Angelica Vargas

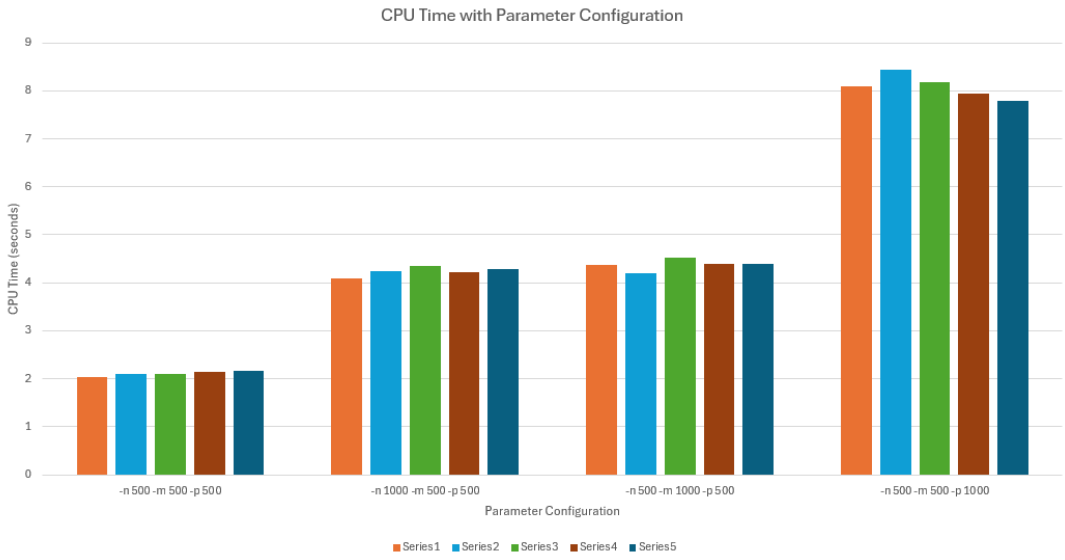
Cpts 360

Lab 10

Part 1

7. How does doubling any of these parameters affect the CPU time?

To ensure the accuracy of my timing values, I conducted five tests using different command line arguments for -n, -m, and -p to achieve run times exceeding two seconds. For example, running the program with -n 500 -m 500 -p 500 consistently resulted in wall clock times ranging from 2.0 to 2.2 seconds. When doubling any of these parameters, such as doubling the size of the matrices, I observed a proportional increase in CPU time. For instance, running the program with -n 1000 -m 500 -p 500 resulted in CPU times ranging from 4.095 to 4.398 seconds, compared to the approximately 2-second CPU times observed with smaller matrices. This increase in workload directly impacted the CPU time required to execute the program.

