ECSE 420 - Parallel Computing

Assignment 1

Arian Omidi - 260835976
Namdar Nejad - 260893536

February 10, 2022

Question 1

1.1

We essentially coded the method we use as "humans" to multiply matrices. We go row by row in matrix a and multiply each row by each column of b and write the results. The running time is $O(n^3)$. We have more explanations on the code as comments.

1.2

Our parallel method uses the same logic as the sequential method, but here for each row we create a new thread and let the thread take care of the multiplication. We have more explanations on the code as comments.

1.3

The method is called *multiplyTimed(double[][] a, double[][] b)*. It'll multiply matrix a and b using both methods and print out the execution time for each method.

1.4

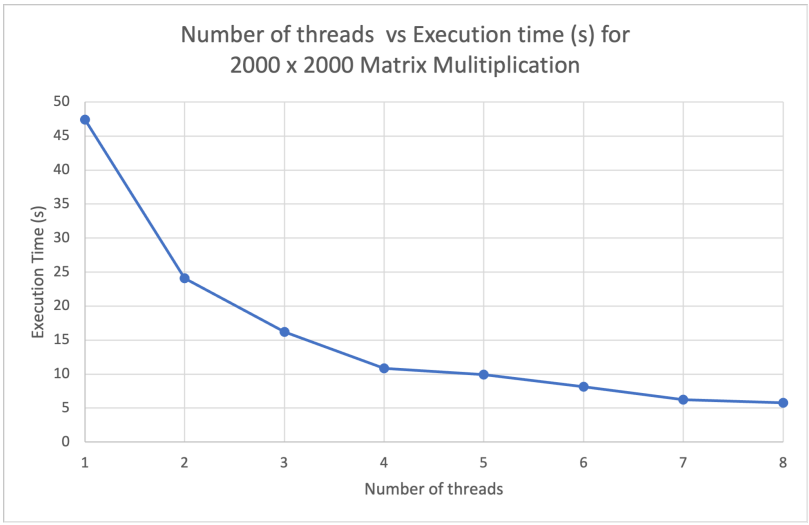

Figure 1: Number of threads vs. Execution time (s)

1.5

Figure 2: Matrix size vs. Execution time (s)

1.6

For 1.4, we can see that there is a clear advantage in using multiple threads in completing your task. However, as seen in the graph when the number of threads exceeds the number of cores in the machine the drop in running time slowly plateau. Using too many threads may diminish the speed up gained, this might be due to the complicated task of managing the multiple threads. For 1.5, it's clear that for smaller matrices the sequential method works better, however, when we go above 300*300 the parallel methods start to perform better. This is probably due to the fact that creating and managing threads is a complicated and rather time-consuming task which will only help out when we have a big task to split between the threads that have been created, this is not the case for small matrices bit as the matrix sizes increase the overhead of handling threads start to pay off.

Question 2

2.1

A deadlock accurs when a set of threads need some resources that are being used by other threads to complete their task, and the other threads are also waiting for other resources to be freed to complete their task. So essentially when a thread is waiting on another thread to complete its task and vice-versa. In this program we have demonstrated a deadlock by creating two threads (t1 & t2) and two resources (r1 & r2). Both threads need both resources to complete their task, each thread acquires one resources first and waits for the other recourse to be freed up by the other thread. t1 acquires r1 first and waits on r2, on the other hand t2 acquires r2 first and waits on r1, this is how we reach a deadlock.

2.2

Deadlocks can't really be absolutely avoided, but there are some stuff we can do to reduce the chances of a deadlock. For example, we could try to avoid nested locks and not distribute one lock between multiple threads, which is the cause of the deadlock in our code. Moreover, we should avoid unnecessary locks and only use locks when it's absolutely necessary and use a safer alternative when we can. Finally, we could use deadlock detection to detect deadlock and fix the problem by freeing up some resources.

Question 3

3.1

See the *DiningPhilosophersDeadlock.java* code below in the appendix.

3.2

To avoid the deadlock scenario, we refactored our code such that the first philosopher will always attempt to pick up the right chopstick first while the remaining philosophers will attempt to pick up the left chopstick first. This is guaranteed to avoid deadlock as we can never enter a scenario where every philosopher is holding only one chopstick. As if the first philosopher has the chopstick to his right, the last philosopher cannot start eating as the chopstick its left is in use, and thus will wait until the first philosopher if finished eating before they pick up the chopstick. Similarly, if the last philosopher has the chopstick to his left, the first philosopher cannot start eating as the chopstick on their right is in use, and will wait for the last philosopher to finish eating.

3.3

To ensure that no philosopher starves, we used *ReentrantLocks* with a fair entrance policy, such that the philosopher who has been waiting the longest to eat is granted priority. We also refactored our code such that if a philosopher cannot obtain both chopsticks, it will drop both and wait until both are available.

