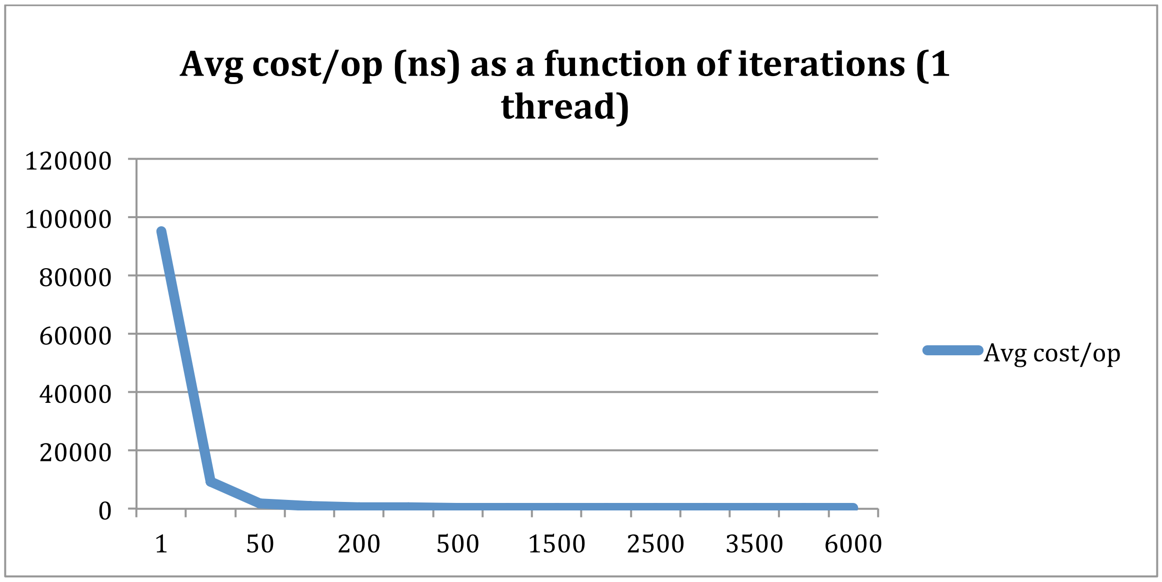
Jayendra Jog
904437296
jayendra@ucla.edu

Yu-Kuan (Anthony) Lai
004445644
yukuan.anthony.lai@gmail.com

1.1
===
1. It generally takes (roughly) greater than 5000 total operations for an error to occur consistently. It takes this many threads and iterations because the race conditions are not accounted for in the add function. It is possible for the value of pointer to change incorrectly if the scheduler gives control to another thread in the middle of the add function. For example, if one thread has sum = 5 + 1 = 6, and then control passes to another thread, which runs 100 iterations of sum = 5 + 1 => 105, and then sets pointer = 105, then when the first thread regains control, it will set pointer = 6, effectively making the 100 additions from the second thread useless. This showcases how a large number of operations causes problems. Moreover, with more iterations and threads, there are more chances for these race conditions to come about.

2. With less iterations, there are significantly less chances for the race conditions to come about. Moreover, with less iterations, each thread has less time when its being run, so there is a lesser chance for the scheduler to interrupt it.

