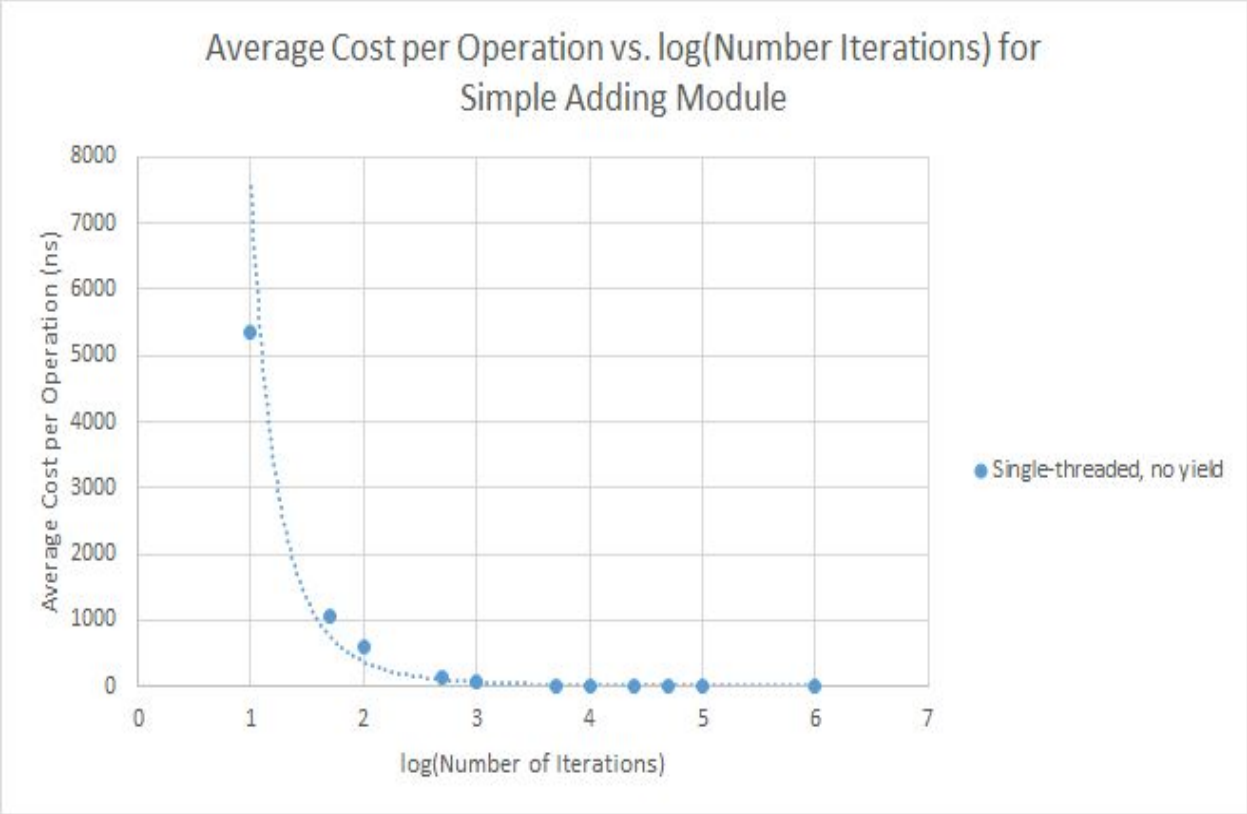
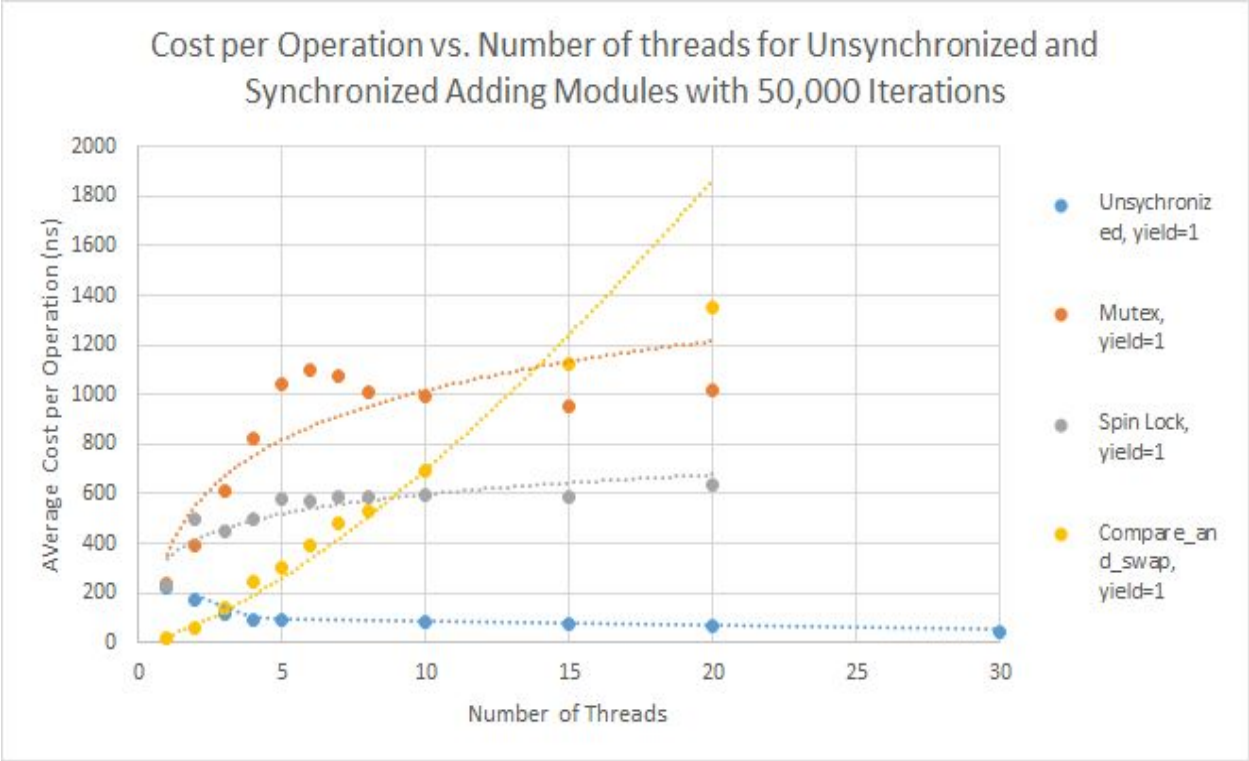
Anderson Huang
CS111
Winter 2016

