Assignment 1 - Q1

Date	@September 11, 2025
Label	Assignment

1. Matrix Multiplication

1.1 Sequential Implementation

Code Snippet

```
/**
 * Returns the result of a sequential matrix multiplication
 * The two matrices are randomly generated
 *
 * @param a is the first matrix
 * @param b is the second matrix
 * @return the result of the multiplication
 */
public static double[][] sequentialMultiplyMatrix(double[][] a, double[][] b) {
    Integer rows = a.length;
    Integer common = a[0].length;
    Integer cols = b[0].length;

    double[][] bTranspose = computeTranspose(b);
    double[][] res = new double[rows][cols];

    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            double sum = 0.0;

            for (int i = 0; i < common; i++) {
                sum += a[r][i] * bTranspose[c][i];
            }

            res[r][c] = sum;
        }
```

```
    }

    return res;
}
```

Description

The implementation for sequential matrix multiplication is fairly straightforward. We start with our two input matrices and compute bounds represented by the `rows`, `common`, and `cols` integers. The `common` integer represents the constraint that the number of columns in matrix `a` must equate to the number of rows in matrix `b`. We take the transpose of matrix `b` to leverage cache locality and speed up the runtime performance of our sequential implementation (refer to the source code file to see `computeTranspose()`'s implementation). After these steps, we can iterate over both input matrices to compute the final result.

Validation

To validate this method, a test suite was created. We used gradle as our build tool, so we could run the unit test suite for each of the project's modules every time a change was made. The goal with unit testing is ensuring proper functionality while minimizing the number of tests. As such, our test suite seeks to cover as many main/edge cases while remaining lean.

1.2 Parallel Implementation

Code Snippet

```
/**
 * Returns the result of a concurrent matrix multiplication
 * The two matrices are randomly generated
 *
 * @param a is the first matrix
 * @param b is the second matrix
 * @return the result of the multiplication
 */
public static double[][] parallelMultiplyMatrix(double[][] a, double[][] b) {
    Integer rows = a.length;
    Integer cols = b[0].length;

    double[][] bTranspose = computeTranspose(b);
    double[][] res = new double[rows][cols];

    ExecutorService executor = Executors.newFixedThreadPool(NUMBER_THREADS);
    Future<?>[] futures = new Future<?>[rows];

    for (int r = 0; r < rows; r++) {
        futures[r] = executor.submit(new MatrixMultiplicationTask(r, a, bTranspose, res));
    }

    for (Future<?> f : futures) {
        try {
            f.get();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    executor.shutdown();
    return res;
}
```

Description

The implementation for the parallel implementation builds on the intuition for the sequential solution. The main structure of the method looks similar. However, there is a key distinction. Previously, we would iterate over each row of matrix `a` and compute the dot products for that row against all columns of matrix `b`. In this new parallelized solution, we create a fixed thread pool bounded by the number of threads defined by `NUMBER_THREADS`. The pool is responsible for running tasks (snippet provided below) that are defined as follows: the individual task is responsible for all computations of a single row in matrix `a` against all the columns in matrix `b` and updating the resulting matrix `res` accordingly. In this case, since we are only reading from the input matrices `a` and `b`, and the write operations to `res` are independent given that each task has a different row, we don't need explicit synchronization between workers of the thread pool.

```
class MatrixMultiplicationTask implements Runnable {
    Integer rowIx; // Index of row in matrix A
    double[][] a;
    double[][] b;
    double[][] res;

    public MatrixMultiplicationTask(Integer rowIx, double[][] a, double[][] b, double[][] res) {
        this.rowIx = rowIx;
        this.a = a;
        this.b = b;
        this.res = res;
    }

    public void run() {
        Integer common = a[0].length;
        Integer cols = res[0].length;

        for (int col = 0; col < cols; col++) {
            double sum = 0.0;

            for (int i = 0; i < common; i++) {
                sum += a[rowIx][i] * b[col][i];
            }

            res[rowIx][col] = sum;
        }
    }
}
```

Validation

Similar to the sequential solution, validation is performed by the unit test suite. We also developed a benchmark suite to ensure that this parallel method performed better than the sequential method as the number of threads scale.

