Raul Martinez

Dr. Niel B Corrigan

CPT_S 360

April 3rd, 2024

Lab 10 Threads Report

Introduction

In this lab, we explored the advantages of multithreading through a compute-intensive task which was matrix multiplication. Matrix operations are common in scientific computing and are well-suited for parallel processing because they involve repetitive and independent calculations. Our goal was to compare the execution time of an unthreaded matrix multiplication program with a multithreaded version and analyze how performance changes with the number of threads.

To measure performance, I recorded both CPU time and wall-clock time under various thread counts. By experimenting with different matrix sizes and thread configurations, I was able to observe how efficiently the workload was distributed across threads and when adding more threads stopped improving performance. This report describes the experiment setup, presents the results, and discusses what they reveal about the system's threading behavior.

Methodology

To test the impact of multithreading on matrix multiplication, I worked with two versions of the same program: one single-threaded (Part 1) and one multithreaded using POSIX threads (Part 2). The programs were run on a WSU-TC elec Linux server using the command line.

Matrix multiplication was implemented with a loop that calculated each row of the result matrix. In the multithreaded version, threads shared access to the input matrices and used a mutex-protected index to pull rows to work on. Each thread tracked how much CPU time it used and how many rows it completed.

For timing, I used the clock_gettime() function to measure both:

- **CPU time** (CLOCK_PROCESS_CPUTIME_ID)
- **Wall clock (elapsed) time** (CLOCK_REALTIME)

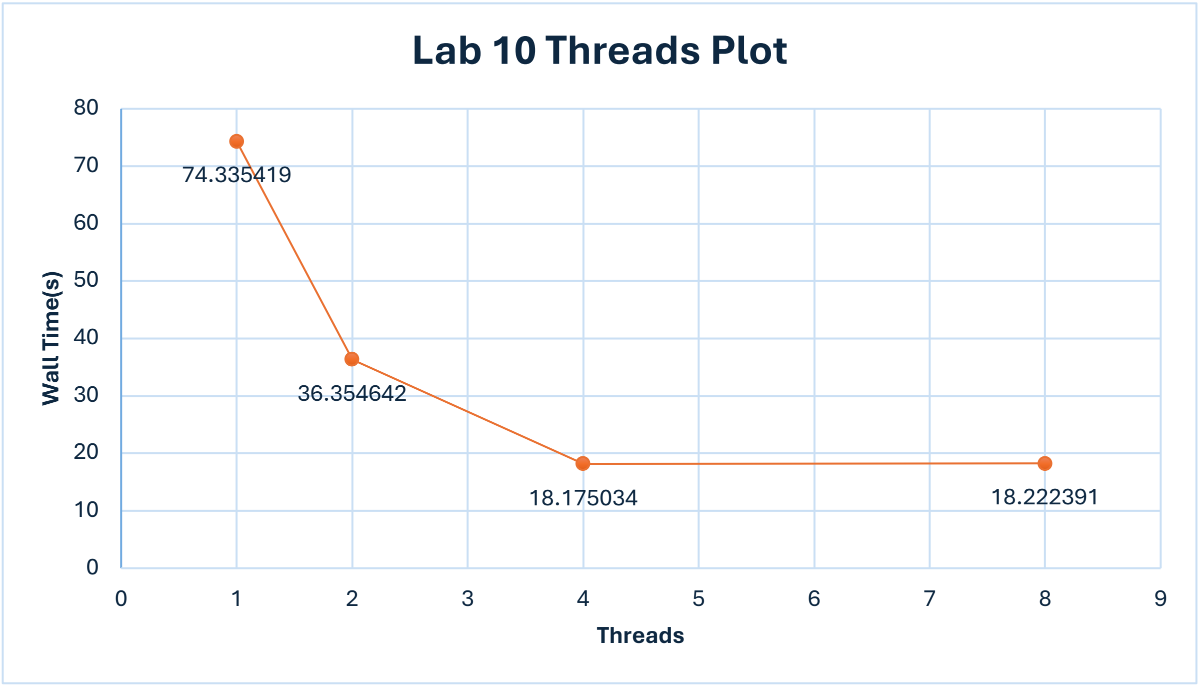
I ran tests using square matrices of size 2000×2000 and varied the number of threads: 1, 2, 4, and 8. Each run printed the time taken and workload distribution among threads. I made sure to run the same size and configuration multiple times to check for consistency in the results.

All measurements were done on a relatively idle system to reduce variability, and the output data was used to analyze how well the program scaled with more threads.

Results

To evaluate the performance of multithreaded matrix multiplication, I ran the experiment_tmm_pt2_tplt program with matrix dimensions n = m = p = 2000, while varying the number of threads. The following table summarizes the wall clock time, total CPU time, and the per-thread CPU time reported for each test run:

Threads	Wall Time (s)	CPU Time (s)	Thread CPU Sum (s)	Rows Done (per thread)
1	74.335419	74.328019	74.32772	2000
2	36.354642	72.694328	72.6938	993, 1007
4	18.175034	72.486978	72.485996	498, 503, 502, 497
8	18.222391	72.758332	72.756602	252, 250, 250, 253, 249, 250, 248, 248



From this data, we can see that *wall-time* drops significantly as we increase the number of threads from 1 to 4. However, the benefit plateaus around 4 threads — increasing to 8 threads results in negligible improvement in wall time. Meanwhile, CPU time remains fairly-consistent across all thread counts, confirming that the total computational workload stays the same and only the parallel distribution changes.

This performance behavior indicates good scalability up to a certain number of threads, likely corresponding to the number of physical cores available on the system. The distribution of rows per thread is also fairly-even, showing balanced workload allocation by the thread dispatcher.

Conclusion

This lab demonstrated how multithreading can reduce computation time for matrix multiplication by taking advantage of parallelism. By comparing performance across different thread counts, I confirmed that the biggest improvements in wall time occurred when increasing threads up to the number of available cores. After that, additional threads offered little to no benefit due to hardware limitations and context switching overhead.

The experiment also showed consistent CPU time across tests, which reflects that the total computational work remains the same regardless of threading. What changes is how efficiently that work is distributed and executed.

Overall, this lab reinforced the importance of understanding both the algorithm and the system it runs on. Performance tuning isn't just about adding more threads—it's about knowing when and how parallelism helps. This kind of testing is key to writing efficient, scalable software.

End.