Student ID	**Name**	**Lastname**
21comp1021	Tuğçe	Yeşilyurt
21comp1004	Hümeyra	Bilgin

Across a range of thread counts, our matrix multiplication benchmark showed intriguing performance trends. The multi-threaded approach, which started with a single thread, unexpectedly outperformed the single-threaded version by about 21% (2705 ms vs 3405 ms). Performance greatly improved as we raised the thread count to two, achieving an execution time of 1978 ms, which is 42% faster than the single-threaded version. With 4 threads, performance reached its peak (1767 ms), but with 8 threads, it significantly worsened (2636 ms). According to this pattern, the "sweet spot" of our hardware for this particular matrix multiplication task is four threads. The data demonstrates that merely increasing the number of threads does not ensure improved performance, and in our instance, there is a clear optimal thread count beyond which performance degrades.

There are a number of reasons why our system's CPU architecture degrades performance as thread counts increase. The physical core count of modern CPUs is usually restricted (probably 4 in our testing equipment), and the operating system has to conduct context switching when the number of threads exceeds the available core count. There is a large overhead associated with this thread state saving and restoration process. Furthermore, there is more competition for shared resources like cache memory and memory bandwidth when there are a lot of threads. Additionally, thread management and creation use system resources. The abrupt decrease in performance after four threads indicates that our system most likely has four physical cores or eight cores with hyperthreading, where the overhead of thread management soon outweighs the performance gain of parallelism.

For good reason, our solution reads Matrix A partially (row by row) and loads Matrix B completely into memory. This method preserves performance while optimizing memory usage. Each thread requires all columns from Matrix B for multiplication, but only the corresponding rows from Matrix A because each thread computes a different row of the output matrix. Redundant file I/O operations are eliminated by loading Matrix B only once and distributing it among threads. We would have used a lot more memory if we had loaded both matrices completely, which would have resulted in thrashing when handling really big matrices. Our method demonstrates a useful implementation of operating system fundamentals such as memory management and I/O optimization by striking a balance between memory efficiency and computational performance.

System.currentTimeMillis() has limitations but is sufficiently reliable for measuring performance. There was some variation in our benchmark results from run to run, especially when the number of threads increased. This variability is caused by a number of factors, including the fact that the method measures wall-clock time rather than CPU time, which leaves it vulnerable to changes in system load; the just-in-time compilation and garbage collection of the JVM can introduce timing inconsistencies; and operating system scheduling decisions have an impact on the timing of thread execution. We would need to perform several iterations (at least five to ten) and compute average execution times in order to obtain more accurate measurements. Using Java's Microbenchmark Harness (JMH) or measuring CPU time rather than wall-clock time are examples of more complex methods. However, our tests were adequate to determine the ideal thread count for our particular hardware setup as well as distinct performance patterns.