Question 4

4.1

$$S_{max} = \lim_{n\to\infty} \frac{1}{1-p+\frac{p}{n}} = \lim_{n\to\infty} \frac{1}{0.4+\frac{0.6}{n}} = \frac{1}{0.4} = 2.5$$

4.2

$$S_n = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{0.3+\frac{0.7}{n}}$$

$$S'_n = \frac{1}{0.3k+\frac{1-0.3k}{n}} > 2S_n = \frac{2}{0.3+\frac{0.7}{n}}$$

$$0.3 + \frac{0.7}{n} > 0.6k + \frac{2 - 0.6k}{n}$$
$$0.3n + 0.7 > 0.6nk + 2 - 0.6k$$
$$0.3n - 1.3 > 0.6k(n-1)$$
$$\frac{3n-13}{6(n-1)} > k$$

Therefore, to double the speedup S_n, we must replace the sequential part with an improved version that decreases the sequential time percentage by a factor of k, where $k < \frac{3n-13}{6(n-1)}$.

4.3

$$S'_n = \frac{1}{\frac{s}{3} + \frac{1-\frac{s}{3}}{n}} = 2S_n = \frac{2}{s + \frac{1-s}{n}}$$
$$\frac{2}{3}s + \frac{2 - \frac{2}{3}s}{n} = s + \frac{1-s}{n}$$
$$\frac{2}{3}ns + 2 - \frac{2}{3}s = ns + 1 - s$$
$$1 = \frac{1}{3}ns - \frac{1}{3}s$$
$$s = \frac{3}{n-1}$$

The fraction of the overall execution time the sequential part account for is equal to $\frac{3}{n-1}$.

Appendix

MatrixMultiplication.java

```java
package ca.mcgill.ecse420.a1;

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

public class MatrixMultiplication {

    private static final int NUMBER_THREADS = 4;
    private static final int MATRIX_SIZE = 1000;

    public static void main(String[] args) {

        // Generate two random matrices, same size
        double[][] a = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
        double[][] b = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);

        multiplyTimed(a, b);
    }

    /**
     * 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) {

        // check if the matrix dimensions are correct
        // the number of columns in a should be equal to the number of rows in b
        if (a[0].length != b.length) {
            try {
                throw new Exception("Invalid input");
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }

        /*
        aRows is the number of rows in matrix a
        bRows is the number of rows in matrix b
        aColumns is the number of columns in matrix a which is equal to the number of columns in matrix b
        prod is the product matrix
         */
        int aRows = a.length;
        int bColumns = b[0].length;
        int aColumns = a[0].length;
        double[][] prod = new double[aRows][bColumns];

        /*
        multiply each row of matrix a with each column of matrix b and store the results in prod, as we
            would on paper
         */
        for (int i = 0; i < aRows; i++) {
            for (int j = 0; j < bColumns; j++) {
                for (int k = 0; k < aColumns; k++) {
                    prod[i][j] += a[i][k] * b[k][j];
                }
            }
        }

        return prod;
    }

    /**
     * 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) {

        // check if the matrix dimensions are correct
        // the number of columns in a should be equal to the number of rows in b
        if (a[0].length != b.length) {
```

```
            try {
                throw new Exception("Invalid input");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        /*
        aRows is the number of rows in matrix a
        bRows is the number of rows in matrix b
        aColumns is the number of columns in matrix a which is equal to the number of columns in matrix b
        prod is the product matrix
         */
        int aRows = a.length;
        int bColumns = b[0].length;
        double[][] prod = new double[aRows][bColumns];

        // Creating threads
        ExecutorService executor = Executors.newFixedThreadPool(NUMBER_THREADS);

        /*
        Create a thread for each row of a that is going to be multiplies, each thread will take take of
            one row multiplication
         */
        for (int i = 0; i < aRows; i++) {
            executor.execute(new RowMultiply(prod, a, b, i));
        }

        executor.shutdown();

        // waiting for threads to complete their execution
        try {
            executor.awaitTermination(MATRIX_SIZE, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return prod;
    }


    public static class RowMultiply implements Runnable {
```

```
    private double[][] prod, a, b;
    private final int row;

    /**
     * Constructor for the RowMultiply class
     *
     * @param prod product matrix till now
     * @param a   first matrix
     * @param b   second matrix
     * @param row row number that is being multiplies by the current thread
     */
    public RowMultiply(double[][] prod, double[][] a, double[][] b, int row) {
        this.prod = prod;
        this.a = a;
        this.b = b;
        this.row = row;
    }

    /**
     * Multiplies the given row and inputs the result in the prod matrix
     */
    public void run() {

        for (int i = 0; i < b[0].length; i++) {
            prod[row][i] = 0;
            for (int j = 0; j < a[row].length; j++) {
                prod[row][i] += a[row][j] * b[j][i];
            }
        }
    }
}


/**
 * multiplies the matracies by using the sequentialMultiplyMatrix and parallelMultiplyMatrix method
 * prints the time it tool to complete the task using each method.
 *
 * @param a first matrix to be multiplied
 * @param b second matrix to be multiplied
 */
public static void multiplyTimed(double[][] a, double[][] b) {
```

```
        Date start, end;
        double[][] res;

        start = new Date();
        res = sequentialMultiplyMatrix(a, b);
        end = new Date();
        System.out.println("\nsequential multiplication (milli seconds): " + (end.getTime() -
            start.getTime()));

        start = new Date();
        res = parallelMultiplyMatrix(a, b);
        end = new Date();
        System.out.println("\nparallel multiplication (milli seconds): " + (end.getTime() -
            start.getTime()));
    }

    /**
     * Populates a matrix of given size with randomly generated integers between 0-10.
     *
     * @param numRows number of rows
     * @param numCols number of cols
     * @return matrix
     */
    private static double[][] generateRandomMatrix(int numRows, int numCols) {
        double matrix[][] = new double[numRows][numCols];
        for (int row = 0; row < numRows; row++) {
            for (int col = 0; col < numCols; col++) {
                matrix[row][col] = (double) ((int) (Math.random() * 10.0));
            }
        }
        return matrix;
    }

}
```