1.3 Comparing Sequential vs Parallel

We added two methods to measure execution time for both the sequential and parallel solutions. These methods can be found in the source code file as `benchmarkSequential()` and `benchmarkParallel()`. The number of threads in the thread pool was set to 5. The two paragraphs below cover how we ensured that the benchmark methods returned plausible results.

Both sequential and parallel implementations were ran against the same input matrices. To ensure the plausibility of the results, we also cross-checked the output of each implementation to ensure consistency. To ensure consistency of our runtime results, we ran the benchmark suites multiple times and manually verified that the runtimes were on the same scale for each matrix size.

Timing sanity checks were also applied. For example, small matrices (sizes below 1000) should run almost instantaneously whereas larger matrices would take longer. We also checked that the relationship between

matrix size and runtime scaled exponentially for the sequential implementation given its exponential runtime. The final part of our timing sanity checks revolved around ensuring that the parallel implementation was generally more performant than its sequential counterpart (given that the number of threads is greater than 1). We say generally because for small matrices, the overhead incurred via context switching between threads may cause the method to run slower than the sequential one.

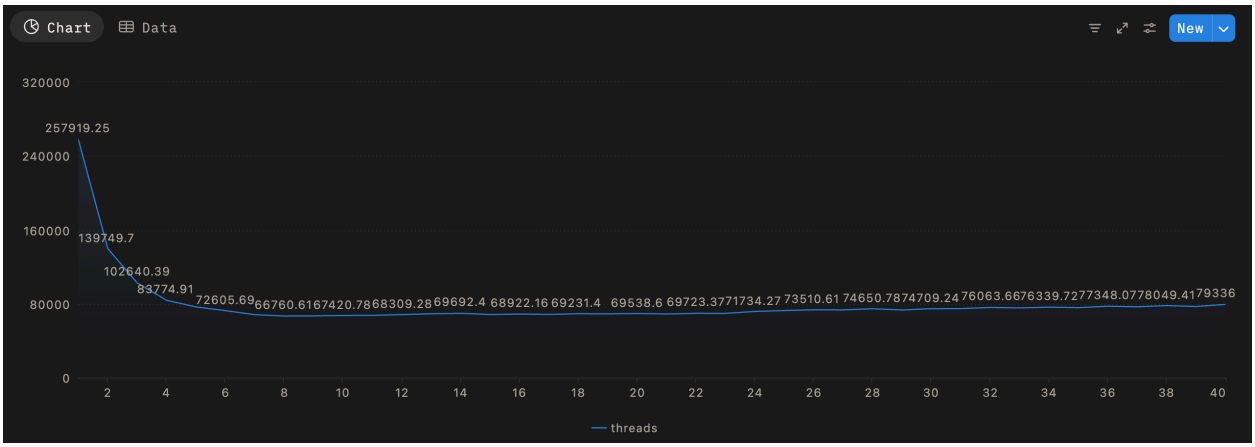
1.4 Varying the Number of Threads

The following output was from running the benchmark functions while varying the number of threads between 1 and 40. The upper bound was chosen mainly because I was able to run this while I went out for a bike ride and had 2 hours of buffer, so I decided to be generous with it.

```
--- Running Benchmark by Threads Suite ---
Sequential multiply took 257375.127 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 1] Parallel multiply took 257919.245 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 1]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 2] Parallel multiply took 139749.696 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 2]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 3] Parallel multiply took 102640.388 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 3]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 4] Parallel multiply took 83774.910 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 4]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 5] Parallel multiply took 76647.784 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 5]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 6] Parallel multiply took 72605.688 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 6]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 7] Parallel multiply took 68229.204 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 7]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 8] Parallel multiply took 66760.609 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 8]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 9] Parallel multiply took 66901.969 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 9]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 10] Parallel multiply took 67420.776 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 10]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 11] Parallel multiply took 67487.678 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 11]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 12] Parallel multiply took 68309.284 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 12]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 13] Parallel multiply took 69196.340 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 13]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 14] Parallel multiply took 69692.398 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 14]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 15] Parallel multiply took 68381.723 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 15]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 16] Parallel multiply took 68922.157 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 16]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 17] Parallel multiply took 68494.445 