Lab 4 Report

Elton Leong 204 457 607

James Wang 904 439 931

Part 1

1.1

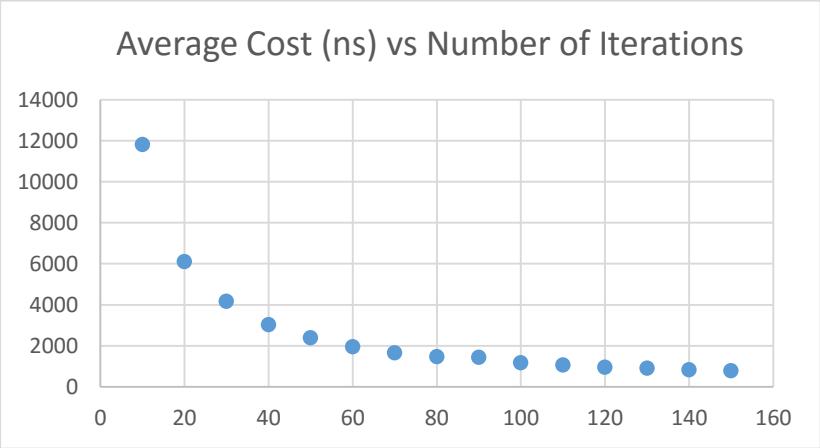

Figure 1: Graph of the average cost per operation in nanoseconds against the number of iterations specified in the input, on a single thread with no race condition protection. This graph deals with the trend for small numbers of iterations.

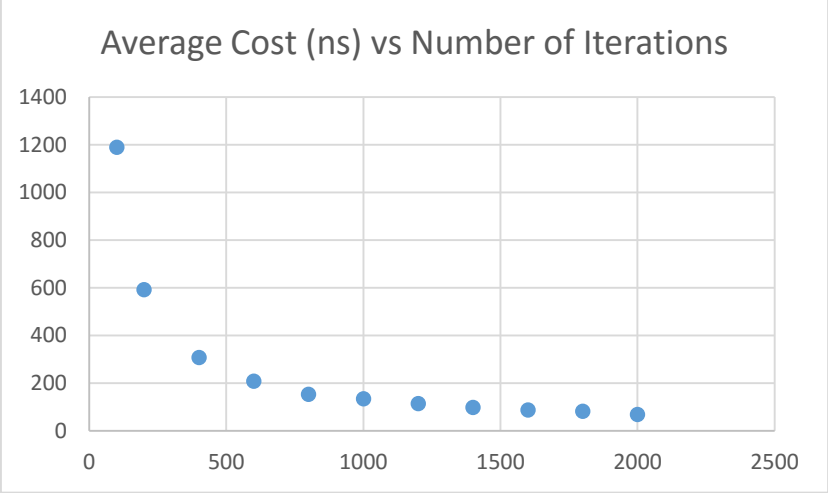

Figure 2: Graph of the average cost per operation in nanoseconds against the number of iterations specified in the input, on a single thread with no race condition protection. This graph deals with the trend for large numbers of iterations.

It takes at least 2 (though having more helps) and around 5000 iterations for the counter to begin consistently producing an incorrect value.

Example call of this script:

$ make addtest

$ ./addtest --sync m --threads 10 --iterations 10000

1.

We realize that if we use only 1 thread and a ridiculously high number of iterations that the number is never inconsistent. Thus, it is reasonable to conclude that the race conditions that are introduced by creating multiple threads without accounting for said race conditions are what is causing the problem. The reason why this occurs is because when we have an unprotected shared variable, the time between changing a value and appropriately updating it may not be atomic. For example, the counter's value may be at 5, then incremented to 6. However, there may be a race condition in which another thread examines the counter at value 5 and increments it to 6 (storing it in sum, just like the previous thread). The problem here is that the value is supposed to be 7, but now no matter what we do we can only update the counter to 6.

2.