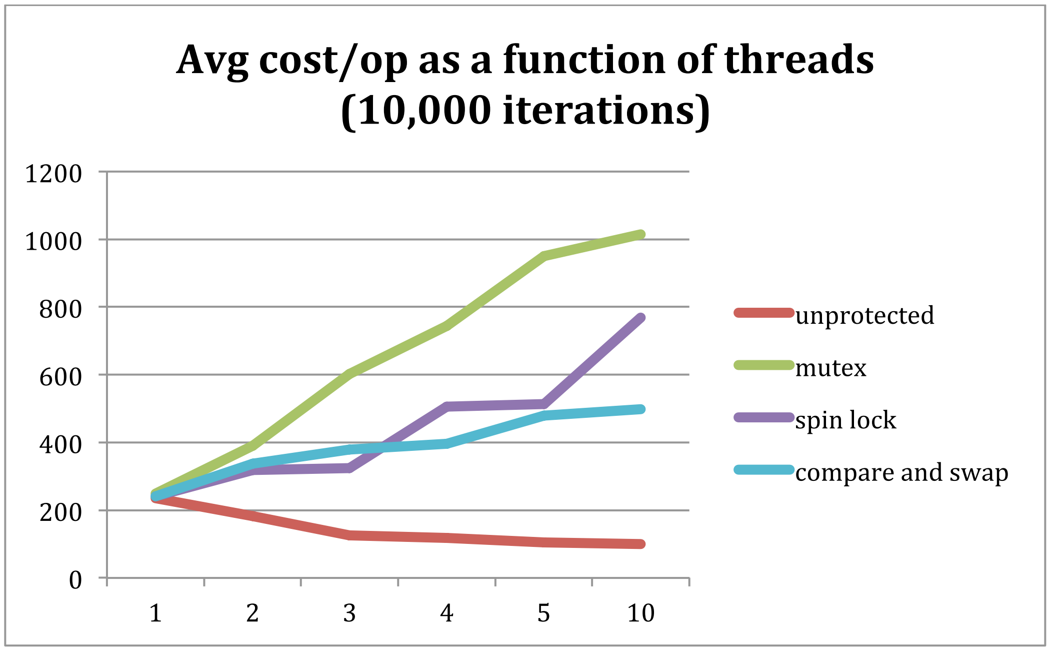


1.2
===
1. The average cost per operation drops because there is a high overhead for the allocation of memory, creating and execution of threads, and the ultimate joining of threads.

2. We can know the correct cost of implementation by seeing what the cost/op converges towards as the number of iterations approaches infinity. When the iterations becomes very high, then the cost due to overhead becomes insignificant, so it represents a much more accurate and "correct" cost per operation.

3. When yield gets called, a system call causes the kernel to receive control and then pass control over to the relevant thread. Passing control over to the kernel is a slow process, and doing this for every single interation of every single thread accounts for the slowdown.

4. Using --yield does not allow us to get valid timings. The yield function gives control to the CPU for a variable amount of time, and thus it becomes impossible to get an exact time calculation. Moreover, each yield call accounts for even more time, so the yield calls in themselves will skew the cost/op to increase, even though the operation doesn't actually need all of that time.



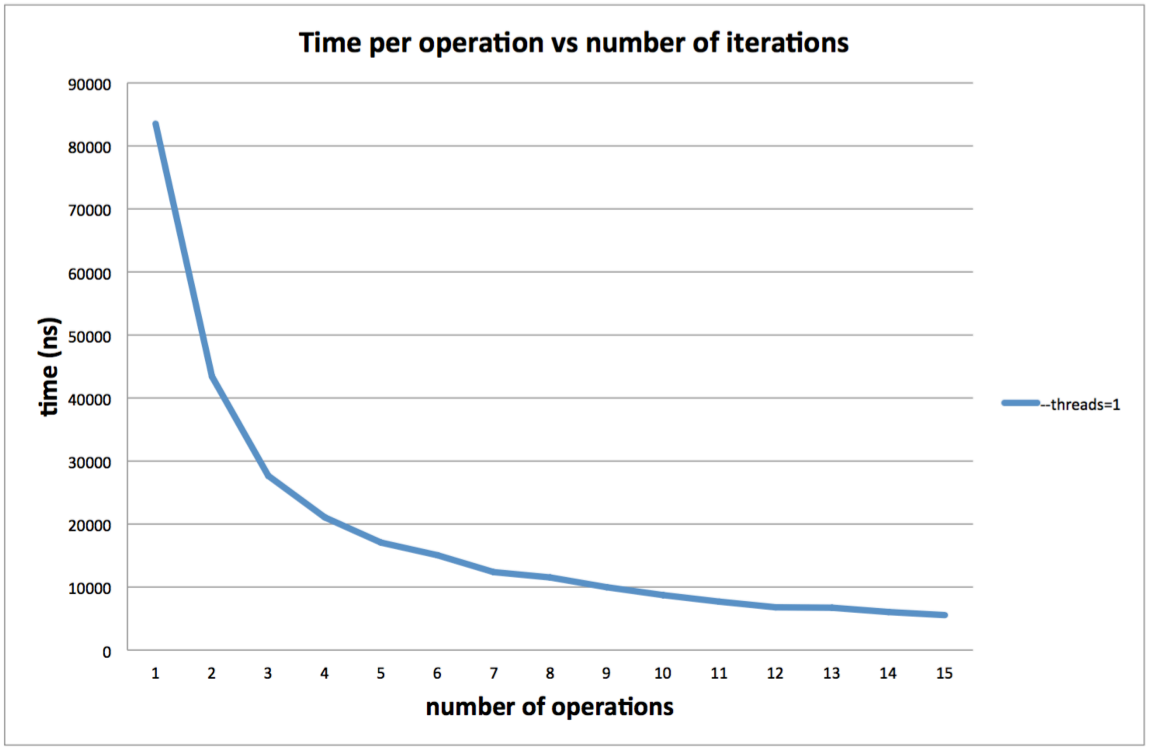
1.3
===
1. All of the options perform similarly for low numbers of threads because the computation of locks for a low number of threads doesn't require too much