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 17]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 18] Parallel multiply took 69231.399 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 18]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 19] Parallel multiply took 69076.667 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 19]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 20] Parallel multiply took 69538.603 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 20]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 21] Parallel multiply took 68941.253 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 21]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 22] Parallel multiply took 69723.365 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 22]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 23] Parallel multiply took 69629.931 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 23]  PASSED
```

```
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 24] Parallel multiply took 71734.266 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 24]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 25] Parallel multiply took 72735.991 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 25]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 26] Parallel multiply took 73510.612 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 26]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 27] Parallel multiply took 73395.429 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 27]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 28] Parallel multiply took 74650.781 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 28]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 29] Parallel multiply took 73337.002 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 29]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 30] Parallel multiply took 74709.238 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 30]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 31] Parallel multiply took 74820.939 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 31]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 32] Parallel multiply took 76063.663 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 32]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 33] Parallel multiply took 75585.870 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 33]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 34] Parallel multiply took 76339.715 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 34]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 35] Parallel multiply took 75893.245 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 35]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 36] Parallel multiply took 77348.073 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 36]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 37] Parallel multiply took 76781.887 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 37]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 38] Parallel multiply took 78049.412 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 38]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 39] Parallel multiply took 77061.408 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 39]  PASSED
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 40] Parallel multiply took 79336.152 ms
✓ [MATRIX SIZE = 4000] [NUMBER OF THREADS = 40]  PASSED
```

The following chart represents the runtime plotted against the number of threads. Note: the runtimes are in milliseconds.



From the output, we know that our sequential implementation took roughly 257.92s to multiply two 4000 by 4000 matrices. This runtime thus acts as our baseline. Our parallel implementation with a single thread exhibits virtually the same runtime. At 2 threads, we have a runtime of 139.75s which yields a speed-up of `257.92s / 139.72s ~= 1.85x` which almost halves the sequential runtime. At 4 threads, we have a speed up of

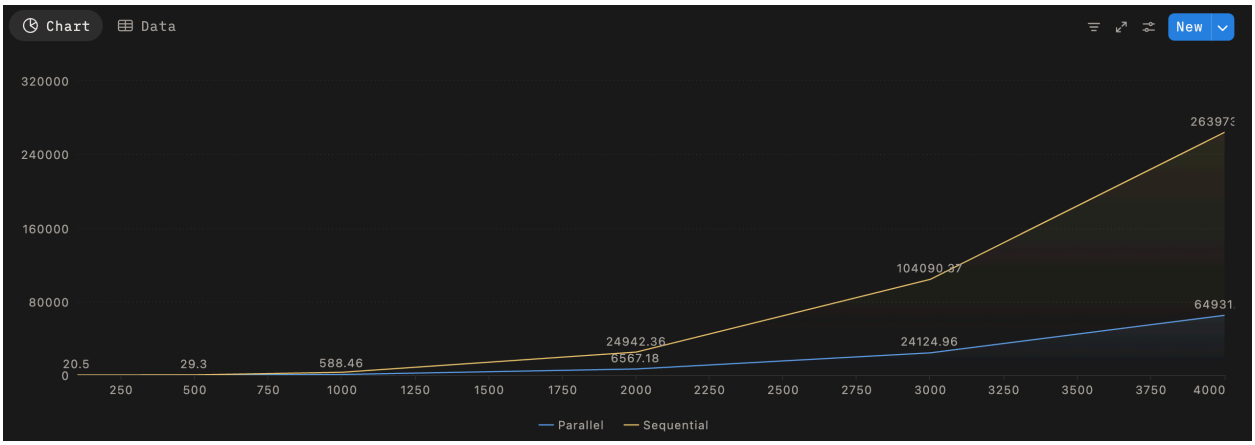
257.92s / 83.77s ~= 3.1x. At 8 threads, we our fastest runtime with a speed-up of 257.92s / 66.76s ~= 3.86x. After that, the runtime plateaus and eventually becomes slightly worst. We can deduce that the speed-up also trends in a similar fashion (plateaus, then becomes slightly worse).