The reason why a small number of iterations is unlikely to produce such a failure is simply because a race condition is much less likely to occur. There is also the mathematical possibility that even if our wrapper function fails to account for, hypothetically, 3 increments, there is still the possibility that a parallel race condition fails to account for 3 decrements, balancing the value by pure happenstance. This possibility is viable for, say, 10 iterations in terms of probability, but the same is not true for, say, 500,000,000,000 iterations (in fact, at this point the chances of success are next to nil).

1.2

1.

At the beginning of our main function we have some overhead that handles parsing and thread creation (even though for this graph it's only the overhead for a single thread). This time remains more or less the same because we only need to perform it once before it starts performing iterations. Thus, as we increase the number of iterations this overhead is spread out more and more thinly,

2.

The "correct" cost can be obtained by accounting for the thread creation overhead and subtracting it from the recorded time value before calculating the average time per operation. If we do this, the relationship should become much more linear regardless of the number of iterations we use.

3.

Yield runs are far slower because there is overhead associated with calling the pthread_yield() function. This overhead is enough to be noticable even if there is only one thread (likely because the entirety of the call isn't just potentially switching to a different thread). The big downside of setting the yield option is that this will make the yield be called for every single iteration! This is drastically more expensive than the simple calculation no matter how we look at it.

4.

We could potentially get valid timings of the operations if we could account for every pthread_yield() call just like we account for the initial overhead in thread creation. Unfortunately the behavior of this function can vary and thus its runtime could hover between significantly different runtimes in nanoseconds. As such it is not very viable to perform timings with yields, though theoretically if we modeled the function's average properly we could account for it as well.

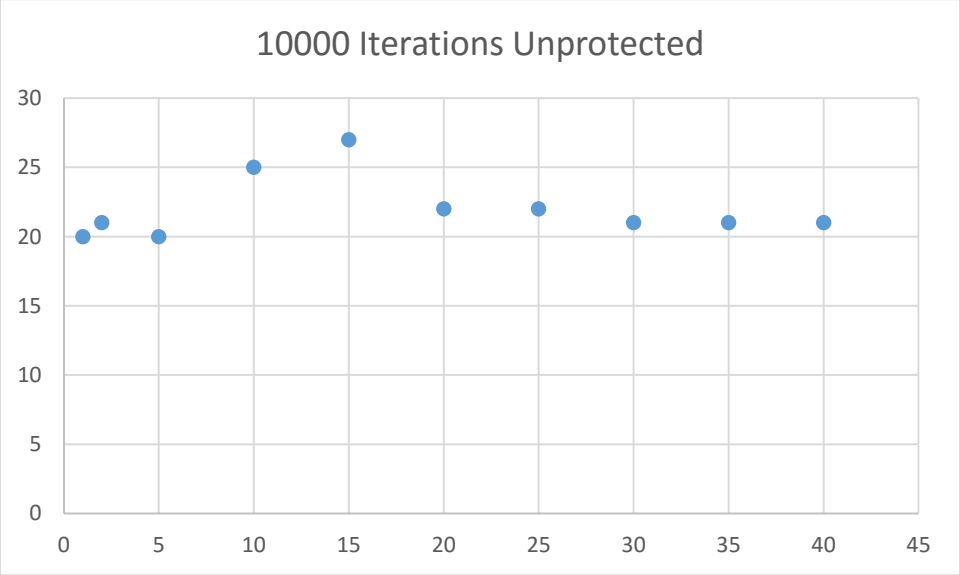

Figure 3: Average cost in nanoseconds graphed against the number of threads with an unprotected add function (add). The number of iterations used for each data point is a constant 10,000.

