CSC 458 Assignment 2

Nick Graham

February 20, 2017

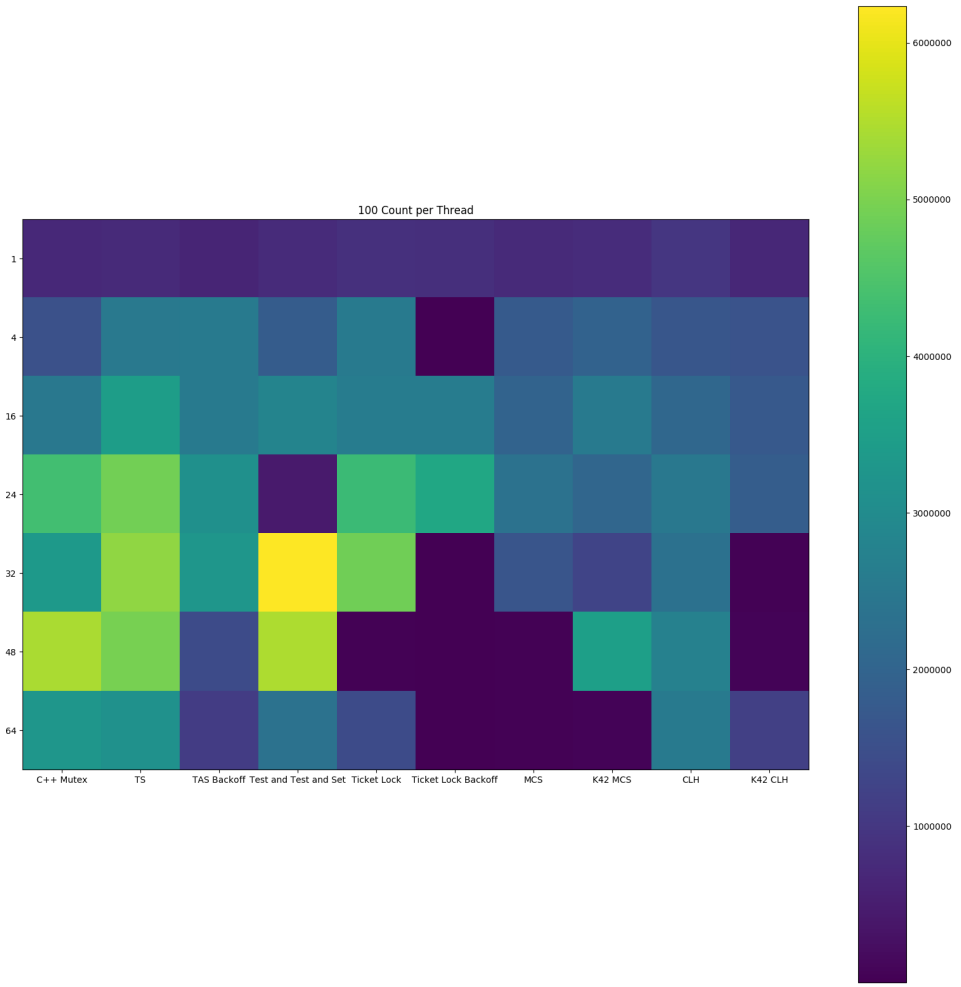
Introduction

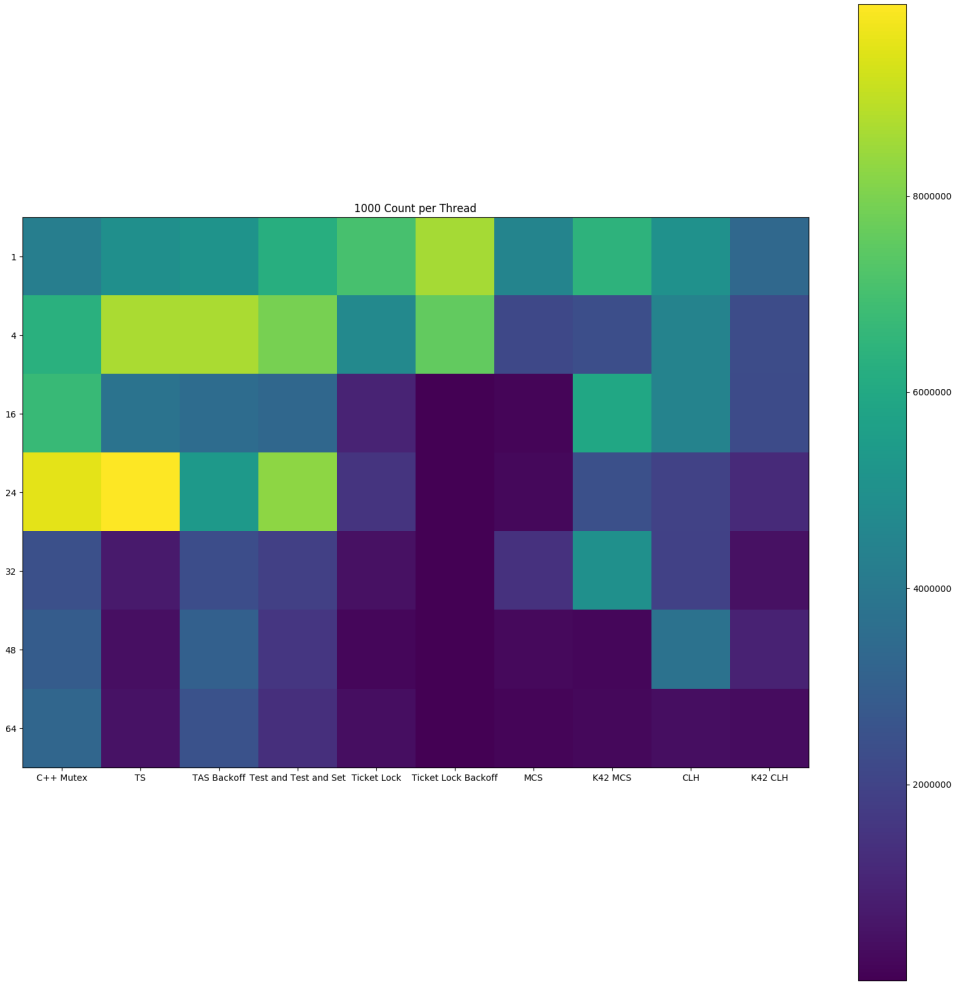
Assignment Two had us look at different implementations of locks and compare them to see which ones performed best at different numbers of threads under different loads. The locks we tested were the built C++11 mutex, a Test and Set lock, a Test and Set lock with backoff, a Test and Test and Set lock, a Ticket lock, a Ticket lock with backoff, a MCS lock, a K42 MCS lock, a CLH Lock, and the K42 CLH lock. A simple counter program was used again to test the locks, having multiple threads access the same counter. The program was tested on both the x86 machines as well as the IBM machines.

