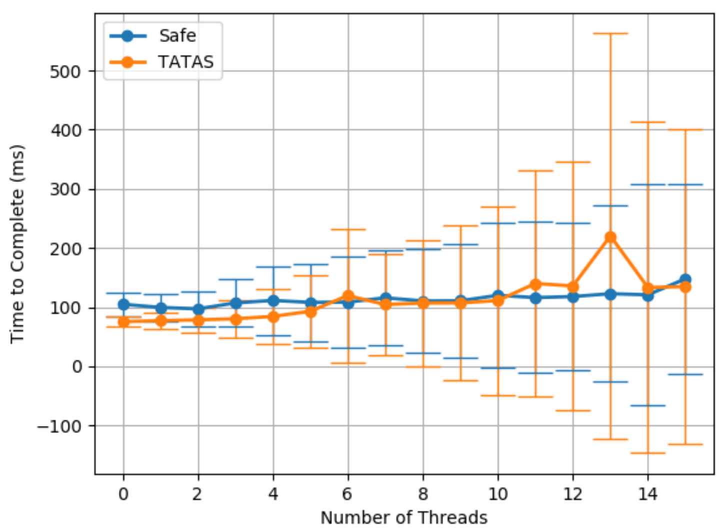

Figure 5: Speed of the mutex-locked and TaTaS shared arrays for $X = 5000$

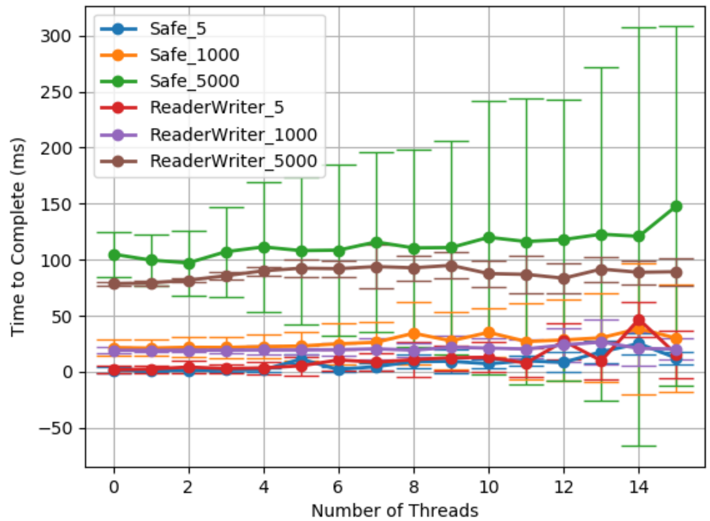

Figure 6: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes

2.5 Reader-Writer Lock

2.6 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.

2.7 Flag-Based Lock

2.8 Reader-Writer Lock

The flag based lock outperformed the mutex-lock less significantly than the reader-writer lock but with a tighter standard deviation. **clearer**

2.9 Reader-Writer Lock

2.10 Write Mode

3 Summary

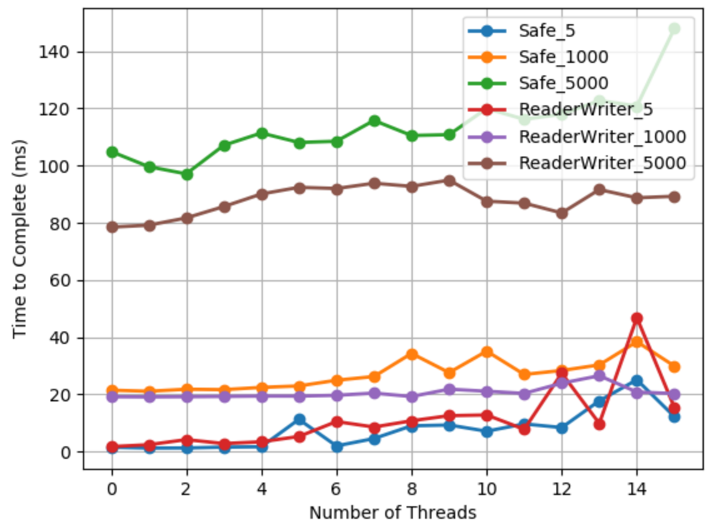

Figure 7: Speed of the mutex-locked array versus the reader-writer locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

