Design Document

Alex Miller

May 4, 2021

Modules, data structures, interfaces, invariants:

Our code base will consist of the following modules:

- The highest ranked modules containing C code will be four files, one called serial.c, another called parallel.c, and a pair called my_test_serial.c and my_test_parallel.c The first two modules will implement the functionality of a serial counter and parallel counter. The last pair will implement a proprietary test of lock performance, this will be discussed later.

  serial.c

  - This file will describe an executable that takes the following input parameter(s)
    - B : a number to count to
  - The purpose of this file is to call a local method called *int counter*(*int B*). This method increments a counter, starting from 0, B times. It returns the value it counts to.

    The file calls this method, measures the amount of time it takes to return, and finally prints the following output to the terminal: $< count >, < time >$, where *count* specifies the value returned by *counter*() and *time* denotes how long it took *counter*() to return.

  parallel.c

  - This file will describe an executable that takes the following input parameter(s)
    - B : a number to count too
    - n : the number of threads to launch
    - L : the type of locking algorithm our threads should acquire to increment the counter. This option will be set with a single character; setting this character to t, p, a, and m will direct threads to use our Test and Set Lock, the pthread Mutex, our Anderson's Array Lock, and our MCS lock. The specifics of these lock implementations will be discussed later.
  - The purpose of this file is to call a local method called *int counter*(*int B*, *int n*, *void *lock_methods*). The parameters of the method represent the following:
    - B : the same as above
    - n : likewise
    - lock_methods : a pointer to a struct containing the initialized lock and pointers to its methods, *lock*() and *unlock*()

    This method increments a shared counter, starting from 0, B times, partitioning the work among n threads; each thread increments the shared counter B/n times. This counter is in turn protected by the lock specified by the *lock_methods* argument; the method's threads call the *lock*() and *unlock*() methods on the *lock* specified in *lock_methods* in order to use this lock. It returns the value it counts to.

    The file takes in user input, initializes the specified lock, initializes the appropriate *lock_method* instance, and calls this method. The file then measures the amount of time *counter*() takes to return, destroys the initialized lock, and finally prints the following output to the terminal:

$< count >, < time >$, where *count* specifies the value returned by *counter*() and *time* denotes how long it took *counter*() to return.

A note on ordering; I chose to initialize locks *before* timing their performance in order to get a better idea of contention and throughput; removing the complication of having to account for lock initialization overhead should simplify our reasoning during analysis. Of course, initialization is not an insignificant factor in lock performance, so this is something to keep in mind.

my_test_serial.c and my_test_parallel.c

- These files will test the effect of critical section size on lock performance and contention. The idea is that by increasing the size of the critical section, we will be able to measure the effect of contention on lock performance; a bigger critical section means more contention.
- my_test_serial.c will describe an executable that takes the following input parameter(s)
  - B : this integer value corresponds to a set amount of work that should be completed. It can be interpreted as a total number of operations. In reality it will represent a set amount of milliseconds.
  - t : how many operations should be completed per loop (the size of the critical section)? Here operations are a proxy for critical section runtime, measured in milliseconds.

  The purpose of this file is to call a local method called *int sleeper*(*int B*, *int t*). This calls *usleep*(*t*) B/t times; this should be generally equivalent to sleeping for B milliseconds.

  The file calls this method, measures the amount of time it takes to return, and finally prints the following output to the terminal: $< time >$, where *time* denotes how long it took *sleeper*() to return.
- my_test_parallel.c will describe an executable that takes the following input parameter(s)
  - B : this integer value corresponds to a set amount of work that should be completed. It can be interpreted as a total number of operations. In reality it will represent a set amount of milliseconds.
  - t : how many operations should be completed per loop (the size of the critical section)? Here operations are a proxy for critical section runtime, measured in milliseconds.
  - n : the number of threads to launch
  - L : the type of locking algorithm our threads should acquire to increment the counter.

  The purpose of this file is to call a local method called *int sleeper*(*int B*, *int t*, *int n*, *void *lock_methods*). The parameters of the method represent the following:
  - B : the same as above
  - t : likewise
  - n : likewise
  - lock_methods : a pointer to a struct containing the initialized lock and pointers to its methods, *lock*() and *unlock*()

  This method calls *usleep*(*t*) $B/(t * n)$ times across n threads; this should be generally equivalent to all threads sleeping for combined total of B milliseconds. The critical section, represented by *usleep*(*t*), is protected by the lock specified by the *lock_methods* argument; the method's threads call the *lock*() and *unlock*() methods on the *lock* specified in *lock_methods* in order to use this lock.