time. Essentially, the price per lock is very cheap when there aren't too many threads, so the time increase due to the lock is relatively insignificant.

2. As the number of threads increases, there are more and more threads competing for the same resource (the lock). This requires more kernel calls to figure out which thread should get access, and the end result is that with more threads, there is a greater amount of time needed.

3. Spin locks spend a lot of time waiting, so they are extremeley inefficient for a large number of threads. The more threads there are, the longer each spin lock takes, which is why there is a net increase in the amount of time when the number of threads increase.

Part 2



2.1

===

(1) It appears that the total time for all operations remains fairly consistent regardless the number of operations. As can be seen from the graph, as the number of operations increase, the time for each operation decrease. This is probably because of the overhead involved in creating the thread and starting the computations. Pthread system call have a lot of overhead compare to inserting and deleting elements from the SortedList. Thus, by increasing the amount of operations, the total time is roughly the same. To correct this effect, we can implement the tester program without any pthread calls. Obviously this is

not a good option because we still want to test multithreads. Thus, we can call the tester program with lots of iterations, averaging the overhead out to each operation.
As the number of iterations increase above a certain point, the time for each operation increase. This is because in insert and lookup, the list become longer and it takes longer to insert and lookup the element with specific key.

(2)
When run the tester with 25 iterations and 5 threads, the tester reports with seg fault for about 50% of the time.
When run the tester with 25 iterations and 10 threads, the tester reports with seg fault for about 90% of the time.

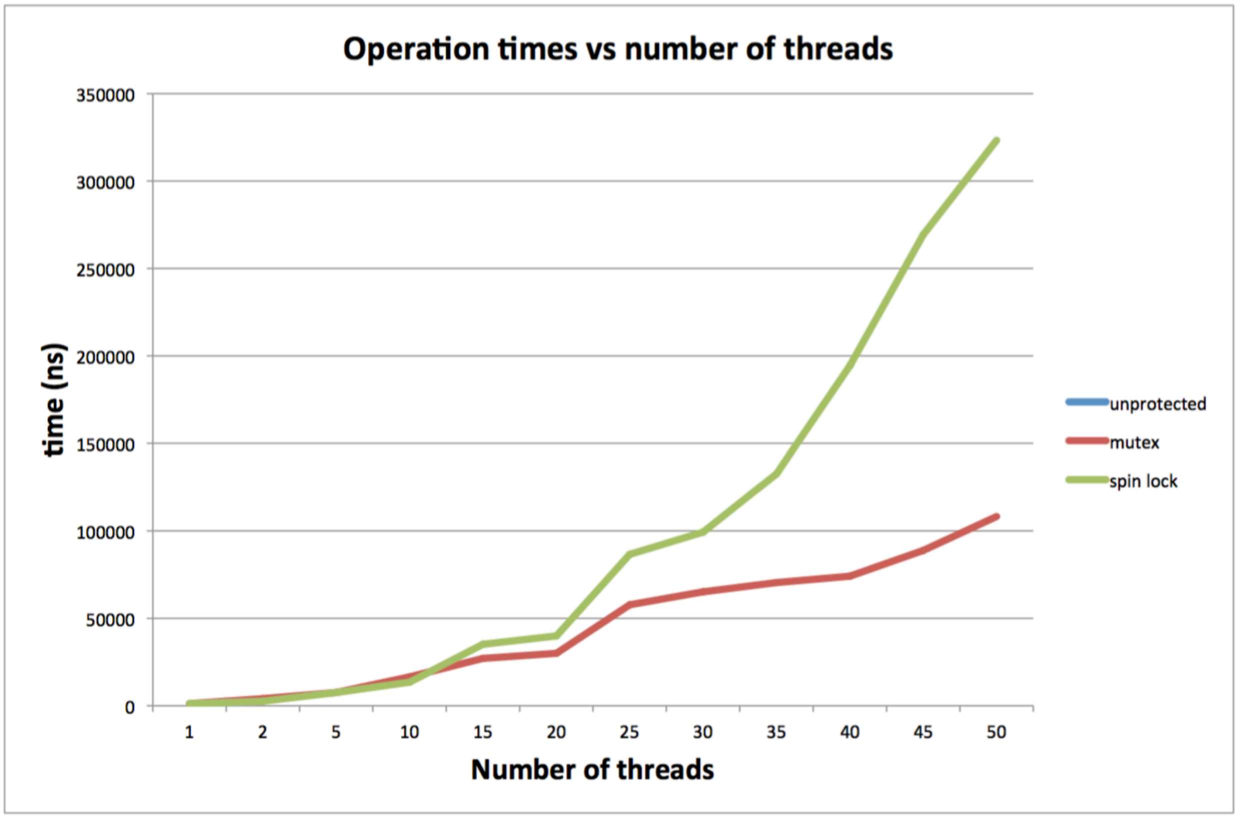
When run with --yield=i
When run the tester with 25 iterations and 2 threads, the tester reports with seg fault for about 50% of the time.
When run the tester with 25 iterations and 3 threads, the tester reports with seg fault for about 90% of the time.