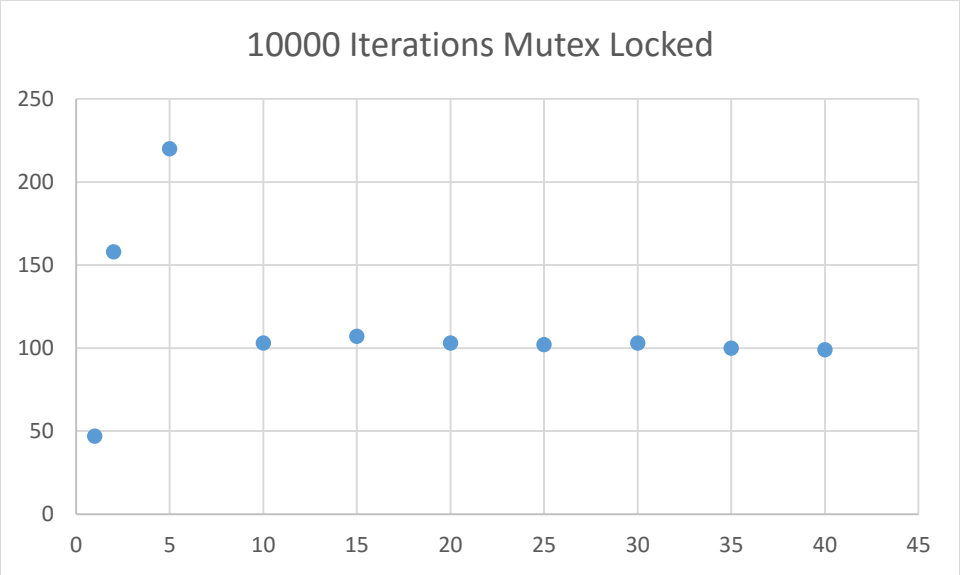

Figure 4: Average cost in nanoseconds graphed against the number of threads with an add function (mutex_add) protected by a mutex lock. The number of iterations used for each data point is a constant 10,000.

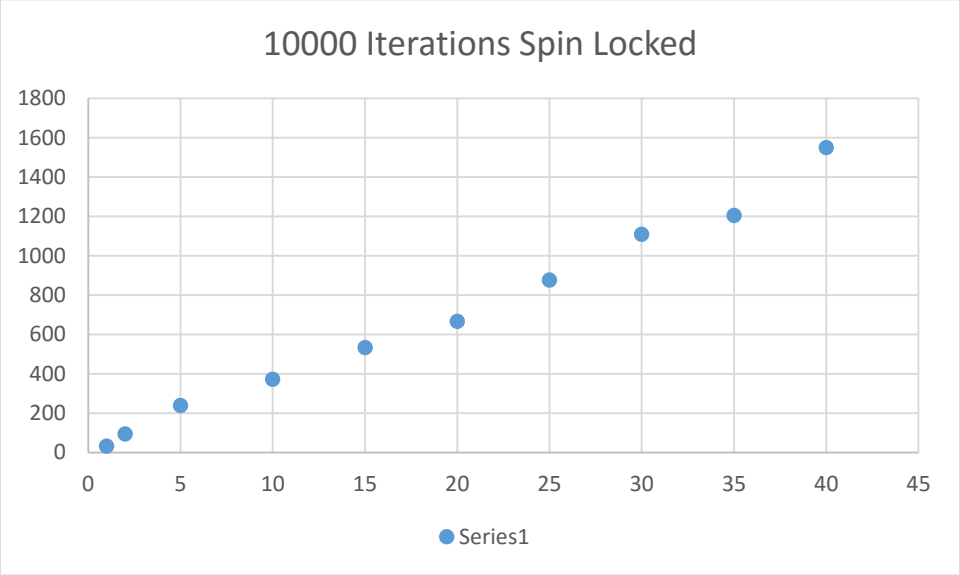

Figure 5: Average cost in nanoseconds graphed against the number of threads with an add function (spin_add) protected by a spin lock. The number of iterations used for each data point is a constant 10,000.

