System & Networks I:
Project 2 User Manual

Experiment Description

The goal of this experiment was to measure how multithreading and locking affect the performance of a Collatz stopping-time calculator. The program computes the Collatz stopping time for every integer from 1 up to a chosen value of N, using T parallel threads. Two modes were tested:

- Locked mode – uses a mutex to synchronize shared data safely.
- No-lock mode – removes synchronization to test performance without safety.

Machine

The experiment was conducted on a laptop with the following specifications:

- Processor: 13th Gen Intel Core i5-13420H
- Cores: 8 physical cores
- Threads: 12 logical processors (hyper-threaded)
- Memory: 16 GB RAM (15.7 GB available)
- Operating System: Windows 11 Home (64-bit)

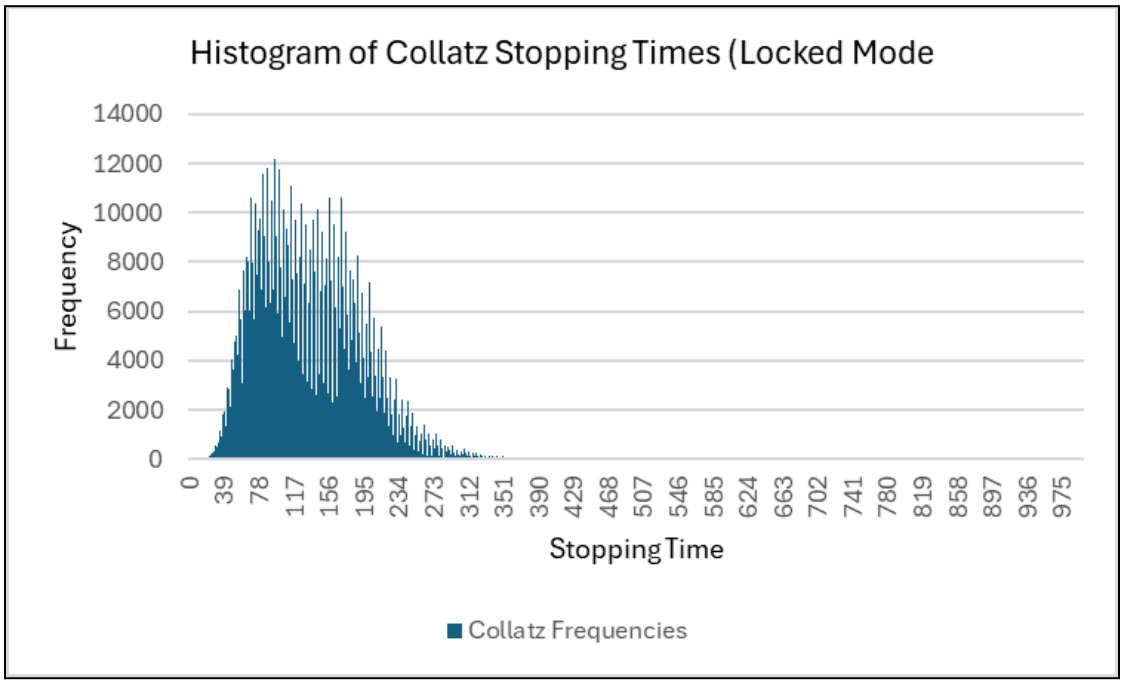
The system's 8 physical cores allow testing up to 8 parallel threads (T = 1..8). Using more than 8 threads would not significantly improve performance since the physical cores would have to share processing time.

Methodology

To ensure accurate and stable measurements:

