|  |  |  |
| --- | --- | --- |
| **Student ID** | **Name** | **Lastname** |
| 21COMP1066 | Çağrı | Çıbuk |
| 20SOFT1018 | Selen | Erk |

Our project's performance significantly varied with different thread counts. Initially, increasing the number of threads resulted in a noticeable reduction in execution time, especially when moving from 1 to 2 or 4 threads. However, adding more threads beyond a certain point led to diminishing returns. On our testing system, the sweet spot was observed at 4 threads, where the computation time was shortest. Beyond this point (e.g., 8 threads), the performance gains plateaued or even worsened slightly due to thread management overhead. This behavior highlights the importance of finding an optimal thread count based on the system's capabilities rather than simply maximizing parallelism.

Performance degradation with excessive threading is primarily linked to the number of available CPU cores. When the number of threads exceeds the number of physical or logical cores, context switching and CPU contention increase. This overhead leads to inefficient CPU usage and longer execution times. In our case, the system had 4 cores, and using more than 4 threads introduced overhead that outweighed the benefits of parallel processing. This behavior illustrates the practical limits of concurrency and underscores the necessity of tailoring thread counts to system specifications.

Matrix A is read partially from the file in each thread because it enables memory efficiency and avoids loading a massive dataset entirely into memory. Each thread only reads the rows it needs, keeping memory usage minimal and allowing the system to handle larger matrices. In contrast, Matrix B is read entirely once and stored in memory because it is used by all threads. Loading it once prevents redundant file reads and ensures faster access during computation. Attempting to load both matrices entirely would consume a large amount of memory and could exceed system capacity for very large matrices, hence the partial-loading strategy for Matrix A.

Using System.currentTimeMillis() for performance measurement provides a rough estimate of execution time but is not highly precise. It measures wall-clock time with millisecond granularity and can be affected by background system processes. To improve reliability, we ran the benchmark script multiple times and compared results in the generated .csv file. Although there was slight variability in results, the overall performance trends were consistent. For more accurate timing, alternatives like System.nanoTime() or dedicated benchmarking tools could be used, but System.currentTimeMillis() was sufficient for observing general performance differences in this context.