David Wade

Concurrent Programming

Lab 2

Write-Up

For this lab, I built my own concurrency primitives using atomic variables and operations. The primitives included three locking algorithms and one barrier algorithm. The primitives were used to implement my Bucketsort from lab 1 as well as control the multithreaded incrementation of a counter. The three locking algorithms I created were a Test and Set lock (tas), Test and Test and Set Lock (ttas), and a Ticket Lock. The tas and ttas locks are both LIFO locks whereas the Ticket lock is FIFO. The tas lock uses an atomic "flag" variable and atomically checks if flag is false. If it is false, it then changes it to true, if not, it spins waiting for lock to become false. I implemented this through the atomic compare and exchange operation. Similarly, the ttas lock performs compare and exchange, but also checks if "flag" is set to true in order to avoid contention misses by waiting threads. It does this by leveraging multiple read copies. The unlock for both of these is setting the flag back to false so another flag can perform the compare and exchange. The ticket lock is a fair lock that uses two atomic variables, "next number" and "now serving". The lock holder is the thread where the local "my number" is equal to "now serving". The unlock is implemented through a fetch and increment, which will give the lock to the next waiting thread. The barrier algorithm is the standard sense reversal barrier algorithm that uses an atomic sense variable to determine if all threads have arrived to the barrier. Each thread has a local "my sense" variable that is flipped when they arrive at the barrier. Once the number of ”my sense” variables have flipped is equal to the number of threads, the barrier then releases the threads. This implementation of the sense reversal barrier strictly utilizes atomic operations vs using a lock in an attempt to improve performance.

The code of this lab is organized into six different C++ files, each with a corresponding header file. There are two driver files, one for each executable for Bucketsort and counter respectively. In addition, there are four encapsulated files that handle the Bucketsort, the two different counter methods, all of the locks, and both barriers respectively.

Each of these files has their own class to keep an object oriented approach to the whole lab. The Locks file and Barriers file are encapsulated and work for both executables. The only thing needed to instantiate the Locks object is a string with the name of the desired lock type. This is determined through user input. The only public locks functions are the acquire and release function. These functions use the lock type passed through instantiation to determine which locking algorithm to implement. Similarly, the Barriers object requires a string with the desired barrier type in addition with the number of threads which is required for both barrier algorithms. The files then use the passed types and an enum in order to determine which algorithms to use. In addition to the Bucketsort from lab 1, the Bucketsort in this lab takes in two extra arguments, lockType and barrierType, in order to determine which concurrency primitive to use for the algorithm. The Counters file is implemented similar to Bucketsort, but also takes in an argument countertype when the Counters object is instantiated in order to determine if the incrementation will be controlled via locks or barriers. There is a driver file for each executable that handles the parsing of command line arguments, handling invalid inputs, and outputting the necessary information to an output file of the user’s choice. Both of these drivers share the same header file. Lastly, a shell script I wrote that uses all the test cases from lab 1 was used for testing of these primitives.

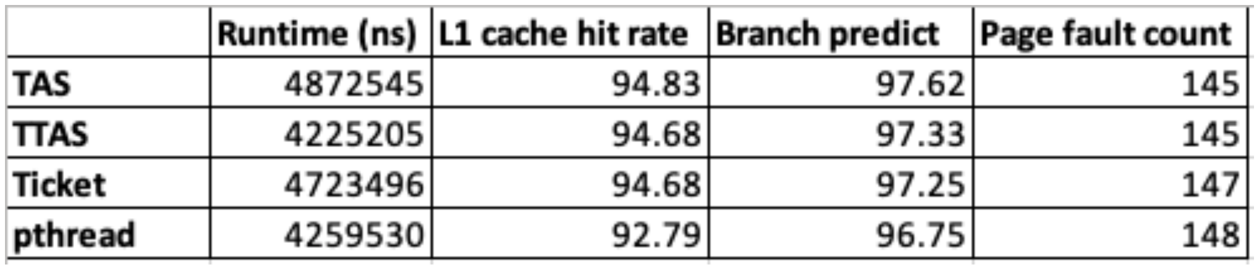
In order to compile the program, the user simply needs to be located in the Lab directory and type a single make command. The entire lab will compile and create object files for each C++ file along with the two executables “mysort” and “counter”.

In order to execute mysort, the user must follow the syntax “./mysort [--name] [sourcefile.txt] [-o out.txt] [-t NUM_THREADS] [--bar=<sense,pthread>] [--lock=< tas,ttas,ticket,pthread >]”. In order to execute counter, the user must follow the syntax “./counter [--name] [-t NUM_THREADS] [-i NUM_ITERATIONS] [--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,pthread>] [-o out.txt]”.

There are no extant bugs with this lab.

