ECE 5750 – Parallel Computer Architecture
Programming Assignment #3

Objective

The objective of this assignment is to become familiar with different lock algorithms for shared-memory machines. You will have the opportunity to measure the performance of your lock algorithms on our parallel server.

Lock Algorithms

In this assignment you will implement and measure the performance of four lock algorithms, namely, test&set (TS), test&test&set (TTS), ticket lock, and array-based queue lock.

The pthreads library provides an easy test&set implementation through the procedures *pthread_cond_wait* and *pthread_cond_signal*.

The ticket-based locks were discussed in class where each processor copies and increments a ticket variable *t* as it tries to acquire the lock. Then it spins on a second variable *r* until the value of *r* equals the ticket value of this processor. When a processor leaves the critical section it increments *r*. Note that the copy and increment (technically known as *fetch&increment*) of *t* must be atomic, in the sense that it should be mutually exclusive. Use the TTS lock provided to you to protect this *fetch&increment*. The increment of *r* need not be protected.

In array-based queue locks each lock is implemented as an array of size *p* where *p* is the total number of processors. The lock acquire involves atomically copying and incrementing a shared index variable and spinning on the array location indexed by the copied value. Releasing a lock involves resetting the corresponding array location and setting the next array location if a processor is waiting. Each array location should be allocated on a different cache line to avoid invalidations due to false-sharing. Assume a large enough cache-line size (say, 256 bytes) and put that amount of padding between consecutive lock locations. Use TTS locks for implementing any atomic code section that may be a part of lock acquire or release. A good treatment of these topics can be found in the recommended textbook (pages 337 to 351).

Microbenchmark

In this assignment you will use a microbenchmark to measure the performance of your lock algorithms. The worker thread of the program should implement the following pseudocode.

```
for i=0 to N-1
    record_timestamp(pid)
```

```
    lock_acquire(lock)
    for j=0 to k-1
        q++
    endfor
    lock_release(lock)
    for j=0 to m-1
        p++
    endfor
endfor
record_timestamp(pid)
a[pid] = p+q
```

where `a` is a shared array.

You will write the procedures `lock_acquire` and `lock_release`, which will implement different lock algorithms. Note that the type of the variable `lock` may change depending on the algorithm.

The `for` loop between `lock_acquire` and `lock_release` essentially simulates a critical section with a certain delay depending on the value of `k`. The `for` loop outside the critical section introduces a certain amount of delay between two consecutive lock acquires depending on the value of `m`. Finally, the assignment `a[pid] = p+q` just ensures that there is some use of the values *p* and *q*, so that the compiler does not optimize the whole code away.

Note how execution time is measured in this code. The overall start time should be computed as the minimum among all start timestamps captured across all threads, while the overall end time should be the maximum across all thread timestamps captured at the end. The difference of these two is the execution time of the parallel section. Carry out this max/min computation in `main`, after all worker threads are done.

You need to experiment with three values of `m`, namely, 0, a suitable positive constant, and a value that is proportional to `pid` (choose a suitable proportionality constant). A constant value introduces lock accesses from all the processors roughly at the same time. A value proportional to `pid` staggers the lock accesses. Experiment with at least five different values of `k`. Make sure that you cover a large enough range of values. Start with 0. Pick `N` such that the timed portion of your program with TTS lock executes for at least 10 seconds for both `k` and `m` zero. Use the same value of `N` for all the experiments. Fix the number of processors to eight (but your code should work for an arbitrary number of processors). To test the correctness of your locks, feel free to write meaningful critical sections (which actually read and write shared variables), but please do not submit these results.

Submission

You must submit the source code of your program and the lock implementations in a ZIP file. You are allowed to use the web or any other consulting resource to find the best possible approach; however, **you must properly cite all your references**.

You must also submit a four-page report (PDF) of your implementation. The report should contain three different plots for three different values of `M`. Each plot should show execution time against the values of k. On the same plot show four different performance curves for four different lock algorithms. Please distinguish the curves by putting appropriate legends (e.g. TS, TTS, ticket, queue). Precisely explain what you understand from the results and draw meaningful conclusions.

All your submissions must be done to CMS; your report should also be submitted to the HotCRP server for double-blind review.