1.5 Varying Matrix Size

The following output was from running the benchmark functions while varying the matrix sizes as defined in the assignment guidelines. The number of threads was set to 8 based on the most optimal runtime from section 1.4.

```
--- Running Benchmark by Matrix Size Suite ---
Benchmark 0: Matrix size = 100
Sequential multiply took 7.196 ms
Parallel multiply took 20.504 ms
✓ Benchmark 0: Matrix size = 100 PASSED
Benchmark 1: Matrix size = 200
Sequential multiply took 27.311 ms
Parallel multiply took 30.963 ms
✓ Benchmark 1: Matrix size = 200 PASSED
Benchmark 2: Matrix size = 500
Sequential multiply took 131.783 ms
Parallel multiply took 29.303 ms
✓ Benchmark 2: Matrix size = 500 PASSED
Benchmark 3: Matrix size = 1000
Sequential multiply took 2984.684 ms
Parallel multiply took 588.456 ms
✓ Benchmark 3: Matrix size = 1000 PASSED
Benchmark 4: Matrix size = 2000
Sequential multiply took 24942.360 ms
Parallel multiply took 6567.175 ms
✓ Benchmark 4: Matrix size = 2000 PASSED
Benchmark 5: Matrix size = 3000
Sequential multiply took 104090.371 ms
Parallel multiply took 24124.961 ms
✓ Benchmark 5: Matrix size = 3000 PASSED
Benchmark 6: Matrix size = 4000
Sequential multiply took 263973.784 ms
Parallel multiply took 64931.440 ms
✓ Benchmark 6: Matrix size = 4000 PASSED
```

The following chart represents the respective runtimes for sequential and parallel plotted against the matrix size. Note: the runtimes are in milliseconds.



1.6 Interpretations

We will start by interpreting the chart generated in section 1.4. From our observations about speed-ups, we notice a sharp increase in performance as we go from 1 thread to 8, followed by a distinct plateau and even a loss in performance as the number of threads grows beyond 8. The explanation for this is fairly straightforward. Having a pool of 1 thread is the same as having a sequential implementation in essence which explains their virtually identical runtimes. As the number of threads grow, more physical cpu cores on the machine are leveraged. The reason performance plateaus beyond 8 threads is because the machine these benchmarks were ran on only have 8 physical cores. As such, having more threads than cores doesn't contribute much to speed-up. On the contrary, having a large amount of threads can hinder performance as runtime cost is incurred via context switching between threads and other thread management overhead. Therefore, we obtain that performance is optimized when the number of threads equates to the number of physical cores.

Section 1.5's chart plots the sequential and parallel runtimes with 8 threads while varying matrix sizes. We have deduced from the previous section that the speed-up with 8 threads is approximately `3.86x ~ 4x` and we can notice that for large matrix sizes (>500-1000), this roughly holds true. The general observation is that the parallel implementation scales much better than the sequential one. The reason behind this is that more threads gives us more workers to break the problem down into smaller chunks and solve them in parallel. An interesting observation in the graph is that for small matrices, the sequential solution exhibits similar if not better runtime than the parallel one. This can be explained by the fact that costs related to thread creation/management overhead actually renders our runtime worse than if we had used a sequential approach. We have also seen in class that parallel solutions generally tend to underperform in comparison to sequential ones when the input problem is small.