Synchronization Lab Data

Part 1: Parallel Updates to a Shared Variable

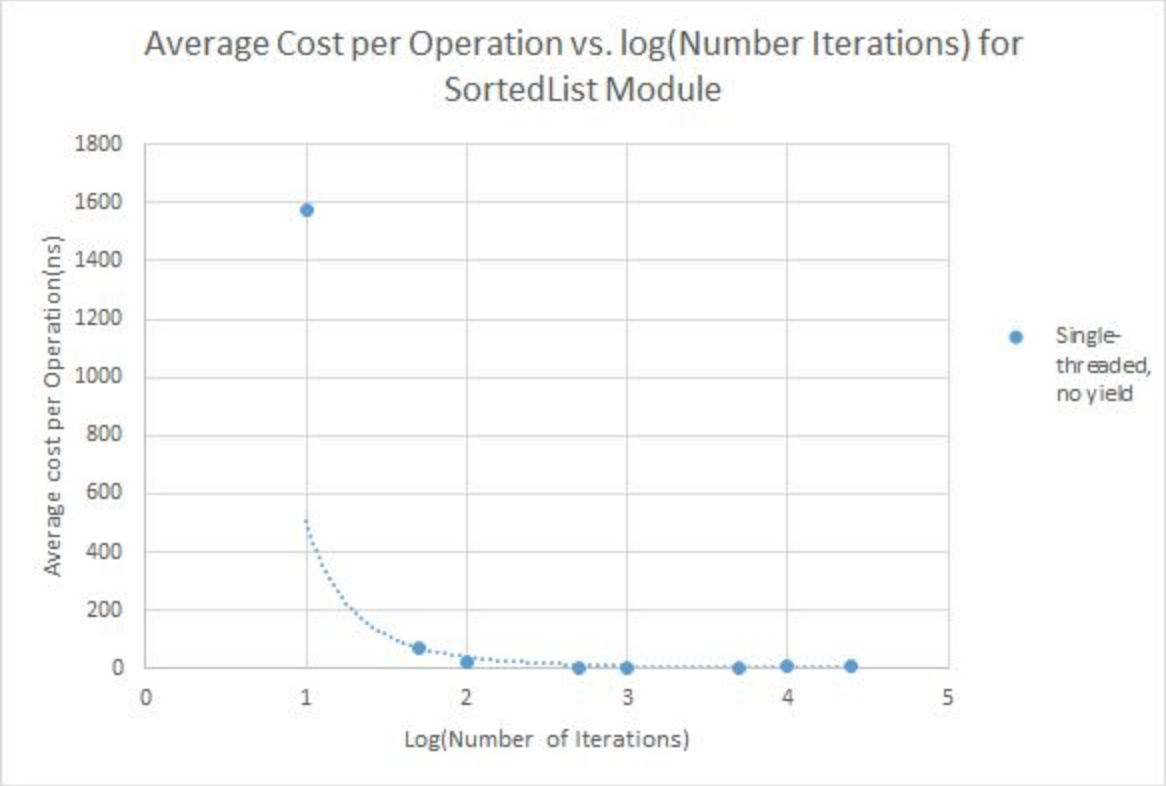


The average cost per operation for a single-threaded implementation of the adding test driver shows that with an increase in the number of iterations, the average cost to perform an operation drastically reduces. This is because creation of a thread takes much longer than executing a simple add instruction.

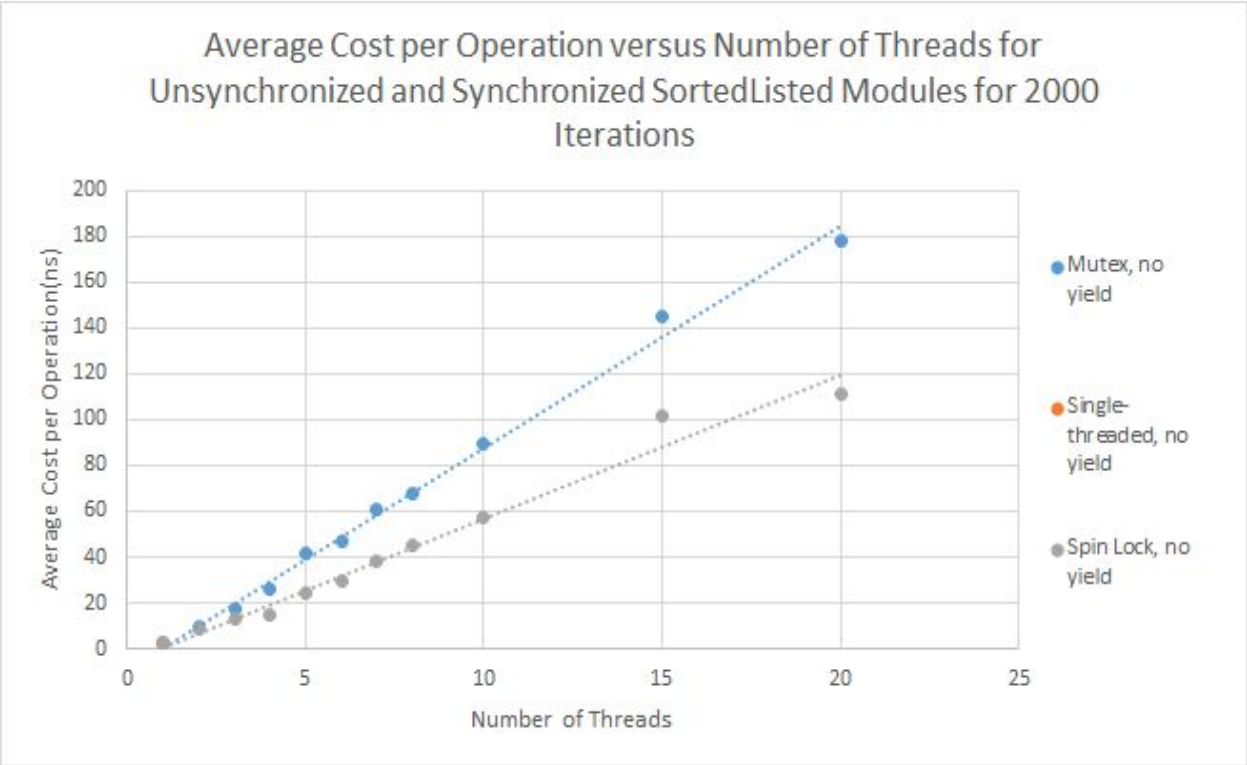


The average cost per operation for unsynchronized and synchronized adding functions are shown above. Notice how the unsynchronized version has the best performance; however, it also yields incorrect results due to race conditions. The tradeoff for correctness is performance, as shown by the three synchronized methods in yellow, gray, and orange. The method using sync_val_compare_and_swap gives the worst overall performance as one extrapolates data. Mutexes and spin locks are comparable. This graph gives the real-time of operations, not the CPU time. If a graph of CPU time were shown, the CPU time taken by the spin locks would be much greater than mutexes, because they busy wait as they continually poll to see if the lock has been unlocked. The general trend is that with an increasing number of threads, the average cost goes up. This makes sense because more race conditions spawn, as only one shared variable is available between however many threads that exist.