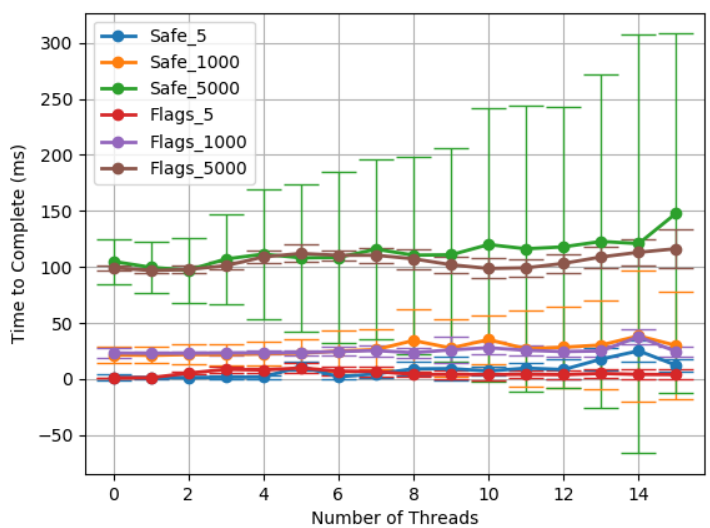

Figure 8: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes

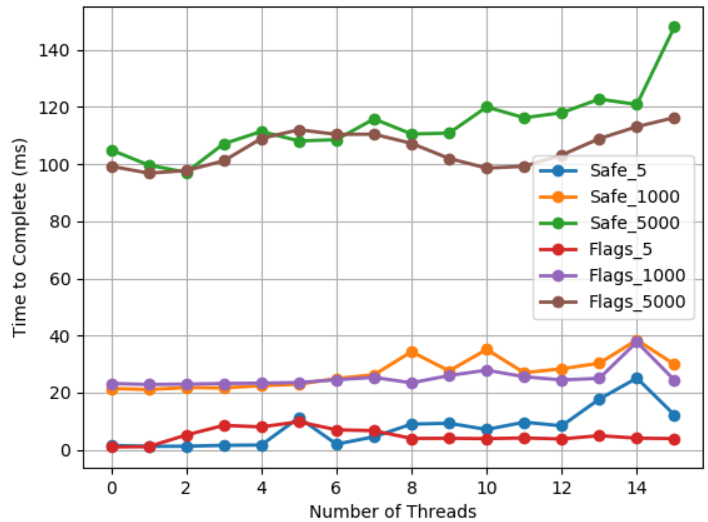

Figure 9: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

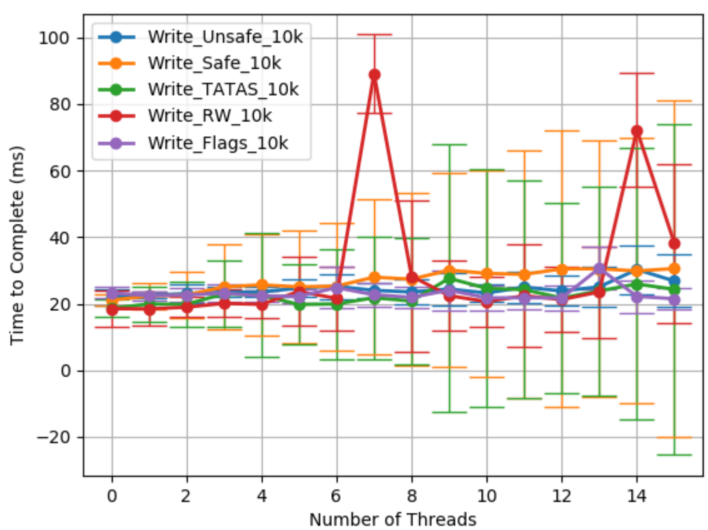

Figure 10: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

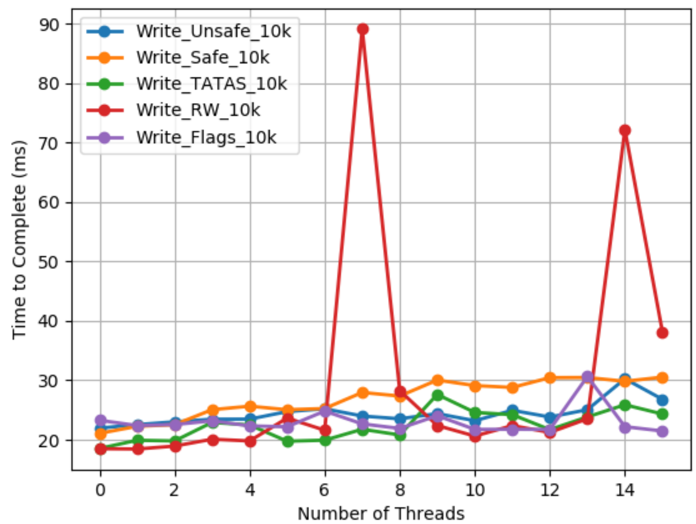

