<u>Question 2: Written Question Answers:</u>

<u>Note: I utilized the 'real' timing results using the time command.</u>

Table 1: Timing Results for medium.txt

#Threads	Observed Timing	Observed Speedup	Expected Speedup
Original Program	0m30.546s	1.0	1.0
1	0m31.466s	0.9707	1.0
2	0m15.828s	1.92987	2.0
3	0m10.499s	2.9094199	3.0
4	0m7.969s	3.83267	4.0
8	0m5.484s	5.57002	8.0
16	0m4.667s	6.545	16.0

Table 2: Timing Results for hard.txt

#Threads	Observed Timing	Observed Speedup	Expected Speedup
Original Program	0m10.338s	1.0	1.0
1	0m10.466s	0.9877699	1.0
2	0m5.380s	1.9215613	2.0
3	0m3.561s	2.903117102	3.0
4	0m2.759s	3.747	4.0
8	0m1.679s	6.15723	8.0
16	0m1.556s	6.66439	16.0

Table 3: Timing Results for hard2.txt

#Threads	Observed Timing	Observed Speedup	Expected Speedup
Original Program	0m10.270s	1.0	1.0
1	0m10.535s	0.9748	1.0
2	0m5.332s	1.92610	2.0
3	0m3.560s	2.8848	3.0
4	0m2.765s	3.726415	4.0
8	0m1.621s	6.33559	8.0
16	0m1.601s	6.41474	16.0

"Once you have created the tables, explain the results you obtained. Are the timings what you expected them to be? If not, explain why they differ."

In all cases, up to N = 4 threads, the results are approximately about what I expected. The threaded solution is more complex, and involves more things which need to be started up (mutexes, barriers, etc), even with just one thread, so it's completely expected that at 1 thread, the threaded solution still takes marginally more time than the pure no-threading original solution. For a similar reason, a pure 1/N speedup is not expected, given that the threaded solution inherently requires more steps per computation than the simple single-threaded solution.

What is not expected in all cases is the drop to only 6x speedup at 8 threads. This UCalgary CPSC Linux server has 8 CPUs, so the expectation is that with 8 threads, it should perform roughly 8 times faster. Instead, we are getting only slightly higher than 6x speed up, consistently. Furthermore, rerunning the 8/16 CPU tests, I found the results can vary significantly, sometimes varying by as much as 2 seconds more or 0.5 second less!

There are many reasons for this. As mentioned before, there is a certain amount of overhead for each thread, and as the number of threads increases, the amount of overhead increases. This reduces the performance gain per thread; from 1 to 4, we already lost .3 seconds; extending that forth to 8 threads, it could easily explain the roughly 0.5 extra seconds it took, which cut its efficiency down from 8x to just 6x faster. As the n-times speed up increase, the Nx speedup becomes more sensitive to fluctuations in performance.

Furthermore, it's possible that as currently implemented, my program needs to wait on the CPU deciding to switch the relevant threads to active in order to hit specific points with the spinlocks, and to do that for all threads, before continuing to the next stage. It would be more efficient if I could tell my program to yield their thread there, rather than spin until the CPU tires of it. Unfortunately, we did not learn anything like pthread_yield() or something.

Finally, use of mutexes corresponds to a system call (futex), which, along with the waiting for thread switching, means that this program is vulnerable to other loads on the system. Running the same program with the exact same configuration periodically throughout the day, I found that at some hours, it slowed down 1.5x times slower, and other times it ran at what appears to be full speed.

Another unexpected thing is that 16 threads is often slightly faster than 8 threads, despite the main server only having 8 threads. I suspect this is due to the nature of how the program was parallelized, as instructed by the assignment (third solution method), TA and professor. Suppose that for a number A, its smallest factor is B. Depending on how the range is divided up, B could be near the beginning of a range, or near the end of a range. With more threads, it is likely that it is near the beginning of a thread's range.

Appendix:

```
Last login: Fri Jun  5 12:26:03 2020 from 136.159.5.13
john.ngo@csx:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           8
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               44
Model name:          Intel(R) Xeon(R) CPU           X5650  @ 2.67GHz
Stepping:            2
CPU MHz:             2660.000
BogoMIPS:            5320.00
Hypervisor vendor:   VMware
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            12288K
NUMA node0 CPU(s):   0-7
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cm
ov pat pse36 clflush dts mmx fxsr sse sse2 ss syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts nopl xtopology tsc_reliable nonstop_tsc cpuid aperfmperf p
ni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes hyper
visor lahf_lm pti tsc_adjust dtherm ida arat
john.ngo@csx:~$
```