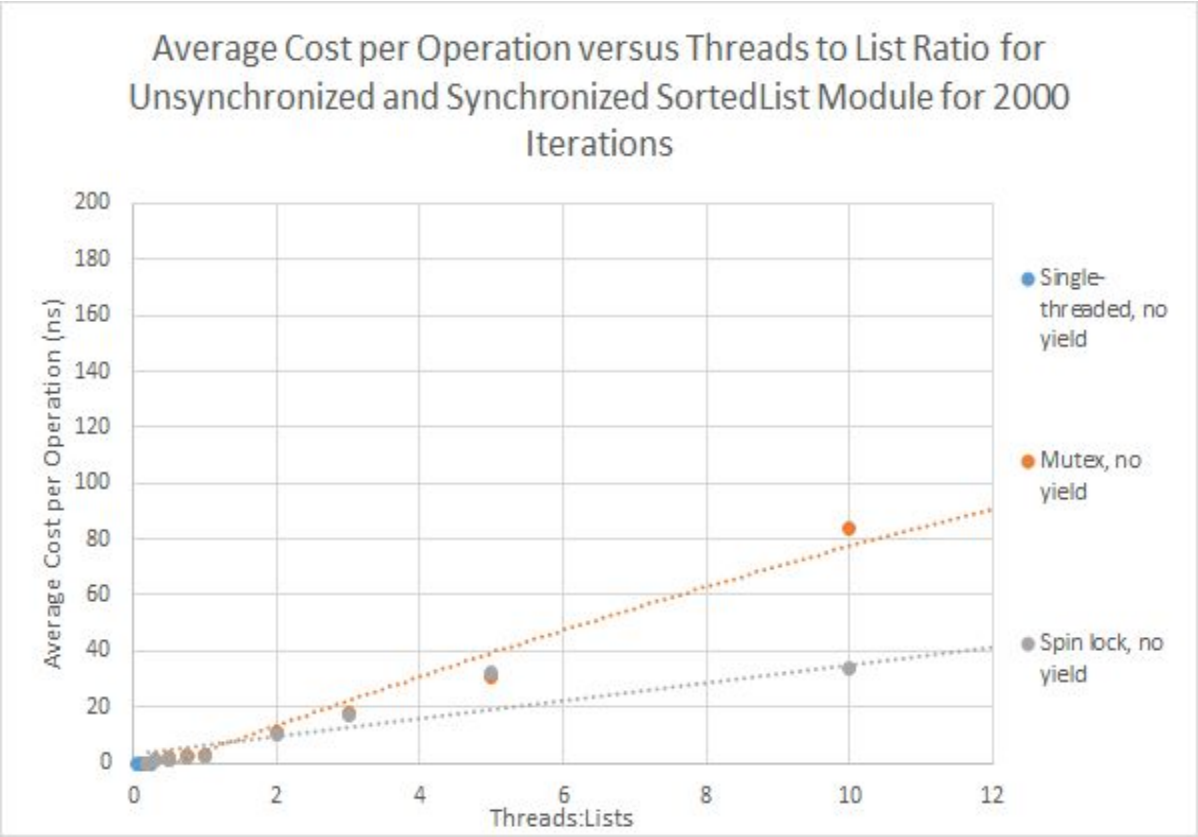
Part 2: Parallel Updates to a Complex Data Structures (Sorted List)



This graph reiterates the fact that with an increasing number of iterations in a single-threaded application, the average cost drops dramatically as iterations go up. The creation of a thread takes a decent amount of time; to neutralize this, the number of iterations must be great enough to exceed this creation time.



The single-threaded version can only run correctly with one thread. Therefore, its graph consists of only one point, which is unfortunately covered by the mutex and spin-lock graph points. Again, the trend is that as threads increase, race conditions become prevalent as there is only one list to work with. These race conditions result in a noticeable increase in the average cost per operation.



This graph shows that parallelism works best when threads are not trampling over each other's memory space. When lists outnumber threads, the average cost per operation is very, very low, and performance, as a result, is very, very good. This is because as lists exceed threads, there is a very high chance that when a thread must operate on a sorted list, it will not conflict with another thread trying to write to a list. When multiple locks are used, and threads can do useful work without having to context switch or wait, this results in good speed.