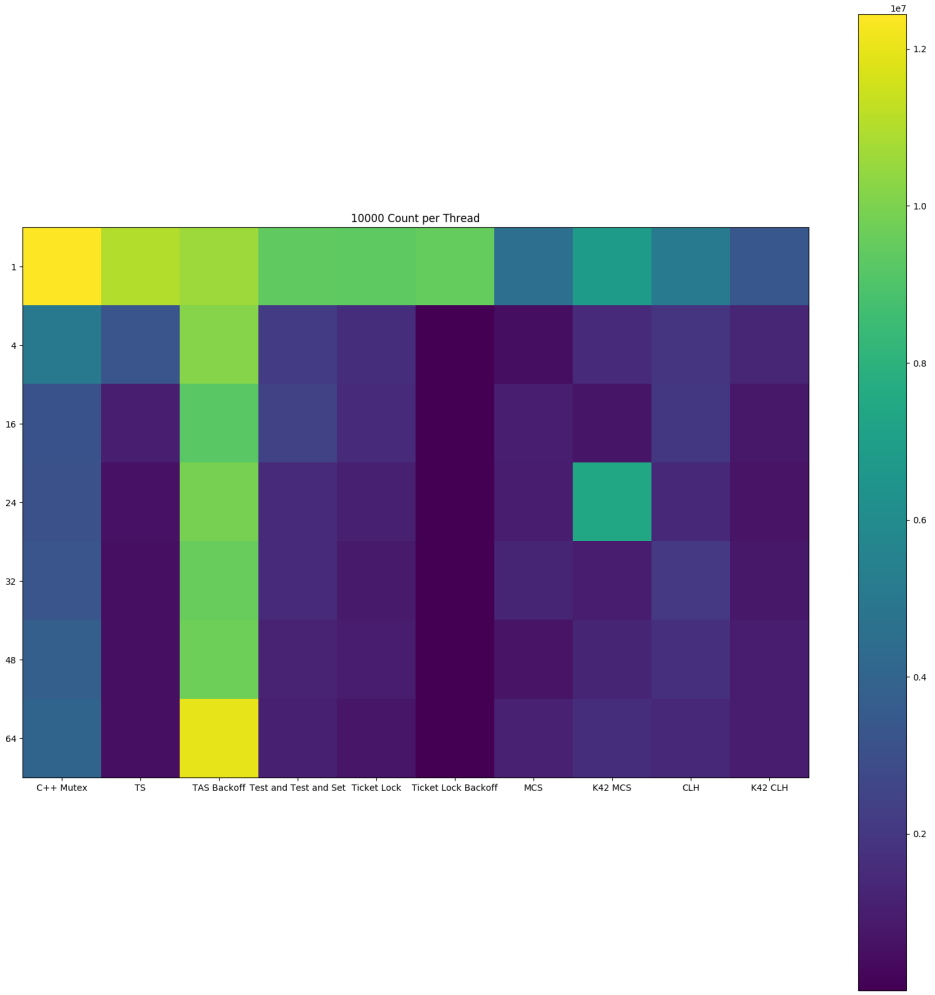
Data

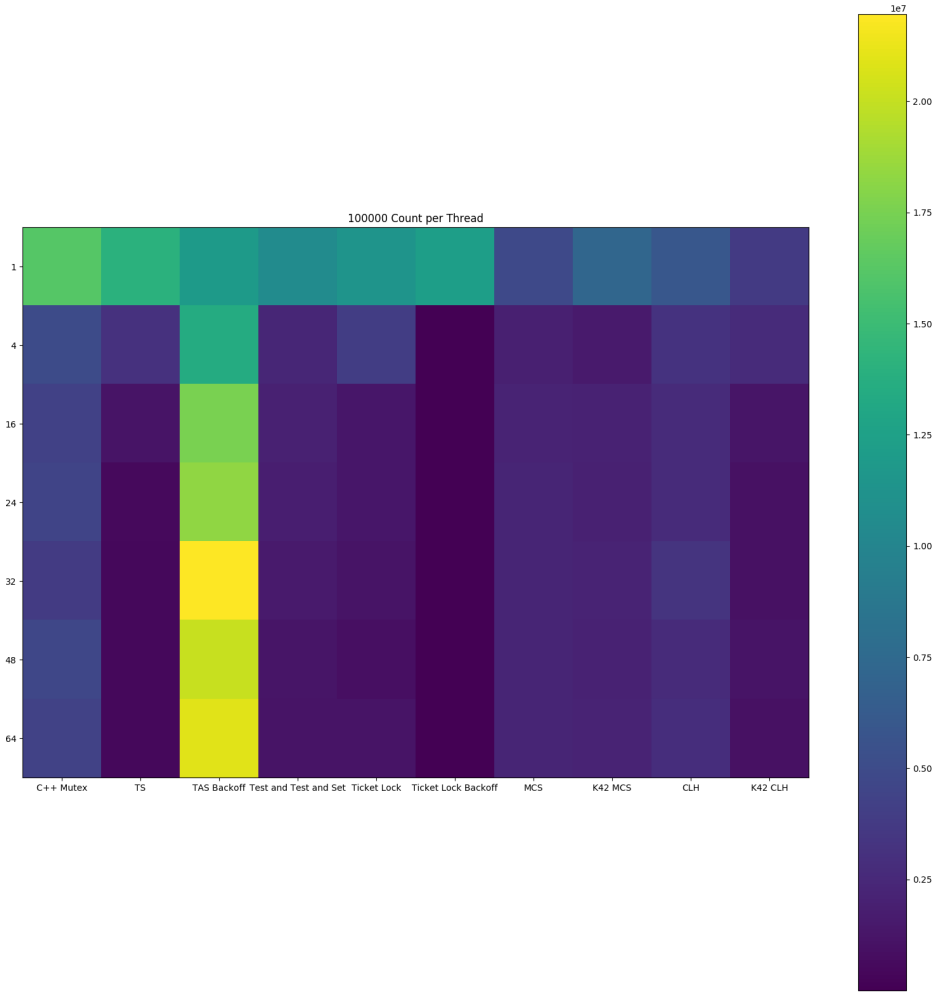
x86

These plots show the number of increments per second through the heatmap. In these plots the larger the number (yellow in color) the better the lock performed