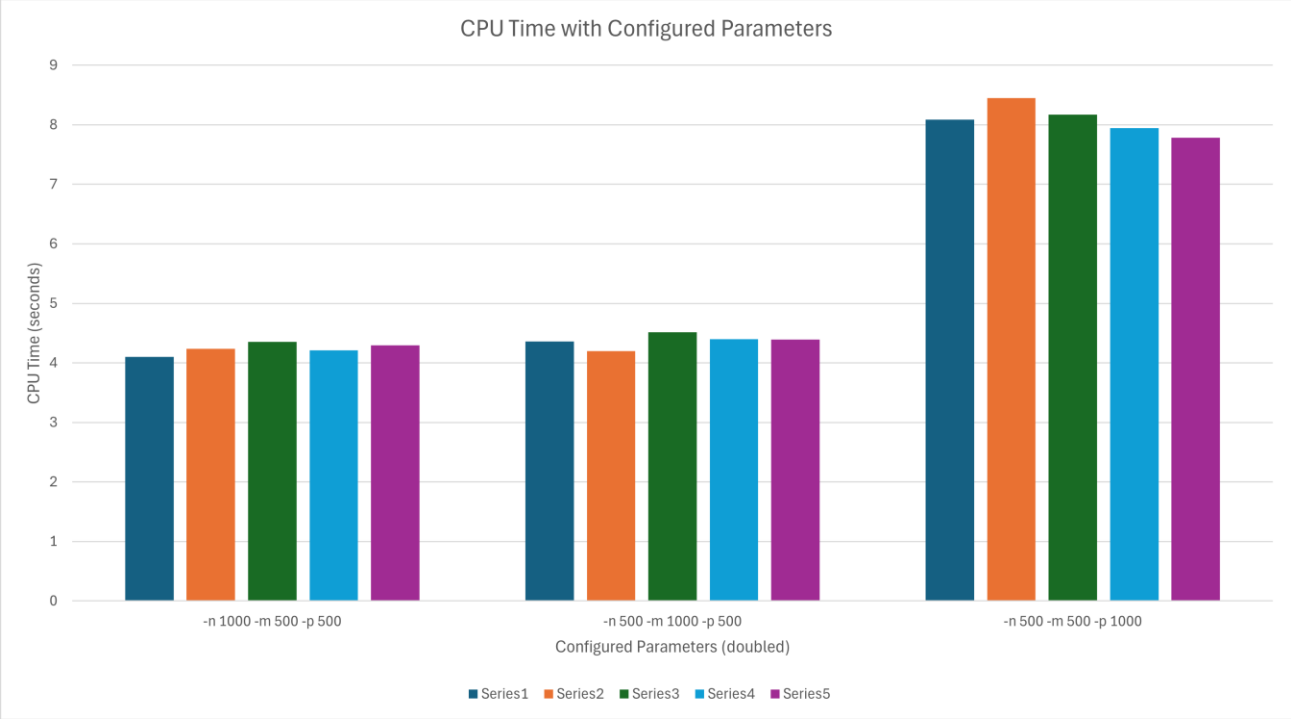


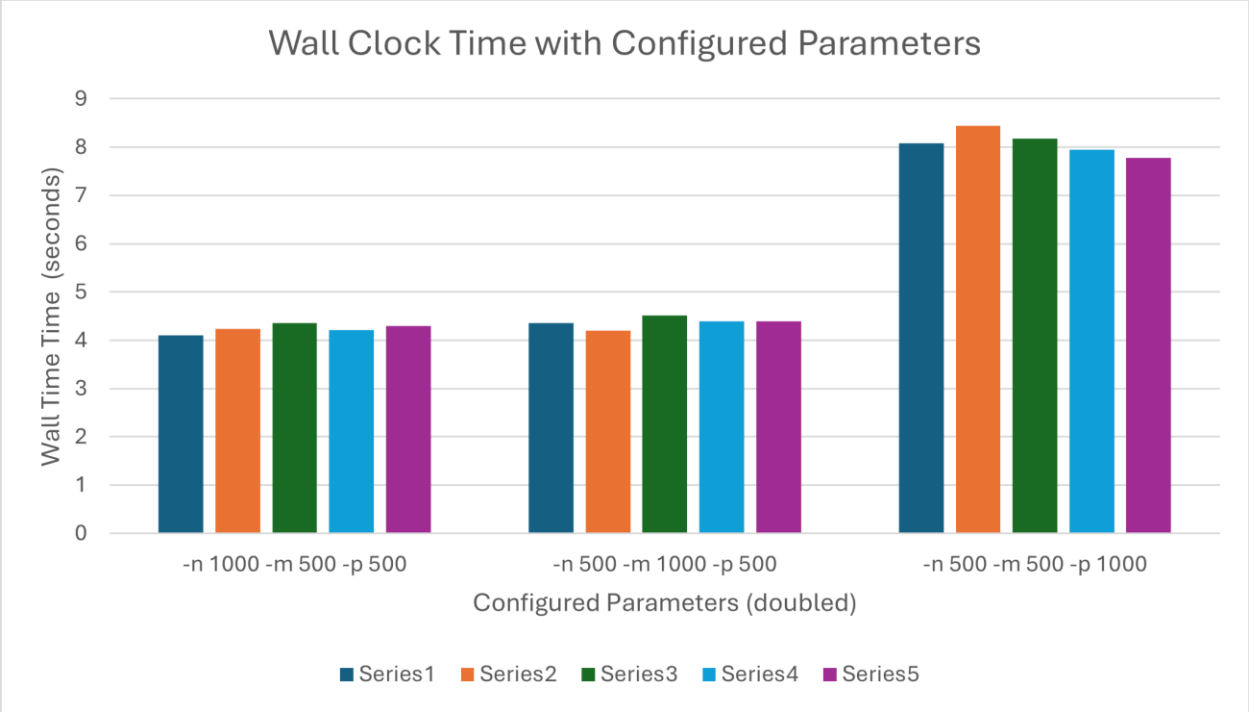
8. Run the same test several times and note the elapsed (aka "wall clock") and CPU times. Are they always the same? If not, how much do they vary, does one vary more than the other, and

why do you think they vary? If you were reporting this to someone else, what would be the best way to describe your result and its variation (if any)?