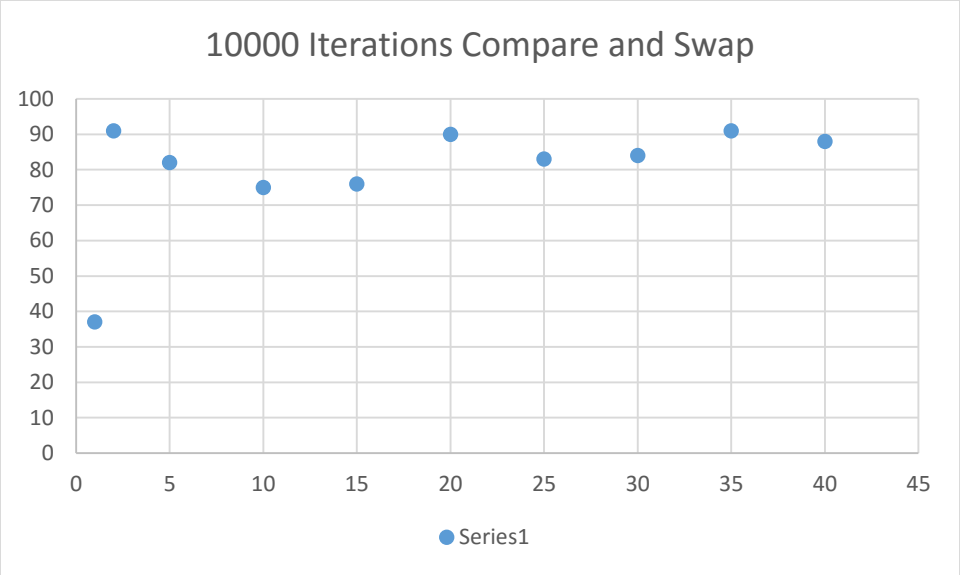

Figure 6: Average cost in nanoseconds graphed against the number of threads with an add function (swap_add) protected by a compare and swap. The number of iterations used for each data point is a constant 10,000.

1.3

1.

The reason why they perform similarly for low numbers of threads is because each thread spends relatively little time actually locked and not performing any operations. The three protected ones are similar (though still slightly worse) because differences in overhead for lock maintenance is relatively insignificant at this point. However, there IS overhead so that's why they are all worse than the unprotected one, which just charges forward recklessly.

2.

As the number of threads goes up, the bottleneck effectively worsens. When we lock a particular section of code, if it is only shared among 5 threads then effectively 4 of them are idle at any time, which is not so bad. If we have 40 threads then 39 of them are effectively idle at any given time. This idleness should take its toll as the number of threads increases.

3.

In contrast with a mutex lock or compare and swap approach, a spin lock doesn't actually block the thread. Rather, it has it keep trying to lock the critical section and failing. In a sense, these threads are always awake, and thus are always running and straining the processor. Mutexes actually perform blocking which allows the threads to sleep. For low numbers of threads this additional overhead might translate to a worse payout (the overhead may be less than the effort saved by that thread sleeping) but this number improves greatly as the number of threads increases (a lot of threads will be spending a majority of their time sleeping). However, this is only accounting for the expenses of the processor. The

data that we obtained showed the the average runtime for spinlocking is actually getting worse and worse, because the total time elapsed is increasing. This may be because the inefficiency described above is actually causing a slowdown in the computation (less resources available or something like that). All in all, I suspect that the data trends obtained for this section are not quite as expected but after repeated examinations of the code I cannot determine why. Everything looks correct, so I must respect the data.