100 Count per Thread
1
4
16
24
32
48
64
C++ Mutex
TS
TAS Backoff
Test and Test and Set
Ticket Lock
Ticket Lock Backoff
MCS
K42 MCS
CLH
K42 CLH
6000000
5000000
4000000
3000000
2000000
1000000


1000 Count per Thread
1
4
16
24
32
48
64
C++ Mutex
TS
TAS Backoff
Test and Test and Set
Ticket Lock
Ticket Lock Backoff
MCS
K42 MCS
CLH
K42 CLH
8000000
6000000
4000000
2000000

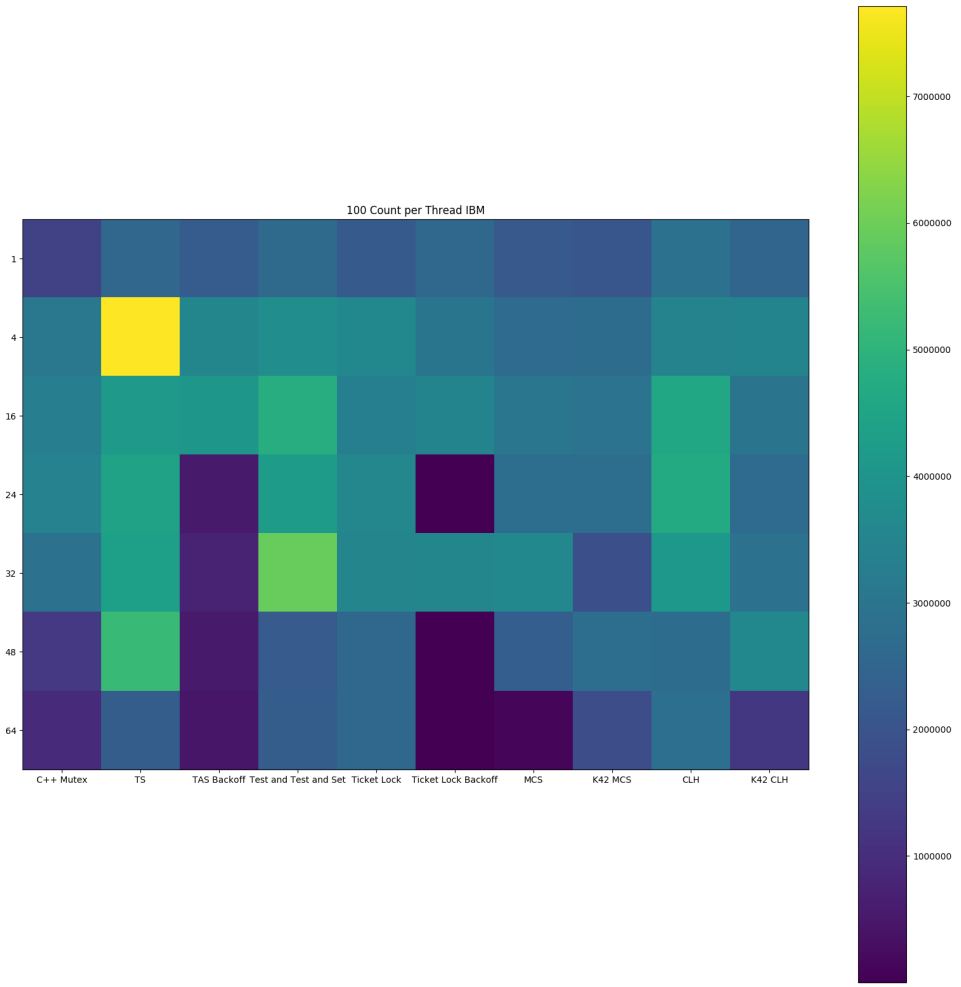

10000 Count per Thread
1
4
16
24
32
48
64
C++ Mutex
TS
TAS Backoff
Test and Test and Set
Ticket Lock
Ticket Lock Backoff
MCS
K42 MCS
CLH
K42 CLH
1e7
1.2
1.0
0.8
0.6
0.4
0.2