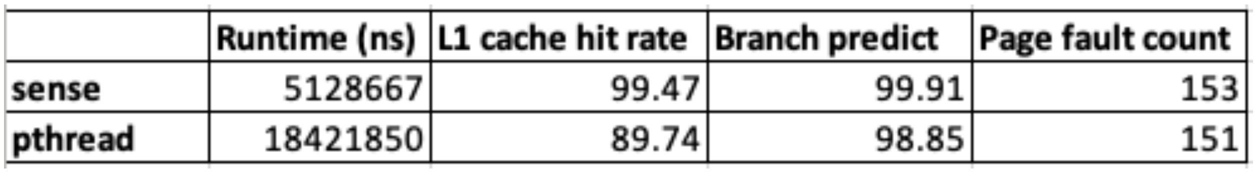
<u>Analysis: Averages after three executions</u>

<u>Locks: ./counter -t 10 -i 100 –lock=<tas,ttas,ticket,pthread> -o out.txt</u>



	Runtime (ns)	L1 cache hit rate	Branch predict	Page fault count
TAS	4872545	94.83	97.62	145
TTAS	4225205	94.68	97.33	145
Ticket	4723496	94.68	97.25	147
pthread	4259530	92.79	96.75	148

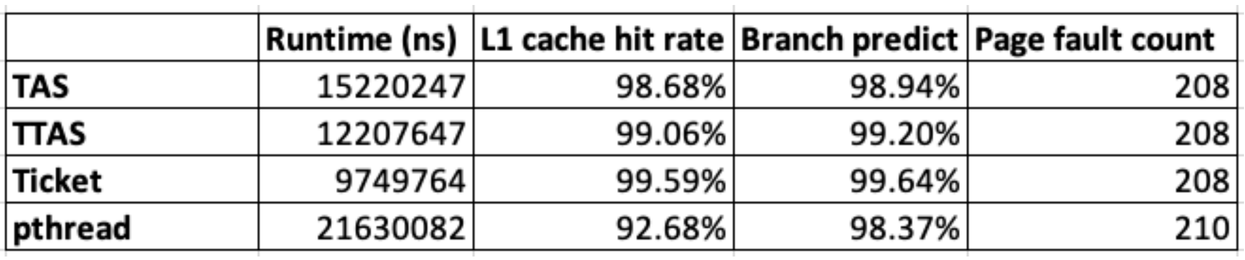
<u>Barriers: ./counter -t 10 -i 100 –bar=<sense,pthread> -o out.tx</u>



	Runtime (ns)	L1 cache hit rate	Branch predict	Page fault count
sense	5128667	99.47	99.91	153
pthread	18421850	89.74	98.85	151

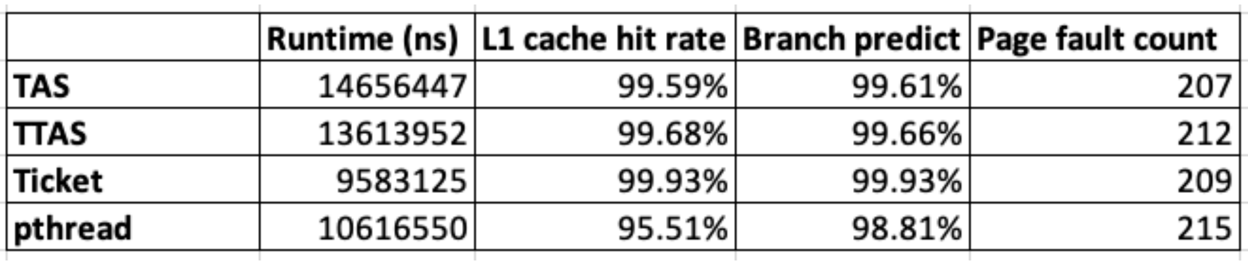
shuf -i0-700000 -n4000 > test.txt

./mysort source.txt -o out.txt -t 10 –lock=<tas,ttas,ticket,pthread> --bar=pthread



	Runtime (ns)	L1 cache hit rate	Branch predict	Page fault count
TAS	15220247	98.68%	98.94%	208
TTAS	12207647	99.06%	99.20%	208
Ticket	9749764	99.59%	99.64%	208
pthread	21630082	92.68%	98.37%	210

/mysort source.txt -o out.txt -t 10 –lock=<tas,ttas,ticket,pthread> --bar=sense



	Runtime (ns)	L1 cache hit rate	Branch predict	Page fault count
TAS	14656447	99.59%	99.61%	207
TTAS	13613952	99.68%	99.66%	212
Ticket	9583125	99.93%	99.93%	209
pthread	10616550	95.51%	98.81%	215

From the data I gathered utilizing perf, it's easy to see my sense barrier algorithm generally leads to better performance than the one through the pthread library. I believe this to be the case because I implement my barrier strictly using atomic operations rather than utilizing a lock. As expected, the tas and ttas locks have similar performance with the ttas lock having less cache misses compared to the tas lock. This is due to the extra conditional the ttas lock has that avoids unnecessary cache misses. As far as the best lock algorithm, it appears the ticket lock is the best performing when implementing the sorting algorithm, but the ttas lock seems to be the best when implementing the counter. This makes sense as the TTAS lock holder has more cache hits in the critical section which should make the counter count faster whereas fairness will perform better in my implementation of Bucketsort.