Part 2

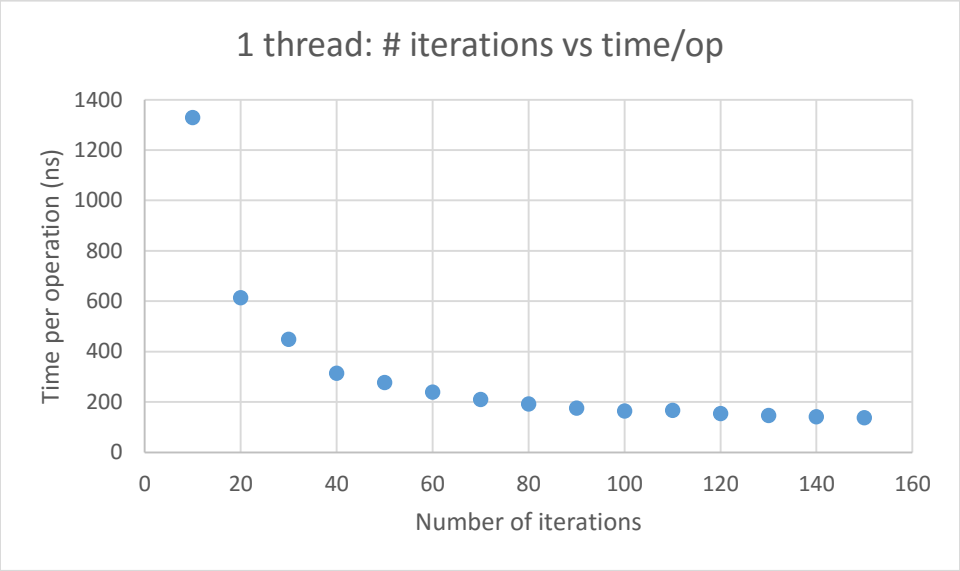

Figure 7: Graph of the number of iterations versus the time per operation for one thread, running unprotected

2.1

To correct for the variation in time per operation vs. the number of iterations, we need to increase the minimum number of iterations such that the processor "warms up" to the code being executed (branch predictor, cache, etc.). Additionally, increased iteration count masks the fixed overhead of thread creation.

4 threads and 50 iterations fairly consistently demonstrates a problem.

With yield = i, 4 threads and 15 iterations.

With yield = d, 4 threads and 12 iterations.

With yield = is, 4 threads and 10 iterations.

With yield = ds, 4 threads and 8 iterations.

With 4 threads and 150 iterations and sync=m, none of the yield options {i,d,is,ds} fail.

With 8 threads and 1500 iterations and sync=s, none of the yield options {i,d,is,ds} fail.

With 8 threads and 1500 iterations and sync=m, none of the yield options {i,d,is,ds} fail.

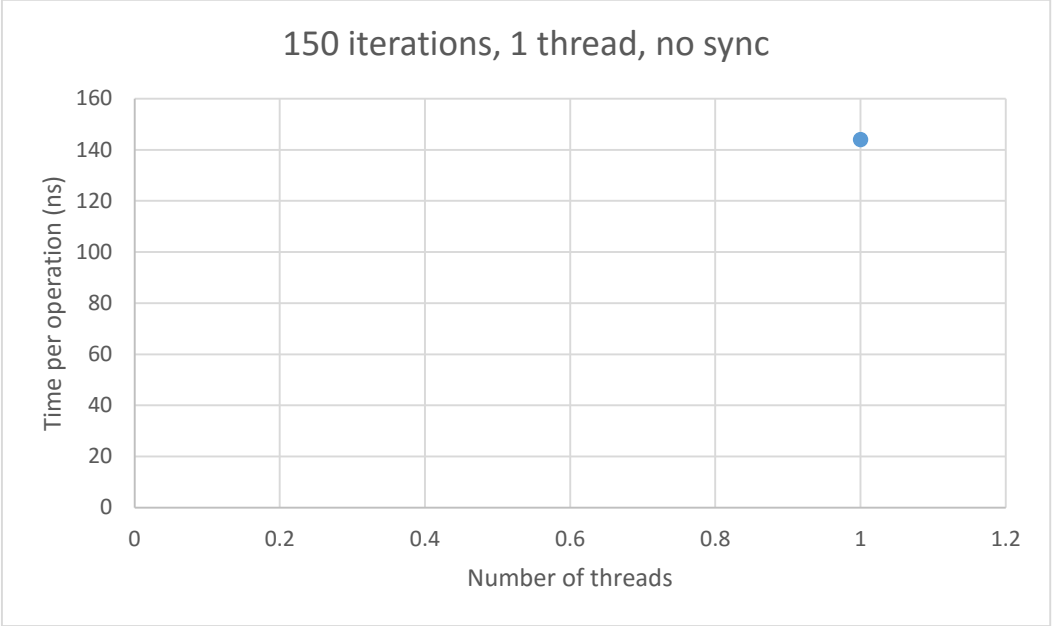

Figure 8: Graph of the number of threads versus the time per operation for 150 iterations with one thread, running unprotected