In my tests, I ran the same test scenario five times with each doubled parameter and noted differences between both the elapsed (wall clock) and CPU times. While the timing values generally showed a high degree of consistency across runs, there were minor variations. For example, running the program with -n 1000 -m 500 -p 500 had a wall clock time ranging from 4.099 to 4.356 seconds and a CPU time ranging from 4.099 to 4.355 seconds. On the other hand, running the program with -n 500 -m 500 -p 1000 had a wall clock time ranging from 7.781 to 8.448 seconds and CPU times ranging from 7.781 to 8.448 seconds.

These variations, although small, can be attributed to factors such as background processes, system resource allocation, and scheduling intricacies. Despite these variations, the overall trends in timing values remained consistent across multiple test runs.

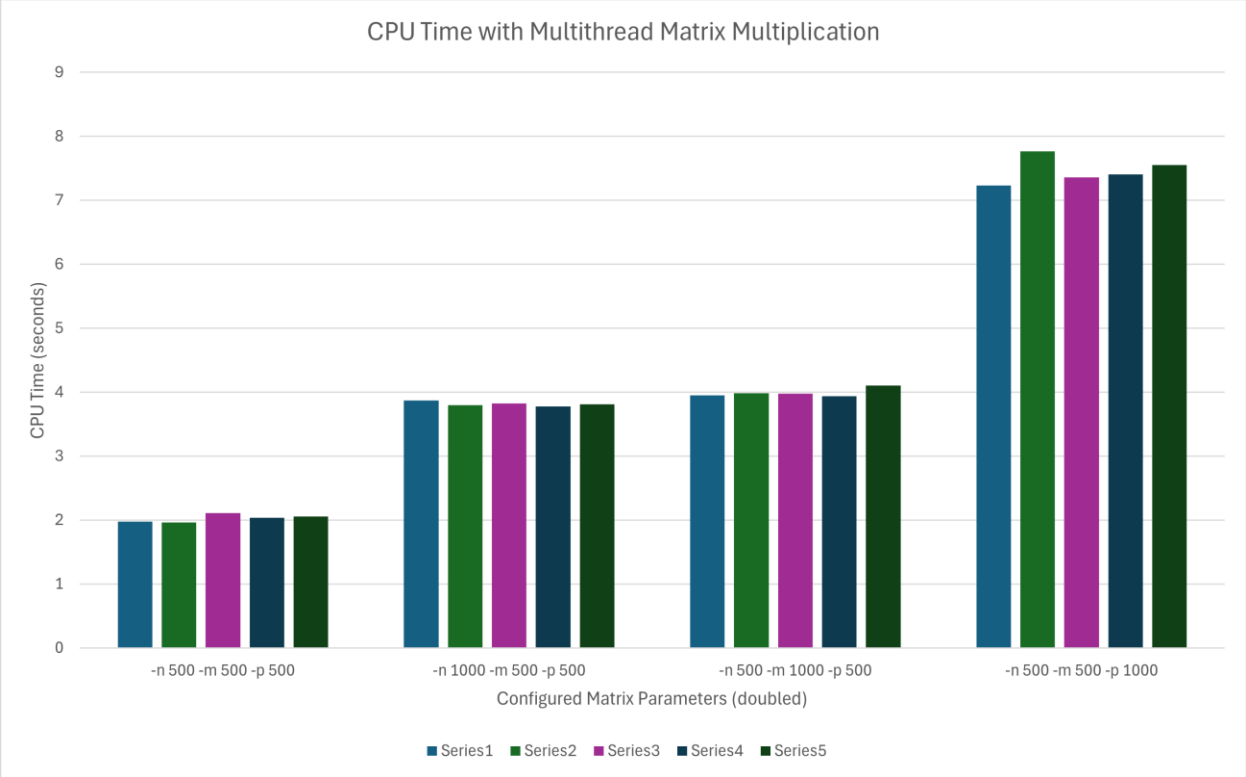




Part 2

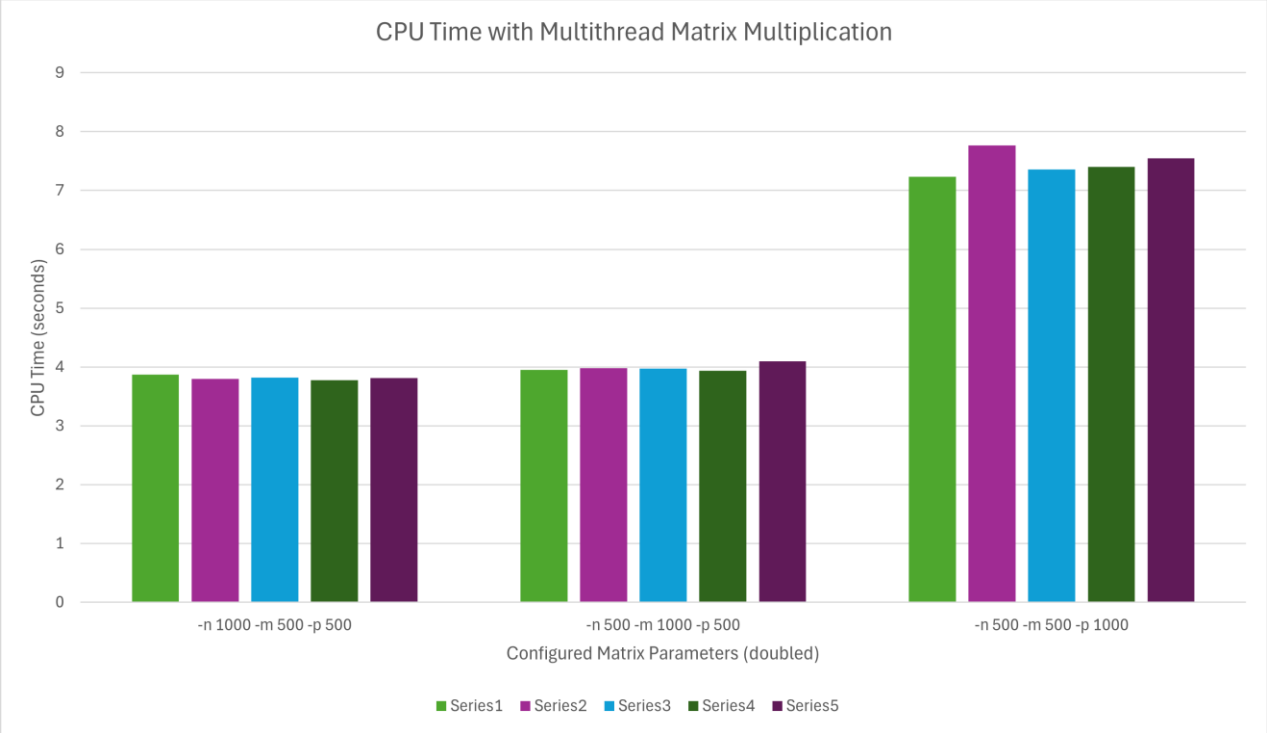
5. How does doubling any of these parameters affect the CPU time?