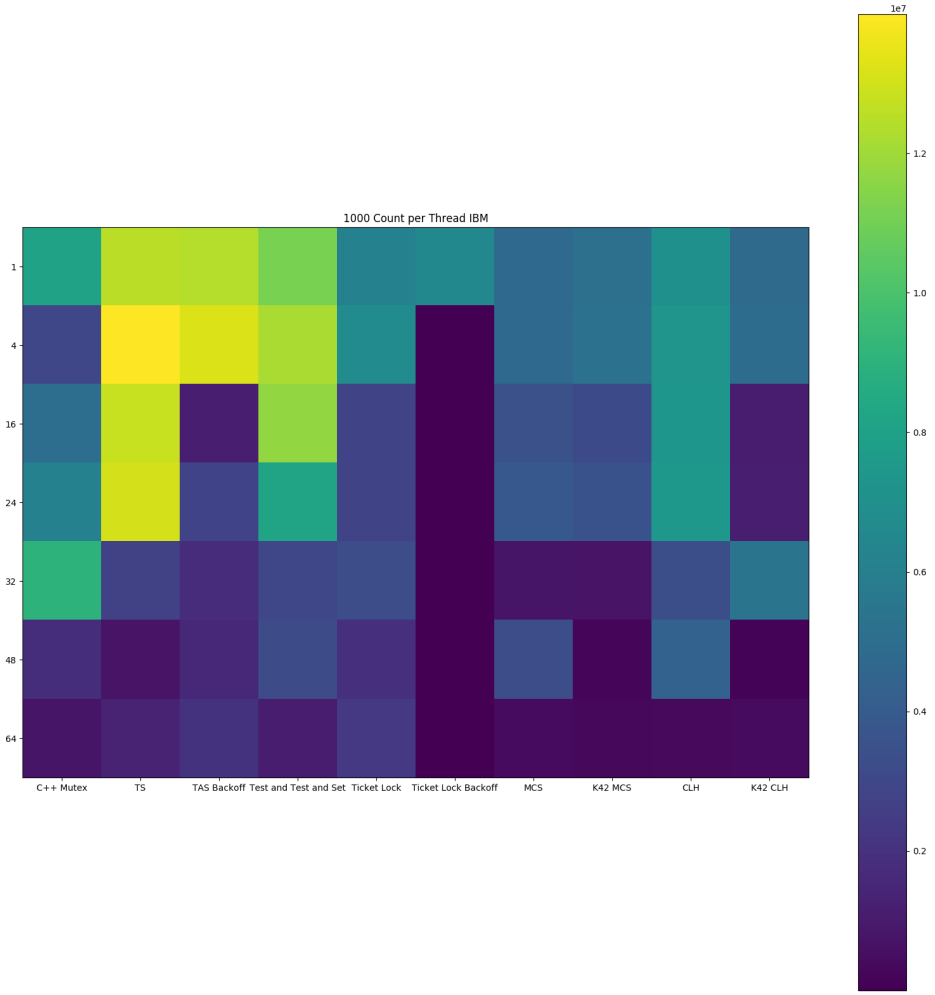


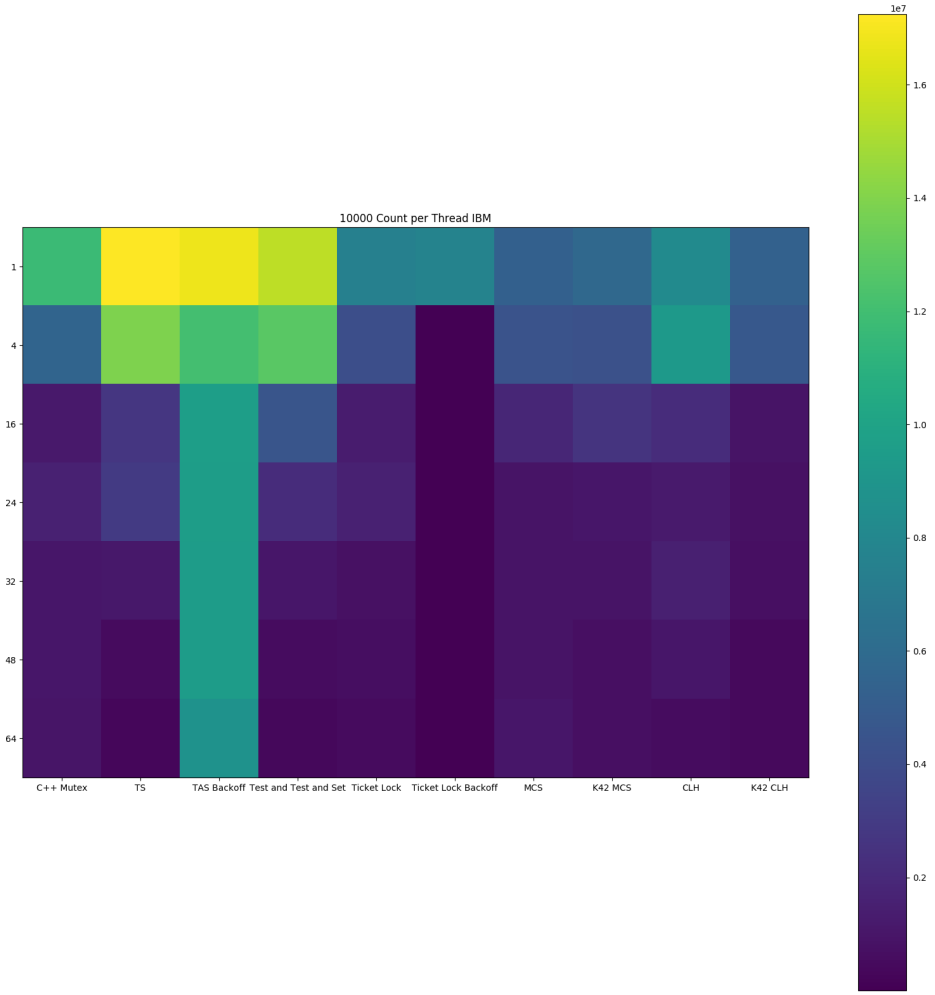
IBM

These plots show the number of increments per second through the heatmap. In these plots the larger the number (yellow in color) the better the lock

performed The 100000 increment test was not run on the IBM machine due to the large amount of time it took to run, and I did not want to monopolise the machine for any longer then I already had.




1000 Count per Thread IBM
1
4
16
24
32
48
64
C++ Mutex
TS
TAS Backoff
Test and Test and Set
Ticket Lock
Ticket Lock Backoff
MCS
K42 MCS
CLH
K42 CLH
1e7
1.2
1.0
0.8
0.6
0.4
0.2


10000 Count per Thread IBM
1
4
16
24
32
48
64
C++ Mutex
TS
TAS Backoff
Test and Test and Set
Ticket Lock
Ticket Lock Backoff
MCS
K42 MCS
CLH
K42 CLH
1e7
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2

1 Analysis

All methods returned the expected counter values of $numberOfThreads \times IncrementNumber$ which is as expected. This means that all of the locks are functioning as designed and only allowing one thread to hold the lock at a given time.

Overall on the x86 machine the Test and Set with Backoff had the best performace overall. At low increment numbers and lower thread numbers (between 4 and 24 threads) the C++11 Mutex, TAS, and TATAS did outperform the Test and Set with backoff, however not but much, and at large Increment counts, the Test and Set with Backoff was the best for any number of threads greater then 1, by a very large margin.