Deadlock.java

```
package ca.mcgill.ecse420.a1;

public class Deadlock {
```

```
    public static void main(String[] args) {

        // The 2 Resources
        String r1 = "r1";
        String r2 = "r2";

        // The 2 Threads
        DeadlockThread t1 = new DeadlockThread(r1, r2, "T1");
        DeadlockThread t2 = new DeadlockThread(r2, r1, "T2");

        // Start the Threads
        t1.start();
        t2.start();
    }

    public static class DeadlockThread extends Thread {
        final String r1;
        final String r2;
        final String threadName;

        public DeadlockThread(String r1, String r2, String threadName) {
            this.r1 = r1;
            this.r2 = r2;
            this.threadName = threadName;
        }

        /**
         * Takes r1 first and waits on r2
         */
        public void run() {
            // takes r1
            synchronized (r1) {
                System.out.println(threadName + ": Locked " + r1);
                try {
                    // delay to make sure we reach a deadlock, to make sure r1 is acquired by the thread
                    Thread.sleep(10);
                } catch (Exception ex) {
                }
                // waits on r2
                synchronized (r2) {
                    System.out.println(threadName + ": Locked " + r2);
                }
```

```
            }
            System.out.println("Completed " + threadName);
        }
    }
}
```

DiningPhilosophersDeadlock.java

```
package ca.mcgill.ecse420.a1;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DiningPhilosophersDeadlock {

    public static void main(String[] args) {

        int numberOfPhilosophers = 5;
        Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
        Lock[] chopsticks = new Lock[numberOfPhilosophers];


        // init locks
        for (int i = 0; i < numberOfPhilosophers; i++) {
            chopsticks[i] = new ReentrantLock();
        }

        // init philosophers threads
        ExecutorService executor = Executors.newCachedThreadPool();

        for (int i = 0; i < numberOfPhilosophers; i++) {
            Lock leftChopstick = chopsticks[i];
            Lock rightChopstick = chopsticks[(i + 1) % 5];
            executor.execute(new Philosopher(leftChopstick, rightChopstick));
        }

        executor.shutdown();
    }
```

```
    public static class Philosopher implements Runnable {

        final Lock leftChopstick;
        final Lock rightChopstick;

        public Philosopher(Lock leftChopstick, Lock rightChopstick) {
            this.leftChopstick = leftChopstick;
            this.rightChopstick = rightChopstick;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    // think
                    System.out.format("%s: Thinking\n", Thread.currentThread().getName());
                    Thread.sleep(((int) (Math.random() * 100)));

                    // try to eat
                    synchronized (leftChopstick) {
                        System.out.format("%s: Picked up left chopstick\n",
                             Thread.currentThread().getName());
                        Thread.sleep(((int) (Math.random() * 100)));
                        synchronized (rightChopstick) {
                            System.out.format("%s: Picked up right chopstick\n",
                                Thread.currentThread().getName());
                            Thread.sleep(((int) (Math.random() * 100)));
                        }
                    }
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }


    }

}
```

