1. What accounts for the inconsistency of the final value of the `count` variable compared to the sum of the local counts for each thread in the version of your program that has no lock/unlock calls?

The time it takes for the CPU to execute a load, add, and store command for that particular variable means that multiple threads can load that variable at the same time, and once each store is called, it will overwrite the value.

2. If you test the version of your program that has no lock/unlock operations with a smaller loop bound, there is often no inconsistency in the final value of `count` compared to when you use a larger loop bound. Why?

I don't know, computer performance?

3. Why are the local variables that are printed out always consistent?

Because these variables are only accessed by one thread each and are not shared among multiple threads, so there is no critical section to worry about.

4. How does your solution ensure the final value of `count` will always be consistent (with any loop bound and increment values)?

By locking the "critical section", where a thread is incrementing count, we force the other threads to wait for the mutex to unlock before it can access the count variable.

5. **Consider the two versions of your ptcount.c code. One with the lock and unlock operations, and one without. Run both with a loop count of 1 million, using the time time command: "bash> time ./ptcount 1000000 1". Real time is total time, User time is time spent in User Mode. SYS time is time spent in OS mode. User and SYS time will not add up to Real for various reasons that need not concern you at this time. Why do you think the times for the two versions of the program are so different?**

```
$ time ./ptcount_mutex 1000000 1
Thread: 2 finished. Counted: 100000
Thread: 1 finished. Counted: 100000
Thread: 0 finished. Counted: 100000

real    0m0.021s
user    0m0.036s
sys     0m0.012s
```

```
$ time ./ptcount_atomic 1000000 1
Thread: 1 finished. Counted: 100000
Thread: 0 finished. Counted: 100000
Thread: 2 finished. Counted: 100000

real    0m0.006s
user    0m0.008s
sys     0m0.000s
```

My assumption is that the mutex locks causes the threads to wait, meaning that it takes longer to complete the task. Because the main program has to wait for all the threads to complete, it takes longer overall to finish the program. The atomic add, however, is a single command that is executed without creating a wait state for other threads, and finishes faster.