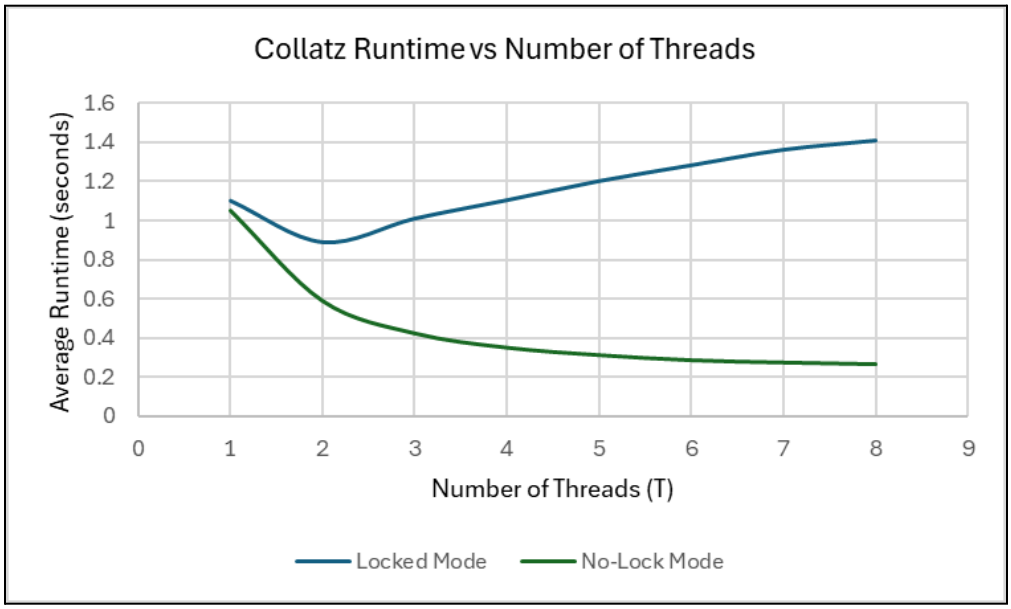
- A large N value was chosen (such as N = 1,000,000) to make runtime measurable.
- For each thread count T = 1, 2, 3, 4, 5, 6, 7, 8, the program was executed 10 times in both locked and no-lock modes.
- The runtime (in seconds) was captured from stderr and averaged over the 10 runs.
- The average of the runtimes was recorded in CSV files for analysis.
- The histogram of Collatz stopping times was generated once using a fixed N in locked mode to verify accuracy.

Histogram Results



In this Histogram, each bar shows how many numbers (from 1 to N) took a certain number of steps to reach 1 in the Collatz sequence. The tall section near the left means most numbers take relatively few steps, while the long tail to the right shows a few numbers take much longer.

Time Experiment Results & Analysis



The results of the experiment show how the runtime of the Collatz stopping-time calculator changes as the number of threads increases.

Parallel Execution Performance:

When increasing the number of threads (T), both versions of the program initially benefit from parallel execution. With a large value of N, the total workload can be split among multiple threads, which reduces the time each thread spends on computation. For small values of T (up to around 4), runtime improves noticeably because all CPU cores are being utilized efficiently. However, beyond that point, performance gains level off as thread management and synchronization overhead begin to dominate.

Impact of Locks on Performance:

In locked mode, a mutex ensures that threads safely update shared data. This makes the program correct and reproducible but slower, since threads often wait for access to the shared resource. The average runtime flattens or even increases after four or more threads because the cost of acquiring and releasing locks outweighs the benefits of parallelism.

In contrast, the no-lock mode removes synchronization overhead, allowing threads to run freely and complete their computations faster. This mode demonstrates the maximum possible parallel performance but produces incorrect results due to race conditions. The difference in speed between locked and no-lock runs directly represents the time cost of synchronization.

Overall, parallel execution improves performance up to a point, but proper synchronization (locking) introduces necessary delays to preserve correctness. The experiment highlights the classic trade-off between speed and accuracy in multithreaded programming.

Conclusions

This command demonstrates exactly the kind of result that's expected for comparing locked vs. no-lock runs with N = 1000000 and T = 4:

```
echo "locked  sum = $(awk -F, '{s+=$2} END{print s}' histogram_locked.csv)"
echo "nolock  sum = $(awk -F, '{s+=$2} END{print s}' histogram_nolock.csv)"
```

These are the results we got from that command:

```
locked  sum = 1000000
```

```
nolock  sum = 1078750
```

The locked sum being 1000000 is exactly as expected, since it matches N = 1,000,000. That means every number from 1 through 1,000,000 was processed exactly once, no duplicates or misses. The unlocked sum being higher than N by 78,750 counts is also correct, since that means race conditions occurred:

- Multiple threads sometimes incremented the same histogram bins at the same time, causing counts to be added incorrectly.
- Also possible: some increments happened before the memory was synchronized, so one update overwrote another.

This is exactly what the experiment is meant to demonstrate: removing locks makes the program faster but unsafe and inaccurate.

This project demonstrated the real-world trade-off between performance and correctness in multithreaded programming. The locked version guaranteed accurate results but required extra runtime due to synchronization overhead, while the no-lock version achieved higher speed at the cost of data integrity. The results also showed that increasing thread count improves performance up to a certain point, after which context switching and locking overhead reduce efficiency.

In summary, parallelism can greatly speed up computation, but synchronization is essential when multiple threads share data. The experiment highlights the importance of balancing performance and accuracy, which is a fundamental principle in system design.