

Code generation

Prompt:

Generate an efficient Java code for matrix multiplication using multi-threading. Please optimize for performance and provide insights into the threading strategy you employ. Consider factors such as thread synchronization, load balancing, and any potential bottlenecks. Additionally, justify yourself choosing this particular method.

Tested (edge) cases:

Small Matrices - Large Matrices (1000x1000) - Empty Matrices - Identity Matrices - Sparse Matrices (matrices where a significant portion of elements is zero) - Matrices with Negative Numbers.

GPT-4

```
import java.util.ArrayList;
import java.util.List;

public class MatrixMultiplier {

    private final int[][] A;
    private final int[][] B;
    private final int[][] C;
    private final int numberOfThreads;

    public MatrixMultiplier(int[][] A, int[][] B, int numberOfThreads) {
        if (A[0].length != B.length) {
            throw new IllegalArgumentException("Matrices can't be multiplied");
        }
        this.A = A;
        this.B = B;
        this.C = new int[A.length][B[0].length];
        this.numberOfThreads = numberOfThreads;
    }

    public int[][] multiply() throws InterruptedException {
        List<Thread> threads = new ArrayList<>();
        int rowsPerThread = A.length / numberOfThreads;

        for (int threadNum = 0; threadNum < numberOfThreads; threadNum++) {
            int startRow = threadNum * rowsPerThread;
            int endRow = (threadNum == numberOfThreads - 1) ? A.length : startRow + rowsPerThread;

            Thread thread = new Thread(new MultiplierTask(A, B, C, startRow, endRow));
            threads.add(thread);
            thread.start();
        }

        for (Thread thread : threads) {
            thread.join();
        }

        return C;
    }
}
```

```

private static class MultiplierTask implements Runnable {
    private final int[][] A;
    private final int[][] B;
    private final int[][] C;
    private final int startRow;
    private final int endRow;

    public MultiplierTask(int[][] A, int[][] B, int[][] C, int startRow, int endRow) {
        this.A = A;
        this.B = B;
        this.C = C;
        this.startRow = startRow;
        this.endRow = endRow;
    }

    @Override
    public void run() {
        for (int i = startRow; i < endRow; i++) {
            for (int j = 0; j < B[0].length; j++) {
                for (int k = 0; k < A[0].length; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    int[][] A = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    int[][] B = {
        {7, 8, 9},
        {10, 11, 12}
    };

    MatrixMultiplier multiplier = new MatrixMultiplier(A, B, 2);
    int[][] result = multiplier.multiply();

    for (int i = 0; i < result.length; i++) {
        for (int j = 0; j < result[0].length; j++) {
            System.out.print(result[i][j] + " ");
        }
        System.out.println();
    }
}

```

GPT-3.5

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class MatrixMultiplier {

```

```

private static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

public static void main(String[] args) {
    int[][] matrixA = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int[][] matrixB = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

    int[][] result = multiplyMatrices(matrixA, matrixB);

    // Print the result matrix
    for (int i = 0; i < result.length; i++) {
        for (int j = 0; j < result[0].length; j++) {
            System.out.print(result[i][j] + " ");
        }
        System.out.println();
    }
}

public static int[][] multiplyMatrices(int[][] matrixA, int[][] matrixB) {
    int numRowsA = matrixA.length;
    int numColsA = matrixA[0].length;
    int numColsB = matrixB[0].length;

    int[][] result = new int[numRowsA][numColsB];

    ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);

    for (int i = 0; i < numRowsA; i++) {
        for (int j = 0; j < numColsB; j++) {
            executor.execute(new MatrixMultiplierTask(matrixA, matrixB, result, i, j, numColsA));
        }
    }

    executor.shutdown();

    try {
        executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    return result;
}

static class MatrixMultiplierTask implements Runnable {
    private final int[][] matrixA;
    private final int[][] matrixB;
    private final int[][] result;
    private final int row;
    private final int col;
    private final int numColsA;

    public MatrixMultiplierTask(int[][] matrixA, int[][] matrixB, int[][] result, int row, int col, int
numColsA) {
        this.matrixA = matrixA;
        this.matrixB = matrixB;
        this.result = result;
        this.row = row;
        this.col = col;
    }
}

```