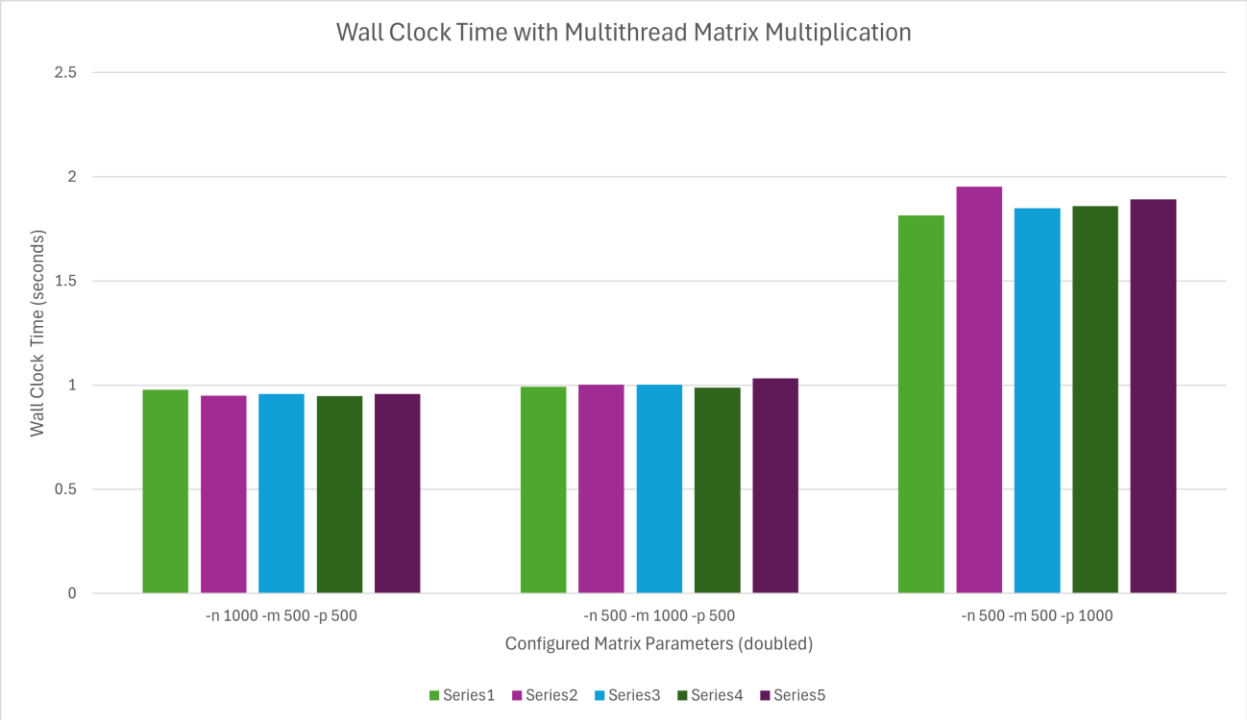
To ensure the accuracy of my timing measurements, I once again conducted multiple tests (five tests) using different command line arguments for -n, -m, and -p to achieve run times of at least a few seconds. For instance, running the program with -n 500 -m 500 -p 500 consistently resulted in CPU times ranging from approximately 0.49 to 0.53 seconds, closely matching the wall clock times of around 0.49 to 0.53 seconds. Similarly, doubling any of the parameters, such as running the program with -n 1000 -m 500 -p 500, led to proportional increases in CPU times to approximately 0.95 to 0.98 seconds, consistent with the expected doubling effect on the computational workload.



6. Run the same test several times and note the elapsed and CPU times. Are they always the same? If not, how much do they vary? Why do you think they vary? If you were reporting this to someone else, what would be the best way to describe your result and its variation (if any)? Can you think of any way to reduce the variation?