When run with --yield=d
When run the tester with 25 iterations and 2 threads, the tester reports with seg fault for about 50% of the time.
When run the tester with 25 iterations and 3 threads, the tester reports with seg fault for about 90% of the time.

When run with --yield=is
When run the tester with 25 iterations and 2 threads, the tester reports with seg fault for about 50% of the time.
When run the tester with 25 iterations and 3 threads, the tester reports with seg fault for about 90% of the time.

When run with --yield=ds
When run the tester with 25 iterations and 2 threads, the tester reports with seg fault for about 50% of the time.
When run the tester with 25 iterations and 3 threads, the tester reports with seg fault for about 90% of the time.

(3)
When using synchronization with either mutex or spin lock, it eliminates most of the problems, specifically those seg fault from previous problems. When running with --yield option and 100 iterations with 25 threads, the program still behaves normally without seg fault.
NOTE: pthread_create fails at --iterations=100 --threads=100

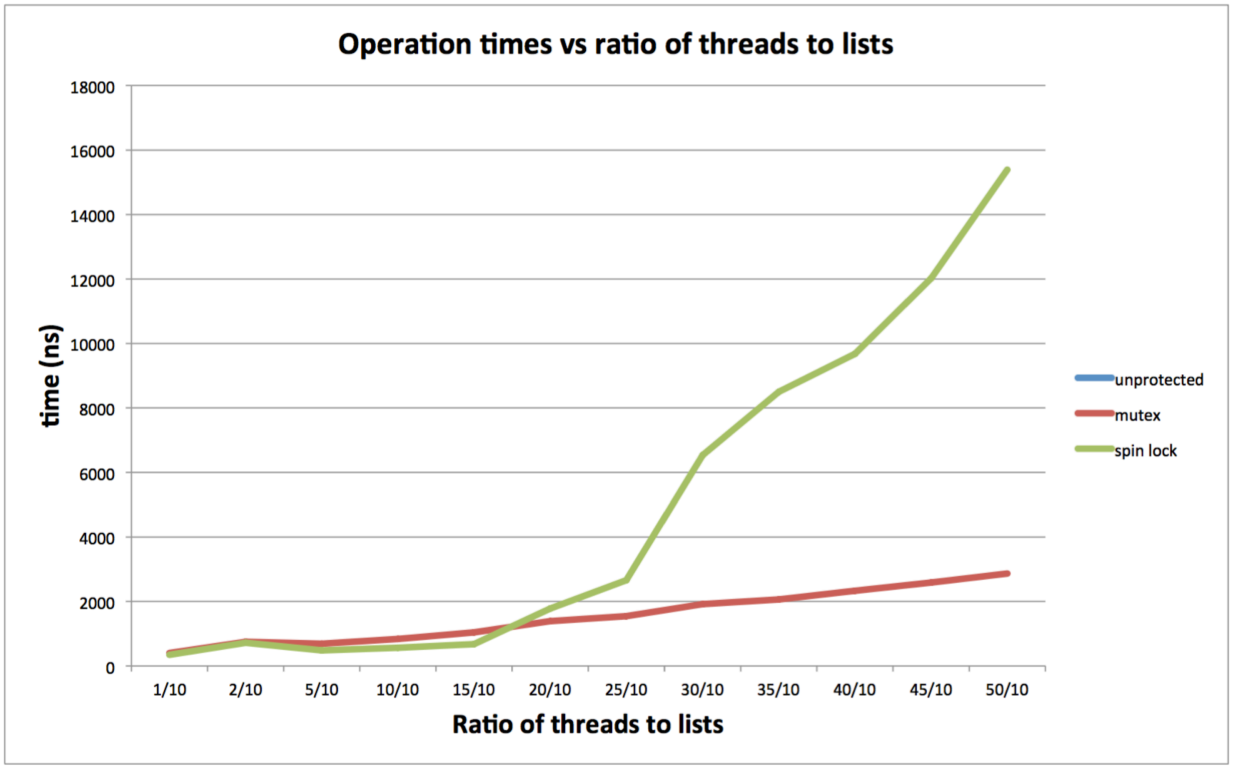


(Note: all tests are ran with 500 iterations; unprotected with 1 thread, which is not visible on the graph has time = 1185ns)

2.2

===

In both Part 1 and Part 2, the time per operation increases as the number of threads increase (in the protected case). This is because as the number of threads increase, each thread has to wait for more threads to finish their jobs and release the lock. The difference is that in Part 2, the slope of the graph is much greater than that of Part 1. This is because that compared to adding integers, inserting and deleting from linked list are more complicated operations. With linked list, there are much more operations involving dereferencing pointers. This can thus explain the greater increase in time per operation in Part 2.



(Note: all tests are ran with 500 iterations; unprotected with 1 thread, which is not visible on the graph has time = 392ns)

2.3

===

(1) Similar to 2.2, as the ratio of threads to lists increase, there are more threads locking and waiting on the same lock. It can be seen clearly from the graph above, especially in the spin lock case, that as the number of threads increase, the time per operation increases.