  The file takes in user input, initializes the specified lock, initializes the appropriate *lock_method* instance, and calls this method. The file then measures the amount of time *sleeper*() takes to return, destroys the initialized lock, and finally prints the following output to the terminal: $< time >$, where *time* denotes how long it took *sleeper*() to return.
- my_test_serial.c will be used as a base measurement of performance; trials will involve calling my_test_serial.c for varying amounts of t and comparing the runtime of my_test_parallel.c for a that value of t, and varying values of n and varying lock methods. All locks will be tested this way. B will remain fixed in these tests.

- By varying the values of n, t, we can get a sense of how different locks perform in the presence of varying levels of contention, and whether the size of the critical section can act as an equalizer of lock performance.

- The locks mentioned above, if applicable, will be implemented in a file called lock.c. The file will describe the implementation of a Test and Set lock, an Anderson's Array Lock, and an MCS lock, as specified in the textbook and translated into C. The Mutex lock will be called from the pthread library, but wrapped so as to work with our *lock_method* scheme. Each lock will have the following methods associated with it:

  - init(int n): initialize a lock to synchronize n threads; n is only used in initializing locks that allocate space for locks prior to use, such as Anderson's lock. Otherwise, n is discarded.
  - destroy(void *lock): destroy the initialized *lock* and free any memory allocated to support it
  - lock(void *lock): lock *lock*
  - unlock(void *lock): unlock *lock*

  By wrapping all methods in a data-type agnostic interface, we can cut down on the complexity of our code base. Of course, this complicates our locks somewhat, but this shouldn't increase most method's runtime. There is the concern that, in wrapping the Mutex lock in our interface, we increase the time needed to call *pthread_mutex_lock* and *pthread_mutex_unlock*, but this increase should be minimal; in the presence of contention, this delay should become negligible. However, it is a concern we should keep in mind while testing.

  A note on padding; the Anderson's Array lock's performance should fluctuate as a function of the padding utilized in its flag array. The machines that we will be testing, explicitly, have 64 byte cache lines. Therefore, in order to get the best performance, our flag variables should be 64 bytes each. However, further is testing is probably needed to validate this value.

- Our last module consists of testing and analysis protocols, test_script.sh and analysis.py

  - test_script.s will consist of a shell script that utilizes serial.c, parallel.c, my_test_serial.c and my_test_parallel.c. This test script will execute these files and record their output, generating experimental data in an automated fashion and storing that data in csv files. It can also verify if first two executables generate good output by comparing the $< count >$ portion of output to the B input.

    I can use this script to automatically run and format my experiments using as many trials as I want, varying the input of B in applicable cases. I will most likely use $B = 1120$ for testing my first two executable. This value is useful in being a common multiple of our differing values of n, but initial testing is needed in order to deduce whether this value is large enough to minimize trial variance. I will use 5 trials per test, in order to get a fair spectrum of inputs while not hogging class resources. Again, more testing is needed to determine if this is an appropriate values.

    For the my proprietary test, I will use $B = 3136$; this test will be performed across various cross products of N, T, and L, where $N = \{1, 2, 4, 8, 14\}$, $T = \{1, 2, 4, 8, 14\}$, and L is the set of locking algorithms we implemented. The value of B was chosen to ensure clean division across all possible trial inputs. I will use 5 trials per test, in order to get a fair spectrum of inputs while not hogging class resources. Again, more testing is needed to determine if these are appropriate values.
  - I have avoided the use of python to analyze my data in past assignments, mostly due to increased project complexity and implementation time. However, I struggled in my last assignment to utilize quantitative analysis, getting lost in the shapes of curves and using approximations to support or deny my hypotheses. My hope is that through more rigorous (and automated) analysis, I can better present my findings without having to do a significant amount of hand-analysis. To that end, analysis.py will be responsible for analyzing data, making graphs, and outputting quantitative data to compare the performance of locks.

Additionally, if I am to use a script to process my data, it will be easier to process and present trial data without having to discard data points, as more data will not overwhelm a python script in the same way it overwhelms my ability to make nice spreadsheets.

Test plan:

Testing the correctness of top level modules as well as our testing scripts will be done through careful audits of the code base and reasoning about output; these files describe testing frameworks who's correctness is relatively trivial to demonstrate by seeing if testing inputs correspond to output.

