HW3 - Multi-threads Prime Number

A description of the system where the programs were executed: Processor, RAM, and Operative System:

Processor (CPU): Tested on both a single-core and a quad-core (4-core) machine running Ubuntu.
RAM: 2048MB (or 2GB)
OS: Linux (Ubuntu)

A comparative table showing, on the search up to 10^7, the execution times of the single-thread version, and the multi-thread with 1, 2, 4, 6, and 8 threads.

Range (2 - 10^7) with **4 processors**

Thread#	Time Taken (seconds)
sequential	**37.39257230**
Thread #: 1	39.88526408
Thread #: 2	18.74999481
Thread #: 4	9.05900717
Thread #: 6	9.33083177
Thread #: 8	9.37050146

Single processors result
(only for our practice)

	Range (2 - 10^7)
Thread #: 1	37.10459694
Thread #: 2	35.11473377
Thread #: 4	35.57618300
Thread #: 6	34.66201290
Thread #: 8	36.38487728

An analysis of the performance of the programs with emphasis on correctness and speedup.

1. Sequential vs. Multi-threaded Execution:
The sequential version takes 37.39 seconds, whereas using just a single thread takes slightly longer at 39.88 seconds. This suggests some overhead in the parallelized version, even with just one thread.

2. Speedup with Additional Threads:
Introducing additional threads results in a noticeable speedup:

- 2 threads reduce time to 18.75, nearly halving the sequential time.
- 4 threads further reduce it to 9.05, showcasing a linear improvement.
- The speedup with 2 and 4 threads reflects the advantages of parallel processing, especially given a 4-core processor.

3. Diminishing Returns Beyond 4 Threads:
Increasing the thread count beyond 4 (to 6 or 8 threads) yields marginal improvements, with times of 9.33 and 9.37 respectively. The almost stagnant performance indicates heightened contention for shared resources, particularly the mutex lock guarding the shared count variable. As more threads try to update the count, they frequently stall, waiting for access.

4. Correctness:

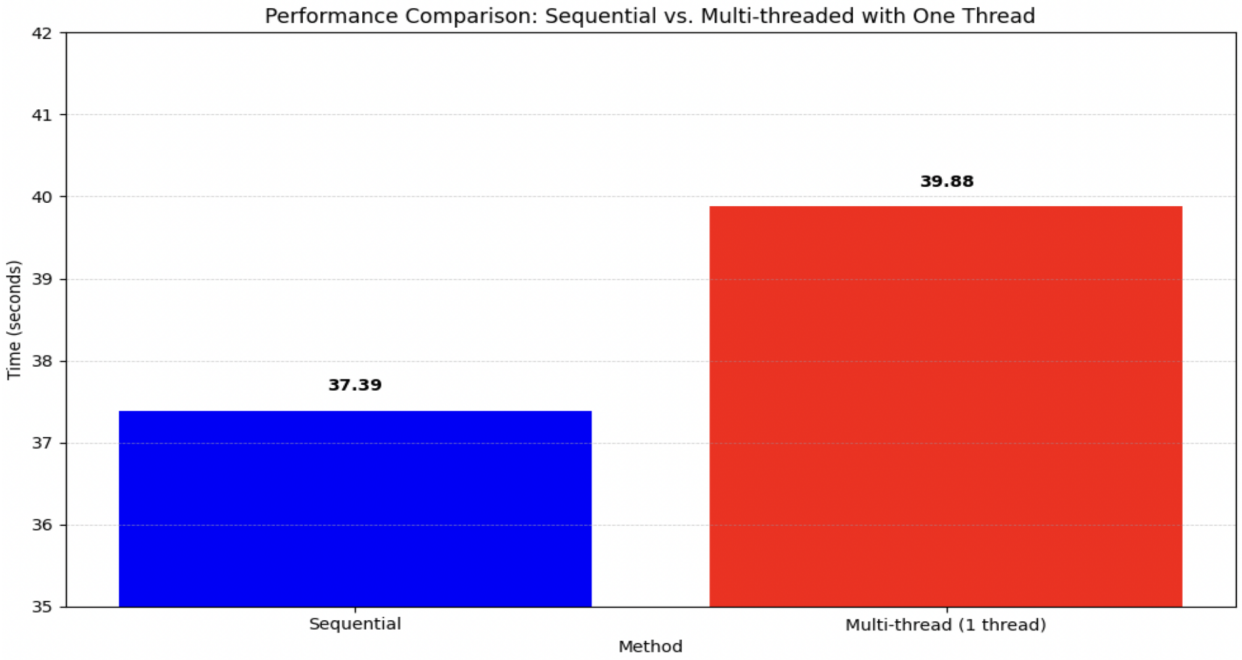
The code ensures that no two threads can update the shared count simultaneously. Hence maintaining the integrity of the data. However, the need to frequently access and update the count is a double-edged sword: while it ensures correct behavior, it also increases contention and thus potentially hinders speedup.

Conclusion:

The program effectively utilizes parallel processing to achieve a speedup, especially evident with up to 4 threads on a 4-core system. However, due to increased contention for shared data, adding more threads beyond the number of cores does not yield significant further improvements.

An analysis of the sequential version versus the multi-thread version with only one thread. Is there a difference? How do you explain the results? Add plots and figures as necessary to explain your results.

1. Results Overview:
   - Sequential Version: 37.39 second
   - Multi-threaded Version with One Thread: 39.88 second
2. Performance Difference:
   - There is a difference of 2.49 seconds, with the multi-threaded version with one thread being slower. This is surprising since one would expect a multi-threaded version with just one thread to closely mirror the performance of the sequential version, if not be identical.
3. The overhead in the multi-threaded version can be attributed to the following:
   - Initialization of Thread Resources: Even though there's only one thread, the system still needs to allocate resources for thread creation, which is absent in the sequential version.
   - Mutex Overhead: The use of mutex locks in the multi-threaded version introduces an additional overhead. Even with a single thread, the act of acquiring and releasing a lock incurs a performance cost.
   - Shared Data Management: The multi-threaded version is designed to handle shared data, such as the shared count. Even with one thread, it still goes through the motions of managing this data as if multiple threads were accessing it, adding slight inefficiencies.



What happens if Simultaneous Multi- Threading (or "Hyper-Threading") is enabled or disabled? You may find the article in logicalincrements.com useful.

Simultaneous Multi-Threading (SMT), known as Hyper-Threading in Intel CPUs, allows one physical core to act as two logical cores. Here are the effects of enabling or disabling it:

Enabled:

- Performance Boost: Can enhance performance for multi-threaded applications.
- Increased Heat & Power: More active threads can lead to higher power consumption and heat.
- Security Concerns: Some vulnerabilities, like Spectre, can exploit SMT, potentially risking data leaks.

Disabled:

- Stable Performance: Beneficial for tasks requiring consistent execution or applications not optimized for multi-threading.
- Reduced Heat & Power: Less power usage and heat generation.
- Better Security: Reduced risk from certain side-channel attacks.

In summary, using SMT can be beneficial for general multitasking, but for specific tasks like gaming or security-sensitive operations, disabling it might be preferable.