(2) In the multiple lists case, there are multiple locks as well. Each list has its own lock and when a particular thread realizes that it needs to add the element to a particular list, based on the computed hash number, it will acquire that lock. Thus, even though there might be a lot of threads, if there are a lot of lists as well and the keys of the elements are truly random, on average, each thread will be adding to a different list and therefore, don't have to wait for each other. This is the reason that for this particular measurement, it's important to consider the ratio between threads to lists, as opposed to just the number of threads.

Part 3

===

1. The mutex must be held while pthread_cond_wait is being called otherwise the shared data between the various threads might get overwritten. Essentially, it's possible for race conditions to arise in the time between the condition getting evaluated and the wait starting to happen. If the thread calling the function doesn't have a mutex, then another thread can access the shared memory in the time of the race condition, and that can have detrimental results.

2. The mutex must be released otherwise the program will run infinitely. The other threads will all be waiting for the mutex in order to run, and the blocked thread will be waiting to be woken up and given the mutex it already possesses.

3. The mutex must be reacquired to ensure that shared data cannot be incorrectly overwritten. When the conditional check is done, then the original thread needs to be able to run without running into race conditions.

4. If the mutex is released prior to the function call, then it's possible for the variables needed for evaluation the conditional to have their values changed by other threads, resulting in the condition getting incorrectly evaluated. Therefore, to ensure proper execution in every case, it must be done inside of pthread_cond_wait.

5. This cannot be done with a user-mode implementation of pthread_cond_wait. This needs to be an atomic operation, and the only way to ensure that it's an atomic operation is by using a kernel system call. In user mode, nothing is guaranteed, so the only safe way to do this is through kernel mode.