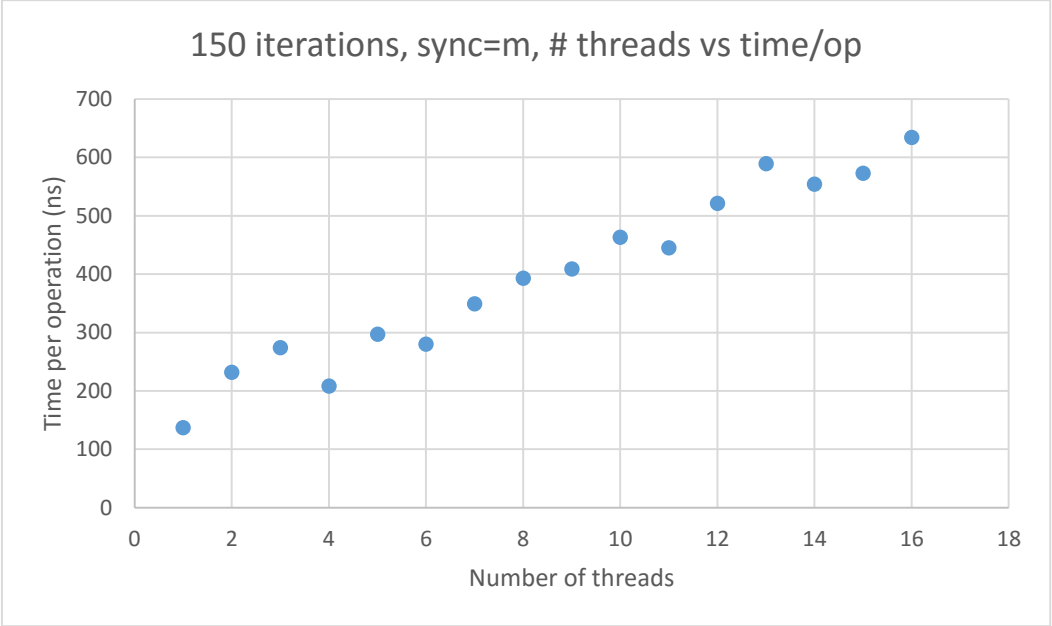

Figure 9: Graph of the number of threads versus the time per operation for 150 iterations using mutexes

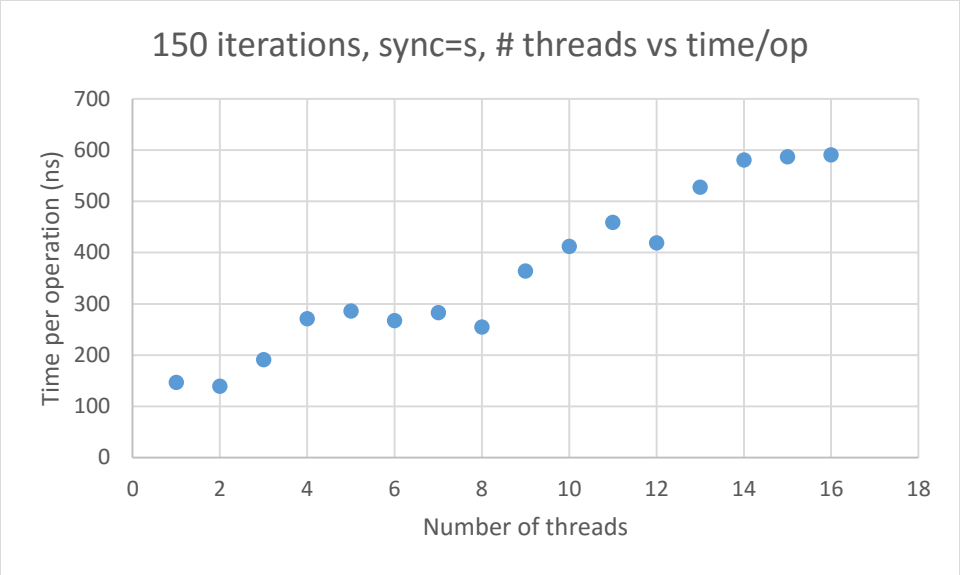

Figure 10: Graph of the number of threads versus the time per operation for 150 iterations using spinlocks

2.2

The critical sections of part 2 are a lot longer than the critical sections of part 1, which increases the chances of it being interrupted by a context switch. We run on the SEASNET server, which has other people demanding CPU time. An increased amount of context switches as a result of running more threads on more cores contributes to this somewhat unexpected pattern.

