1. **(50 points, requires programming) Replicate the experiment of Slide 66 (Fig. 7.4). That is, implement the TAS and TTAS locks and compare them for a different number of threads, up to 100 threads. The critical section is a counter that increments by 1. Each thread repeatedly executes the critical section (by reacquiring the lock). The program terminates when the counter value reaches 1 million.**

See programing attachement.

This project requires cmake

In order to build the project, simply run ./run.sh

The script will handle the building and running process

The data will be written to the build folder, you can change it by modifying run.sh

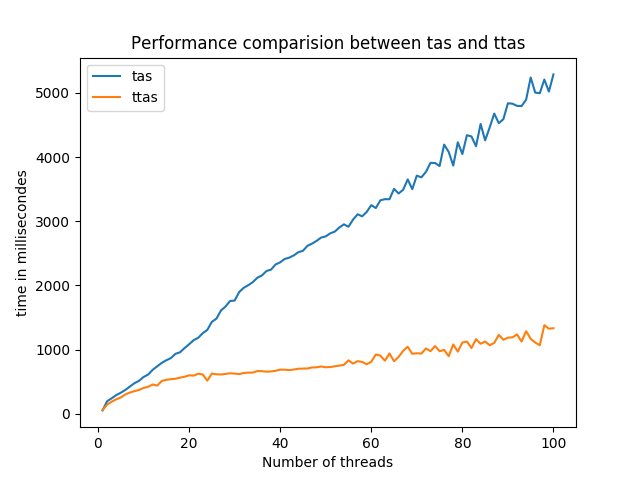
As my example, the data are under build\_ch7, the filenames are \_tas\_1.txt etc...

I run each lock 3 times, you can also simply change it by editing the python code and run.sh

python3 result.py will run the python code to plot the data

Running sample are provided under folder build\_ch7

experiment\_result.png is the final result



**2. (50 points, theoretical question) Imagine n threads, each of which executes method foo() followed by method bar(). Suppose we want to make sure that no thread starts bar() until all threads have finished foo(). For this kind of synchronization, we place a barrier between foo() and bar().**

**First barrier implementation: We have a counter protected by a test–and–test–and–set lock. Each thread locks the counter, increments it, releases the lock, and spins, rereading the counter until it reaches n.**

**Second barrier implementation: We have an n-element Boolean array b, all false. Thread zero sets b[0] to true. Every thread i, for 0 < i < n, spins until b[i−1] is true, sets b[i] to true, and proceeds.**

**Compare the behavior of these two implementations on a bus-based cache coherent architecture.**

For the first barrier implementation, when a thread finishes foo() and arrives the barrier, it will acquire the lock, increment the value of the counter and then release the lock. When it is acquiring lock, writing to the counter, or releasing the lock, it will cause invalidation traffic on the bus. After incrementing the counter, all the thread spinning on the counter will experience a cache miss, and reread the value, which is expensive. Moreover, the number of threads is increasing as more threads arrives the barrier after finishing foo(); The number of threads spinning on the lock will be random, if one threads sets the Boolean flag, other threads will experience a cache miss, reread and try TAS, which will do the writing the lock Boolean flag, causing invalidation traffic.

For the second barrier implementation, when a thread k finishes foo() and arrives the barrier, it will spin on b[k-1] (except thread 0), once b[k-1] becomes true, it will set b[k] to true, and spin on b[n-1]. When one thread sets the value, only one thread is possibly experiencing cache miss because other thread is likely using different cache line. When all other thread is spinning on b[n-1], invalidation traffic happens only when b[n-1] is set, which only happens once of the entire process. However, we need to be careful about the false sharing problem, properly padding the array can solve the issue.

Thus, the second barrier is likely to give better performance outcome than the first one.