```

        this.numColsA = numColsA;
    }

    @Override
    public void run() {
        for (int k = 0; k < numColsA; k++) {
            result[row][col] += matrixA[row][k] * matrixB[k][col];
        }
    }
}
}
}

```

BingAI

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class MatrixMultiplier {
    private static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

    public static int[][] multiply(int[][] matrixA, int[][] matrixB) throws InterruptedException {
        int rowsA = matrixA.length;
        int colsA = matrixA[0].length;
        int colsB = matrixB[0].length;

        int[][] result = new int[rowsA][colsB];

        ExecutorService executorService = Executors.newFixedThreadPool(NUM_THREADS);

        for (int i = 0; i < rowsA; i++) {
            for (int j = 0; j < colsB; j++) {
                executorService.execute(new MatrixMultiplierTask(matrixA, matrixB, result, i, j));
            }
        }

        executorService.shutdown();
        executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);

        return result;
    }

    private static class MatrixMultiplierTask implements Runnable {
        private final int[][] matrixA;
        private final int[][] matrixB;
        private final int[][] result;
        private final int row;
        private final int col;

        public MatrixMultiplierTask(int[][] matrixA, int[][] matrixB, int[][] result, int row, int col) {
            this.matrixA = matrixA;
            this.matrixB = matrixB;
            this.result = result;
            this.row = row;
            this.col = col;
        }

        @Override

```

```

    public void run() {
        int sum = 0;
        for (int k = 0; k < matrixA[0].length; k++) {
            sum += matrixA[row][k] * matrixB[k][col];
        }
        result[row][col] = sum;
    }
}
}

```

You

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class MatrixMultiplier {
    private static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

    public static int[][] multiply(int[][] matrixA, int[][] matrixB) throws InterruptedException {
        int rowsA = matrixA.length;
        int colsA = matrixA[0].length;
        int colsB = matrixB[0].length;

        int[][] result = new int[rowsA][colsB];

        ExecutorService executorService = Executors.newFixedThreadPool(NUM_THREADS);

        for (int i = 0; i < rowsA; i++) {
            for (int j = 0; j < colsB; j++) {
                executorService.execute(new MatrixMultiplierTask(matrixA, matrixB, result, i, j));
            }
        }

        executorService.shutdown();
        executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);

        return result;
    }

    private static class MatrixMultiplierTask implements Runnable {
        private final int[][] matrixA;
        private final int[][] matrixB;
        private final int[][] result;
        private final int row;
        private final int col;

        public MatrixMultiplierTask(int[][] matrixA, int[][] matrixB, int[][] result, int row, int col) {
            this.matrixA = matrixA;
            this.matrixB = matrixB;
            this.result = result;
            this.row = row;
            this.col = col;
        }

        @Override
        public void run() {
            int sum = 0;

```

```

        for (int k = 0; k < matrixA[0].length; k++) {
            sum += matrixA[row][k] * matrixB[k][col];
        }
        result[row][col] = sum;
    }
}
}

```

Google Bard

```

public class MatrixMultiplier {

    private static final int NUM_THREADS = Runtime.getRuntime().availableProcessors();

    public static double[][] multiply(double[][] matrix1, double[][] matrix2) {
        double[][] resultMatrix = new double[matrix1.length][matrix2[0].length];

        // Create an array of threads to perform matrix multiplication
        Thread[] threads = new Thread[NUM_THREADS];

        // Divide the rows of the result matrix into segments based on the number of threads
        int rowsPerSegment = resultMatrix.length / NUM_THREADS;

        // Create a task for each thread to multiply a segment of the result matrix
        for (int i = 0; i < NUM_THREADS; i++) {
            int startRow = i * rowsPerSegment;
            int endRow = startRow + rowsPerSegment - 1;

            threads[i] = new Thread(new MatrixMultiplicationTask(matrix1, matrix2, resultMatrix, startRow,
endRow));
        }

        // Start all of the threads
        for (Thread thread : threads) {
            thread.start();
        }

        // Wait for all of the threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        return resultMatrix;
    }

    private static class MatrixMultiplicationTask implements Runnable {

        private final double[][] matrix1;
        private final double[][] matrix2;
        private final double[][] resultMatrix;
        private final int startRow;
        private final int endRow;
    }
}

```

```

    public MatrixMultiplicationTask(double[][] matrix1, double[][] matrix2, double[][] resultMatrix, int
startRow, int endRow) {
        this.matrix1 = matrix1;
        this.matrix2 = matrix2;
        this.resultMatrix = resultMatrix;
        this.startRow = startRow;
        this.endRow = endRow;
    }

    @Override
    public void run() {
        for (int i = startRow; i <= endRow; i++) {
            for (int j = 0; j < resultMatrix[0].length; j++) {
                for (int k = 0; k < matrix1[0].length; k++) {
                    resultMatrix[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
    }
}

```