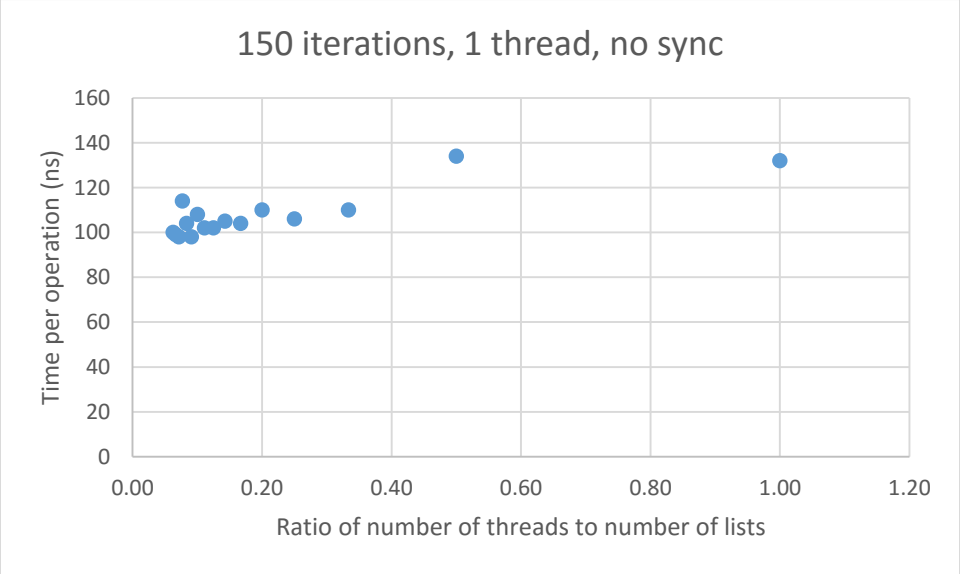

Figure 11: Graph of the ratio of threads to lists versus the time per operation for one thread with no protection

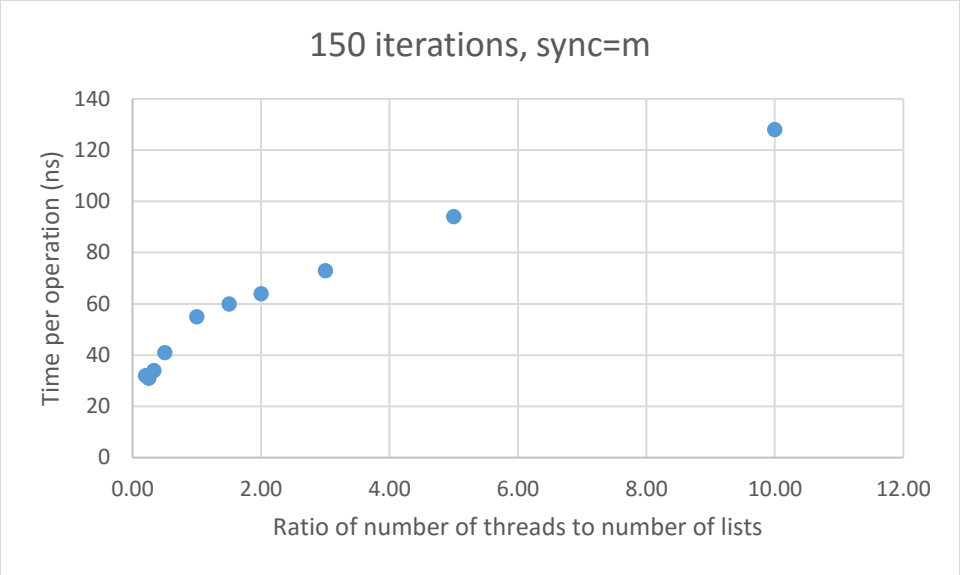

Figure 12: Graph of the ratio of threads to lists versus the time per operation using mutexes

