Homework 3: Prime
CS231P
4/25/2025
TEAM: Destin Wong 64848542, Midhuna Mohanraj 39922268, Eric Huang 59504944

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

System:
- Processor: Apple M4 Pro
- RAM: 24GB
- Operating System: MacOS Sequoia 15.3.2

2. 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, 8, 16, 32 and 64 threads.

Number of Threads

	Single-thread	Multi-thread with 1 thread	Multi-thread with 2 threads	Multi-thread with 4 threads	Multi-thread with 8 threads	Multi-thread with 16 threads	Multi-thread with 32 threads	Multi-thread with 64 threads
Execution Time	2.15 s	2.05 s	1.855 s	2.32 s	2.27 s	2.49 s	2.47 s	2.49 s
Speedup	0%	4.87%	15.9%	-7.3%	-5%	-14%	-13%	-14%

Times were calculated by averaging the execution times

3. An analysis of the performance of the programs with emphasis on correctness and speedup.
We noticed that compared to the single-thread version, there was a speedup when we switched to multi-threads with 1 and 2 threads. The highest speedup occurred when we ran the multi-thread with 2 threads. When more than 2 threads were added, however, the execution time of the multi-threaded versions became slower than the single-thread execution time. With 4 threads and 8 threads, the execution time became slower than with two threads, and the execution time took the longest when we used 16 and 64

threads. Using 32 threads also ended up with an execution time similar to the 16 and 64 threads.

The trend of the execution time decreasing initially, then increasing with the number of threads can be explained by multiple factors. The first factor is the wait time caused by the threads needing to access the shared counter. Every thread will need to lock the mutex, read and increment the counter, and unlock the mutex before they can continue their task. Having more threads leads to more threads idling and waiting for the shared resource. Eventually, with too many threads, the threads could spend more time waiting for other threads to unlock the mutex. Adding too many threads can also lead to increased overhead, which can slow down the program.

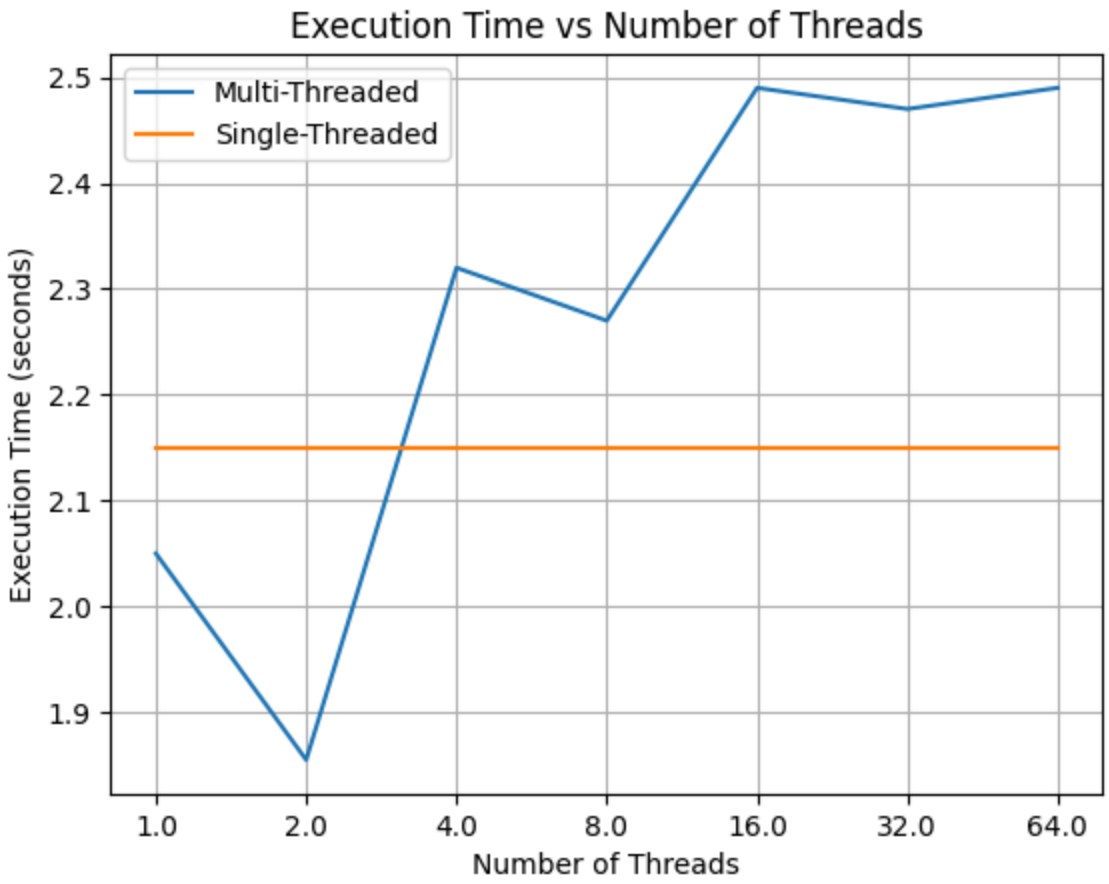
Another factor is the overhead required for the CPU to switch between threads. In order to maintain concurrency, the CPU must switch between threads, which can add up quickly when we make a lot of threads. That overhead won't be much if it's just 1 or 2 threads, but after that the overhead time outweighs the performance benefits from concurrency.[1]

4. An analysis of the single-thread 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.

The average runtime for the multi-thread version with one thread consistently runs about 5% faster than the single-thread version. We get the same results running on every team member's device. The single-threaded function to calculate the primes and the worker thread function are very similar, with a few differences. The single-thread function runs on nested for loops, which go over all the numbers in the range and calculate if they're prime. The worker thread function iterates by running an infinite while loop that accesses the mutex, increments it, calculates if it's prime, and checks for the termination condition. While in the end the function is identical, the multi-thread version's way of looping may be slightly more efficient.

Another possibility is the difference between how the main thread and how a pthread thread function executes. While the main thread runs just fine, in the makefile is the argument "-pthread", which can make our program more optimized to do multithreading than single-threading.

[1] https://thelazyadmin.blog/threads-vs-performance



What happens if Simultaneous Multi-Threading is enabled or disabled?

When Simultaneous Multi-Threading is enabled, the operating system will see a single physical CPU core as two virtual (logical) CPU cores, allowing certain operations to be broken down into smaller parts to be processed more efficiently by the two virtual CPU cores. This effectively allows us to run two threads at once, without context switching (if it works as advertised). As long as the core is not using the same data for each task, the information can be split up. But if each core requires the same data, then it will still have to deal with information synchronization, like we have done for our program's shared counter. If the code is optimized for SMT, we would see a large performance benefit, but without optimizing, there would be little gain or even worse performance. If our single-threaded program is run with SMT, there shouldn't be any performance gain, as the program isn't optimized for parallel processing, it always has to go through the for loop. If run on our multi-threaded program, it would have a bit of an impact, as more threads can do their prime calculation at the same time, but they're still bottlenecked by being able to access the mutex in order to do their work.[2]

[2] https://blog.logicalincrements.com/2017/10/what-is-hyper-threading-simultaneous-multithreading/