On the IBM Machine the performance overall was worse. For lower increment counts the C++11 Mutex, and all of the TAS locks were about equivelent, however much like on IBM as the increment count increased the Test and Set with Backoff pulled ahead as the best performing lock.

I believe that the IBM has worse performace due to its implementation on the atomic instructions. I believe in class it was discussed that a lot of the atomic instructions on IBM are implemented as Load Linked/Store Conditional which has a high chance of failing when there is a large amount of contention on the object.

1.1 Backoff Tuning

I do not think that my backoff tuning is the best it could be. For the Test And Set Lock I found that a starting point of 1 ms, with a max of 1000ms and a multiplier of 2 gave the best performance. For the Ticket Lock a base of 1 ms was the best I could do. In both cases the base was the lowest I could go for my sleeping function. I believe that if I had instead slept for shorter amounts of time, the locks would have been able to perform much better.

2 Conclusions

Overall my tests showed that the Test and Set with backoff was the most efficient algorithm for locking. I also found the IBM machine had worse performance overall. There are a few sources of error in my implemention mainly in my use of atomics and my sleep function. I do not always use the

lowest possible memory order in the locks, which will increase the time it takes to access and release the lock, hurting the overall performance, and my sleep function for the locks with backoff had a minimum amount of time that I believe was too high, which made them not perform as optimally as they could have.