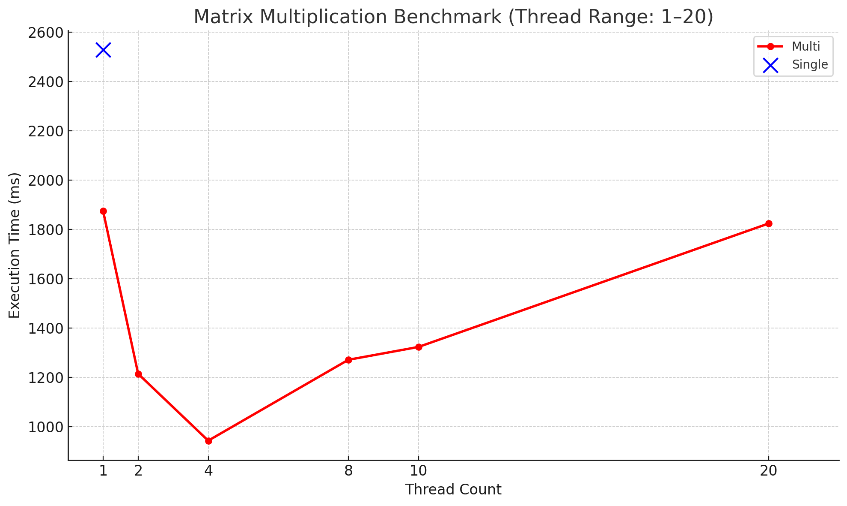
|  |  |  |
| --- | --- | --- |
| **Student ID** | **Name** | **Lastname** |
| 21COMP1066 | Çağrı | ÇIBUK |
| 20SOFT1018 | Selen | ERK |

Our project's performance significantly varied with different thread counts. Based on the results collected in matrix\_benchmark\_results.csv, increasing the number of threads from 1 to 4 provided the most substantial improvement. Execution time dropped from 2529 ms (single-threaded) to 944 ms at 4 threads, which was the best-performing configuration. This clearly indicates that the “sweet spot” for our system was 4 threads. However, beyond this point, performance gains diminished. For example, using 10 threads resulted in a higher execution time (1324 ms) compared to 4 threads. This behavior was consistent across multiple benchmark runs and confirms that blindly increasing parallelism does not guarantee better results.

metin, çizgi, öykü gelişim çizgisi; kumpas; grafiğini çıkarma, diyagram içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

The performance degradation observed at higher thread counts is closely related to the number of physical CPU cores available on the system. Our test machine was equipped with an Intel® Core™ i5-9300H processor featuring 4 physical cores and 8 logical threads. When the thread count exceeded the number of available cores, the CPU was forced to context switch between threads, resulting in overhead and decreased efficiency. This explains why performance improved up to 4 threads. Which aligned with the number of physical cores but then began to decline as more threads were introduced. Interestingly, we observed an exception between 100 and 200 threads, where execution time unexpectedly dropped from 2145 ms to 1415 ms. This anomaly may be attributed to JVM-level optimizations or transient OS-level scheduling advantages. However, the benefit was not sustained: performance degraded again at 400 threads (1863 ms), reaffirming that excessive threading generally results in inefficiencies on hardware-limited systems.

Matrix A was partially read from the file by each thread, whereas Matrix B was fully loaded into memory at the beginning. This design choice was made to optimize memory efficiency. Matrix A can be extremely large depending on its dimension, and reading only the required rows per thread reduces memory usage and avoids storing unused data. Matrix B, on the other hand, is used by all threads for every multiplication operation, so preloading it once avoids repeated file I/O and ensures consistent access. If both matrices were fully loaded for each thread, it would consume large amounts of memory and possibly lead to resource exhaustion in systems with limited RAM.

To measure execution time, we used System.currentTimeMillis(). While this method offers only millisecond-level granularity and can be affected by system background processes, it was sufficient for our benchmarking purposes. We ran the test script multiple times and consistently observed the same performance trends. The matrix\_benchmark\_results.csv file captured these results, and although minor fluctuations occurred between runs, the overall outcomes were reliable. For high-precision measurements, tools such as System.nanoTime() or external profilers could be used. However, for our use case of comparing thread-based performance scaling, System.currentTimeMillis() was an appropriate and practical choice.