Ethan Kanyid

Cpts 360

Professor Corrigan

4-7-25

Lab 10 – Threads

In this lab, we studied the performance of matrix multiplication, focusing on how matrix size and the number of threads impact computation time. Matrix multiplication is computationally intensive and optimizing it through multi-threading can reduce execution time. The goal was to measure CPU and elapsed times under various conditions, explore thread distribution, and understand the diminishing returns of adding more threads. The experiment was divided into two parts: Part 1 focused on single-threaded execution, while Part 2 examined multi-threading.

In Part 1, we tested matrix multiplication with different matrix sizes (n, m, p) to ensure the timing was accurate. Doubling one dimension roughly doubled the CPU time, and doubling all three dimensions increased the time by a factor of eight, indicating the time complexity scales linearly with matrix size (see figure 1). After running the tests multiple times, the CPU and elapsed times showed little variation, especially when no threads were used, demonstrating stable performance in single-threaded execution.

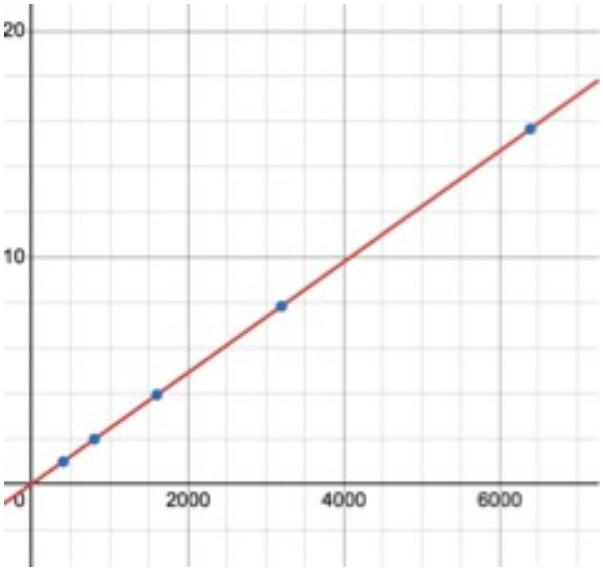

Figure 1: Linear Progression with Increasing Matrix Size

Part 2 extended the experiment to multi-threading, where we tested the effects of varying the number of threads on performance. As expected, increasing the number of threads reduced the elapsed time exponentially initially (see figure 2), but this improvement slowed down after about four threads. The performance gains from adding threads plateaued beyond that point, likely due to hardware limitations such as the number of CPU cores and the overhead from managing multiple threads.

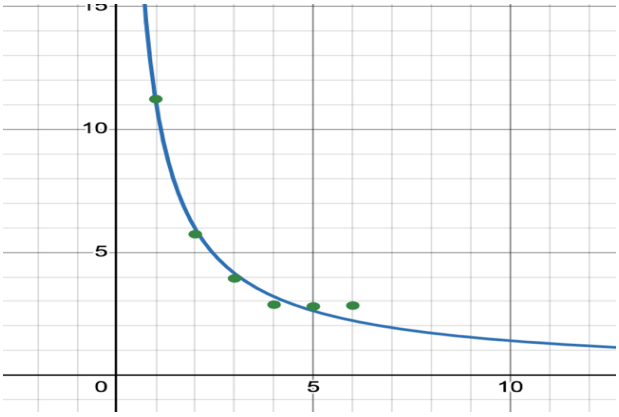

Figure 2: Exponential Regression with Increasing Threads

We also observed how threads were distributed across the matrix rows. In smaller matrices, some threads didn't receive any rows to process, causing imbalances. In larger matrices, the distribution was more even, with variations in row processing within 10% of the average. This indicates that the program was generally effective at distributing work, though smaller matrices posed challenges for perfect parallelization. Additionally, after using more than four threads, performance slowed down, which could be due to resource locks, such as mutex locking, where threads must wait to access shared resources.

The results confirmed that multi-threading improves matrix multiplication performance, but only up to a certain point. While larger matrix dimensions increase CPU time and elapsed time, adding more threads initially decreases elapsed time. However, after around four threads, additional threads offer diminishing returns. This suggests that optimizing the number of threads based on hardware capabilities is essential for maximizing performance. The findings also highlight the challenge of achieving perfect parallelization, particularly in smaller matrices, where work distribution can be uneven.