COMP3432 – OPERATING SYSTEMS
PROJECT REPORT FILE
SPRING 2025

IŞIK UNIVERSITY
COMPUTER SCIENCE AND ENGINEERING

1. **How did the performance change with different thread counts? Did adding more threads always make it faster? There is usually a "sweet spot" (that means a number of threads that gives the best speed). What is the sweet spot for your hardware?**

   The single-threaded execution completed in 1197 ms. Initially, increasing the number of threads did not immediately lead to faster execution times. For instance, 1 thread took 1546 ms, 2 threads took 2266 ms, 4 threads took 2674 ms, and 8 threads took 2577 ms. This suggests an initial overhead associated with thread creation. There was a clear improvement in speed when using 10 threads, with the task finishing in 1599 ms. This was much faster than the earlier multi-threaded tests. The best result came from using 50 threads, which completed the task in 1586 ms. This suggests that 50 threads gave the best performance in this test. Other high thread counts, like 400 (1777 ms) and 500 (1727 ms), also gave good results, but they were still slower than the 50-thread run. However, the results were not always consistent. For example, 20 threads took a much longer 16834 ms, and 100 and 200 threads took 7479 ms and 6974 ms, which were also quite slow.

2. **Why did performance degrade with too many threads? How does this relate to the number of CPU cores on your system?**

   When the number of threads is much higher than the number of CPU cores, the system becomes less efficient. This happens because the operating system has to constantly switch between threads, saving and loading their states. This process takes time and reduces the time spent on real work. Also, shared resources like matrixB and resultMatrix need to be protected to avoid errors. To do this, threads use locks or other safety methods. However, these methods can slow things down because threads sometimes have to wait for each other. As a result, the advantages of using many threads can be reduced.

3. **Why read Matrix A partially from file but load Matrix B entirely? Why not load both matrices entirely?**

   The project is designed to load Matrix B fully into memory, while Matrix A is read only one part at a time—row by row—by each thread. This method helps save memory. If both matrices were loaded completely into memory, the program might run out of memory, especially when working with large matrices. Matrix B is fully loaded because its columns are needed many times for each calculation in the result matrix. If Matrix B were also read in parts, each thread would need to access many different columns from

the file again and again. This would make the program very slow due to inefficient file reading. On the other hand, Matrix A is read in parts. Each thread works on a set of rows from Matrix A and uses them to calculate part of the result matrix. Since each thread uses rows that are next to each other, this type of reading called as sequential access which is much faster.

4. **How reliable are the performance measurements with System.currentTimeMillis()? You might need to run the test script a couple of times and check the generated .csv file each time to answer this question.**

'System.currentTimeMillis()`This method is easy to use, but its accuracy depends on the system clock, which usually only measures in milliseconds. Because of this, it might not be accurate enough for very short tasks. To check how reliable the results are, the test was run several times. At the `matrix_benchmark_results.csv` file from different runs, there are some differences, even for the same settings (like single-thread or a specific number of threads). These differences can happen for many reasons: the operating system might give different amounts of time to the program, other programs might be running in the background, the Java Virtual Machine (JVM) may run slower at first due to warm-up time, or garbage collection might interrupt the program. These factors can affect which can change the measured execution times. For more accurate results, `System.nanoTime()` can be used because it gives more precise timing and isn't affected by the system clock in the same way.