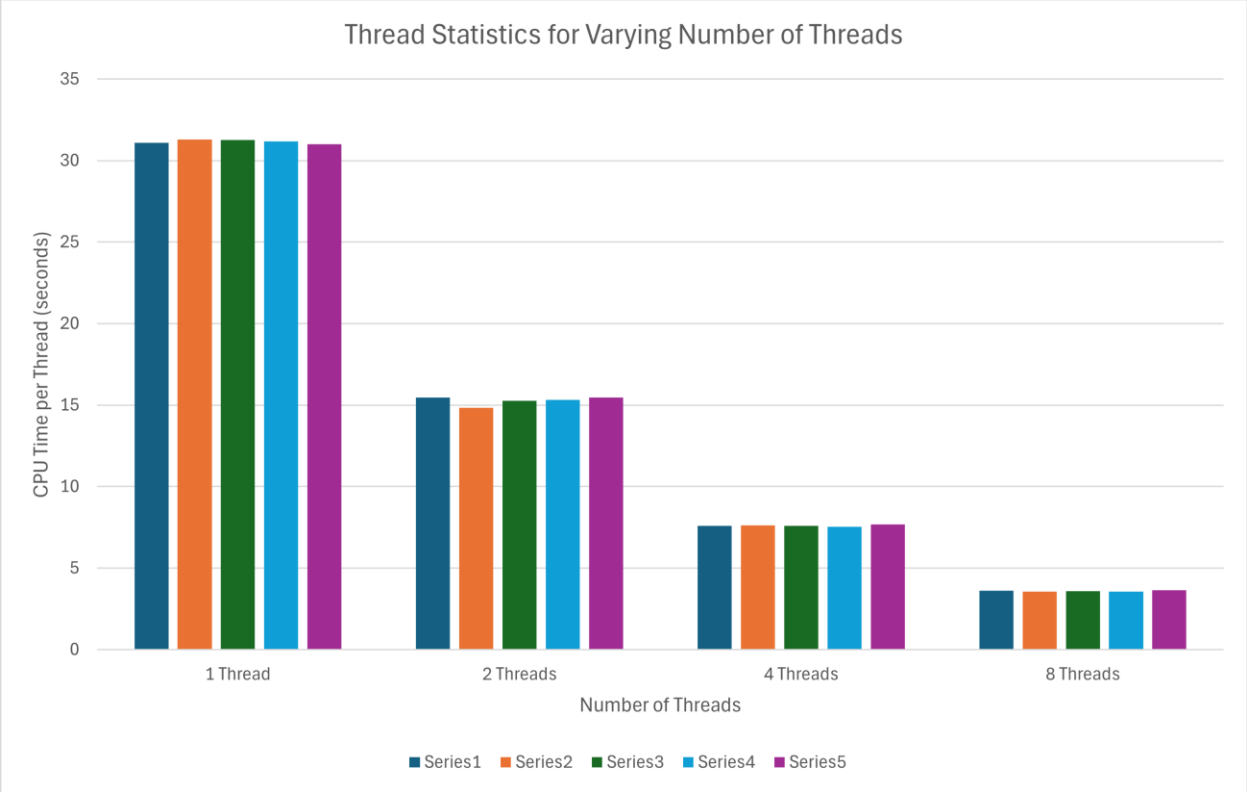
Despite my efforts to maintain consistency, minor variations were observed in both elapsed (wall clock) and CPU times across multiple test runs. These variations could be attributed to factors such as background processes, system resource allocation, and scheduling intricacies, along with fluctuations in system load and CPU utilization. To effectively describe these results and their variation, providing a comprehensive overview of the range of CPU times observed across multiple runs, along with their average and standard deviation, is essential. To reduce variation, strategies such as minimizing background processes, system load, and averaging results from multiple runs could be employed.





7. Look at the individual thread statistics for varying numbers of threads (-t). Do all threads get about the same amount of time? Do all threads get about the same number of rows to multiply?

This metric indicates how much CPU time each thread consumes on average during the execution of the matrix multiplication task. In this case, we observe that as we increase the number of threads from 1 to 8, the CPU time per thread decreases. This suggests that each thread is getting roughly the same share of CPU time, regardless of the number of threads running concurrently. The "Multiplied By Thread" data shows how many rows each thread is responsible for multiplying. In a parallel matrix multiplication task, the workload is typically divided among threads, with each thread responsible for processing a subset of the data. The number of rows assigned to each thread varies slightly but not significantly across different configurations.



8. Investigate the effect of varying the number of threads on the elapsed and CPU times. Is there a number of threads after which that decrease stops or slows down a lot? If so, what is that number and can you think of a reason for this? (Hint: hardware)

Based on the provided data for different numbers of threads (1, 2, 4, and 8) in the previous question, it's evident that the average CPU time per thread decreases as the number of threads increases. This suggests that the parallelization of the matrix multiplication task is effective, as dividing the workload among multiple threads reduces the computational time per thread. However, the scalability of the implementation appears to plateau or even slightly worsen as the number of threads increases from 4 to 8. This indicates that beyond a certain point, adding more threads does not significantly improve performance, possibly due to overhead or resource contention. Therefore, while parallelization shows noticeable gains in efficiency with a moderate number of threads, increasing the number of threads may not be completely efficient.