Figure 11: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

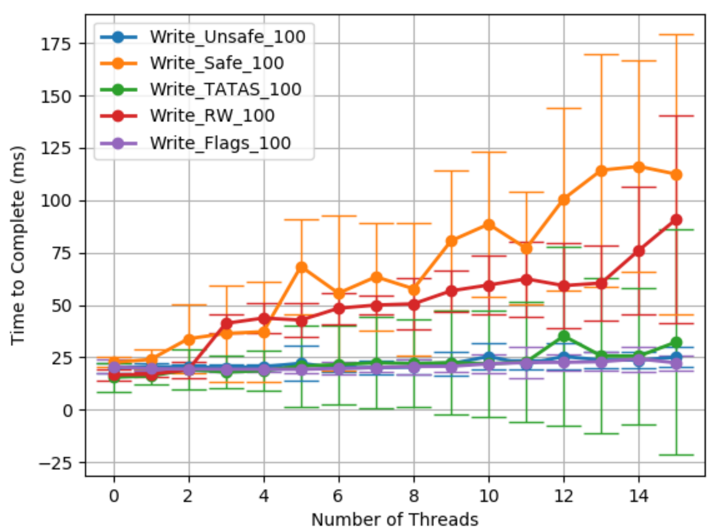

Figure 12: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.

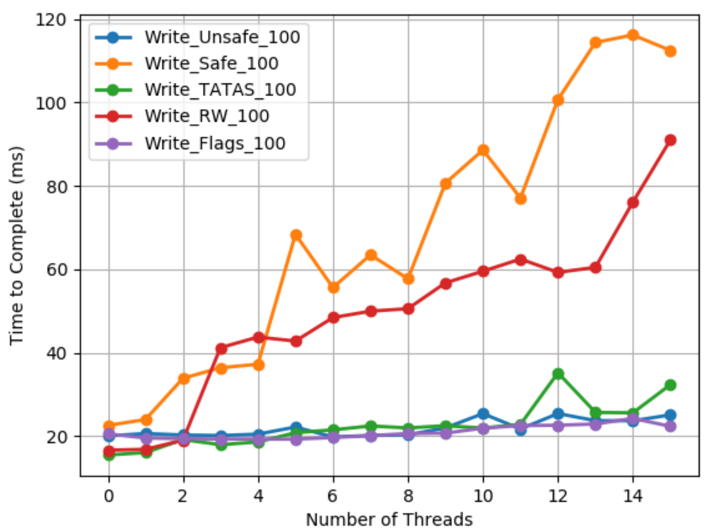

Figure 13: Speed of the mutex-locked array versus the flag-locked array when instantiated with various sizes, drawn without error bars to more clearly show the mean-behaviour.