DiningPhilosophersNoDeadlock.java

```java
package ca.mcgill.ecse420.a1;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DiningPhilosophersNoDeadlock {

    public static void main(String[] args) {

        int numberOfPhilosophers = 5;
        Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
        Lock[] chopsticks = new Lock[numberOfPhilosophers];


        // init locks
        for (int i = 0; i < numberOfPhilosophers; i++) {
            chopsticks[i] = new ReentrantLock();
        }

        // init philosophers threads
        ExecutorService executor = Executors.newFixedThreadPool(numberOfPhilosophers);

        for (int i = 0; i < numberOfPhilosophers; i++) {
            Lock firstChopstick, secondChopstick;
            // the first philosopher will pick up his right chopstick first
            if (i == 0) {
                firstChopstick = chopsticks[(i + 1) % 5];
                secondChopstick = chopsticks[i];
            } else {
                firstChopstick = chopsticks[i];
                secondChopstick = chopsticks[(i + 1) % 5];
            }

            executor.execute(new Philosopher(firstChopstick, secondChopstick));
        }

        executor.shutdown();
    }
```

```
    public static class Philosopher implements Runnable {

        final Lock leftChopstick;
        final Lock rightChopstick;

        public Philosopher(Lock leftChopstick, Lock rightChopstick) {
            this.leftChopstick = leftChopstick;
            this.rightChopstick = rightChopstick;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    // think
                    System.out.format("%s: Thinking\n", Thread.currentThread().getName());
                    Thread.sleep(((int) (Math.random() * 100)));

                    // try to eat
                    synchronized (leftChopstick) {
                        System.out.format("%s: Picked up left chopstick\n",
                            Thread.currentThread().getName());
                        Thread.sleep(((int) (Math.random() * 100)));
                        synchronized (rightChopstick) {
                            System.out.format("%s: Picked up right chopstick\n",
                                Thread.currentThread().getName());
                            Thread.sleep(((int) (Math.random() * 100)));
                        }
                    }
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }


    }

}
```

DiningPhilosophersNoStarvation.java

```
package ca.mcgill.ecse420.a1;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class DiningPhilosophersNoStarvation {

    public static void main(String[] args) {

        int numberOfPhilosophers = 5;
        Lock[] chopsticks = new Lock[numberOfPhilosophers];

        // init locks
        for (int i = 0; i < numberOfPhilosophers; i++) {
            // fair lock policy
            chopsticks[i] = new ReentrantLock(true);
        }

        // init philosophers threads
        ExecutorService executor = Executors.newFixedThreadPool(numberOfPhilosophers);

        for (int i = 0; i < numberOfPhilosophers; i++) {
            Lock firstChopstick, secondChopstick;
            // the first philosopher will pick up his right chopstick first
            if (i == 0) {
                firstChopstick = chopsticks[(i + 1) % 5];
                secondChopstick = chopsticks[i];
            } else {
                firstChopstick = chopsticks[i];
                secondChopstick = chopsticks[(i + 1) % 5];
            }

            executor.execute(new Philosopher(firstChopstick, secondChopstick));
        }

        // shutdown threads
        executor.shutdown();
```

```
}

public static class Philosopher implements Runnable {

    private static final long RUN_TIME = 20; // seconds

    private final Lock leftChopstick;
    private final Lock rightChopstick;

    public Philosopher(Lock leftChopstick, Lock rightChopstick) {
        this.leftChopstick = leftChopstick;
        this.rightChopstick = rightChopstick;
    }

    @Override
    public void run() {
        try {
            long startTime = System.currentTimeMillis();
            long eatingTime = 0;
            long waitingTime = 0;

            // run for 60 secs
            while (System.currentTimeMillis() - startTime < RUN_TIME * 1000) {
                // think
                System.out.format("%s: Thinking\n", Thread.currentThread().getName());
                Thread.sleep(((int) (Math.random() * 100)));

                // eat
                long startWait = System.currentTimeMillis();
                // try to pick up left chopstick if available (keep fairness policy)
                if (leftChopstick.tryLock(0, TimeUnit.SECONDS)) {
                    System.out.format("%s: Picked up left chopstick\n",
                        Thread.currentThread().getName());
                    Thread.sleep(((int) (Math.random() * 100)));

                    // try to pick up right chopstick if available (keep fairness policy)
                    if (rightChopstick.tryLock(0, TimeUnit.SECONDS)) {
                        System.out.format("%s: Picked up right chopstick\n",
                            Thread.currentThread().getName());
                        waitingTime += System.currentTimeMillis() - startWait;

                        long startEating = System.currentTimeMillis();
```

```
                    Thread.sleep(((int) (Math.random() * 100)));
                    eatingTime += System.currentTimeMillis() - startEating;

                    rightChopstick.unlock();
                }

                leftChopstick.unlock();
            }
        }

        System.out.format("%s: Waited = %d, Ate = %d\n", Thread.currentThread().getName(),
            waitingTime, eatingTime);
      } catch (Exception e) {
        leftChopstick.unlock();
        rightChopstick.unlock();
      }
    }


  }

}
```