1. The three threads that are executing concurrently account for the inconsistency of the final value of the *count* variable compared to the sum of the local variables. Given that these threads are running concurrently, it is possible for one thread (T1) to begin, pull the current value of *count*, then another thread (T2) begin and iterate the *count* variable before T1 has finished executing. T1 will then iterate the original count variable, not the new count variable that had been updated by T2. Over the course of millions of executions, this will create a large inconsistency in the final value of count.
2. For smaller loop bounds, there are often no inconsistencies in the final count value, because, given the fewer accesses to the critical region, the likelihood of two threads accessing it at the same time goes down significantly, and the risk becomes minimal at very small sizes.
3. The local variables are always consistent, because this data is not shared between multiple threads, but by a single thread. With only one access point, the information cannot become corrupted.
4. My solution ensures the final value of count will always be consistent, because it bounds the critical region with a lock/unlock. The lock prevents other concurrent threads from accessing the data that is inside the lock, while the unlock tells them that that particular thread is finished and other threads may now access the shared data. By only allowing one thread to access the critical region at a time, I have prevented corruption of the shared data.
5. The times for the two versions of the program are different, because the mutex solution is expensive for a simple add operation. The atomic solution uses a GCC specific built-in function that was built specifically to increment values.