What is necessary to verify the correctness of are the testing methods specified in these files and the locks they utilize. As to the correctness of the counter specified in serial.c, this should be trivial, as it should just be a for loop that increments some variable. We will know if this counter is working if its output is reliably equivalent to the input of B.
The correctness of the counter specified in parallel.c will be a bit trickier to verify; though it itself might not be very complicated, the locks it utilizes will need to be verified. That being said, we can test the structure of the counter by calling it using the Mutex lock (which is reliably correct) and seeing if its output corresponds reliably with that of the counter in serial.c for common, and sufficiently large, values of B. Once we have assured ourselves of the correctness of the parallel counter, we can test the correctness of our other locks by seeing if the parallel counter reliably outputs the correct value of B while using said locks. This should be enough to infer the correct functioning of our locks.

As to the correctness of my implementations of my_test_serial.c and my_test_parallel.c; initially we will need to see if our method of varying the size of the critical section is valid. This can be done by testing the outputs of my_test_serial.c and seeing if it is proportional, if not identical, to varying inputs of B and t; the sleeper specified in my_test_serial.c should return in B seconds for all such inputs.
The correctness of the sleeper specified in my_test_parallel.c will be inferred from both the apparent correctness of our locks following testing with serial.c and parallel.c as well as the correctness of our method of increasing the size of the critical section following testing of my_test_serial.c. Since the sleeper specified my_test_parallel.c utilizes the same basic framework and data structures as these files, the correctness of the former should follow from that of the latter.

Expected Performance (Performance Hypotheses):

We take as an assumption that, across all tests, serial implementations will run faster than comparable parallel implementations. This is because, in all cases, our tests center around comparing different locks ability to operate on a single shared resource or critical section; there is therefore no opportunity for parallelism from which we can expect to see speedup over serial implementations.

Idle Lock Overhead

The purpose of this test is to test lock performance in the complete absence of contention. Therefore, locks will perform better in proportion to the relative simplicity of their lock and unlock methods. Locking algorithms with simpler lock and unlock methods will experience greater throughput, and be more likely to approach ideal, serial performance.
We should, then, expect to see the best performance from our Test and Set lock, followed by the Mutex lock, then our MCS lock, and lastly our Anderson's Array lock.

The Mutex lock operates by performing atomic get and increment operations and testing the output. Our Test and Set lock operates by performing a test and set operation. Both of these methods should be quick in the absence of contention; however, the former method might not be as quick as the latter, leading us to expect greater performance out of the Test and Set lock.

Both of these locks are less complex than our MCS lock which, in the absence of contention, involves get

and getAndSet invocations. MCS, in turn, should outperform our Anderson Array Lock, which utilizes both getAndIncrement as well as a modulo operation, even in the absence of contention.

Lock Scaling

The purpose of this test is to test lock performance in the presence of contention from multiple threads trying to enter a shared critical section. Therefore, locks will perform better in proportion to their ability to handle contention. Performance will vary, then, as a function of n; locks that are better at handling contention will experience less slowdown than locks that are worse at it. Additionally, we should expect all parallel implementations to experience greater slowdown for greater values of n; since more threads merely implies greater contention and not increased parallelism, we should expect more threads to worsen performance for parallel lock tests.
In terms of relative performance in the face of larger values of n, we should expect our MCS lock to perform best, followed by our Anderson's Array lock, then the Mutex lock, and finally our Test and Set lock.

The MCS and Anderson's Array lock should showcase similar performance, as they utilize the same principle of spinning on local cached memory, which cuts down on cache invalidation and misses. MCS should still outperform Anderson's due to the facts that it doesn't use a costly modulo operation, and uses less memory; each flag in Anderson's Array lock should be 64 bytes in our implementation, which is far larger than the 16 bytes required by MCS's QNode structure (8 bytes to store a pointer to the next QNode, 1 to store a boolean, and 5 bytes of padding).

Both of these locks should outperform the Mutex and our Test and Set lock, as these locking algorithms don't spin on local sections of memory, and therefore bring about more cache misses and invalidation. This effect should be exacerbated for larger values of n. Furthermore, we should expect the Mutex to outperform our Test and Set lock; the Mutex employs sleeping in order to reduce the number of cache misses and invalidation, while the Test and Set lock does not. Thus Mutex should perform better than Test and Set by virtue of employing backoff.

My Test

The purpose of this test if to determine the effect of critical section size on lock performance. In general, we should expect less relative slowdown of our parallel implementations over our serial implementation for larger critical sections (larger values of t), as the increased necessary compute times in the critical section should otherwise cancel out increased waiting time to acquire a lock due to contention. That being said within the tests for a given lock, we should still see more relative slowdown for larger values of n; this reflects our previous hypothesis that increased contention for a lock should increase slowdown in all cases.

In lieu of a hypothesis on relative performance, we should expect there to be a point for all locking algorithms at which the size of the critical section outweighs slowdown due to contention. Therefore, for a large enough value of t, we should expect all locks to showcase near-ideal performance for all values of n.