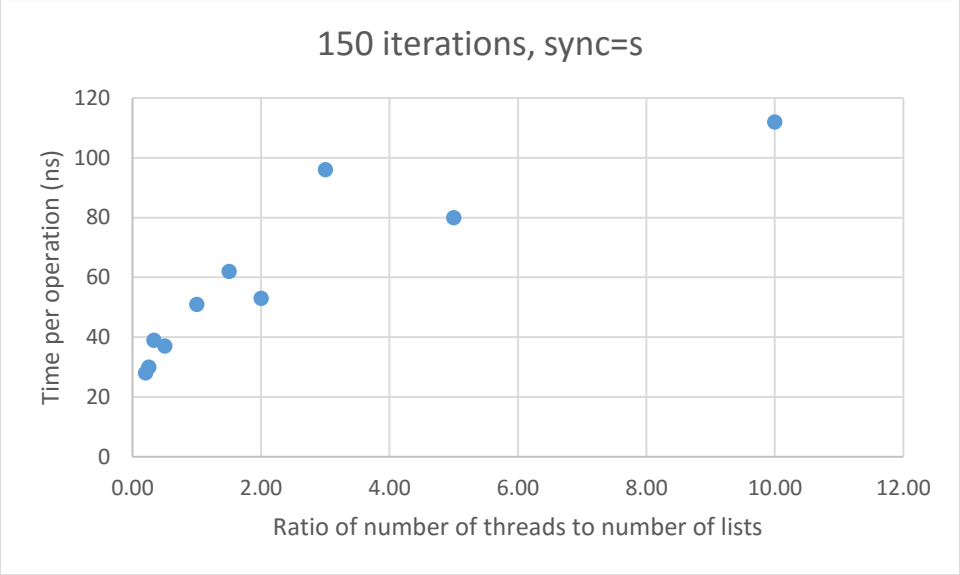

Figure 13: Graph of the ratio of threads to lists versus the time per operation using spinlocks

2.3

The change in performance of the synchronized methods as a function of the number of threads per list appears to be linear and negative. As the number of threads per list increases, the performance of the synchronized methods decreases.

Threads per list is a more interesting number than threads for this particular measurement because introducing sublists gives us the ability to measure a more "parallelized" workload. With multiple lists with their own locks, we decrease the amount of starvation for each thread. With more threads per list, the amount of starvation in the system will increase, and that is what the graphs above demonstrate. As

a result of increased starvation, the time per operation increases, as more threads wait around doing nothing.

Part 3

3-1.

1. pthread_cond_wait releases the mutex, so it must be held before it can be released. The idea here is that the lock is released so that another thread can acquire the lock and process some data, then signal the condition variable and release the mutex so that the original thread can then use the processed data or whatever that was a result of the signalling thread.

2. The mutex must be released when the waiting thread is blocked, otherwise there will be deadlock if other threads attempt to acquire the mutex to perform processing that affects the condition variable.

3. The mutex must be reacquired when the calling thread resumes in order to guarantee that the condition variable is not accessed by any other threads and to guarantee that nothing is accessing the data that the thread that called pthread_cond_wait uses.

4. If pthread_cond_wait does not release the mutex, a third thread could modify the condition that the while loop surrounding the pthread_cond_wait relies on or perform some other actions before the first thread is supposed to block. This would cause race conditions and unintended behavior because the original thread might not block or the unexpected thread might modify data in ways the original thread does not expect.

5. No, this must be implemented by a system call because in order to atomically release the mutex and cause the calling thread to block on the condition variable, we need guarantees that no other thread is accessing the mutex and condition variable at the same time. We need kernel mode privileges to guarantee these conditions.