Parallel Computing.
Assignment Report 1

Estelle Saab - 261027752
Rafi Latif - 261120830
Group 37
October 1st,2025

ECSE 420
Parallel Computing

Question Explanations

1) Matrix multiplication

1.1) Matrix Multiplication in Sequential

Matrix multiplication involves calculating the dot product of each row of one matrix with each column from a second matrix. The result of this dot product for a particular row and column is stored in the corresponding cell of the resulting matrix.

```
------------------------Matrix a---------------------------
6.0 8.0
9.0 7.0
------------------------Matrix b---------------------------
7.0 1.0
2.0 5.0
------------------------Sequential-------------------------
58.0 46.0
77.0 44.0
```

To validate the method produces the correct results we calculate it manually and obtain the same results:

$$\begin{bmatrix} 6 & 8 \\ 9 & 7 \end{bmatrix} \cdot \begin{bmatrix} 7 & 1 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 6\times7+8\times2 & 6\times1+8\times5 \\ 9\times7+7\times2 & 9\times1+5\times7 \end{bmatrix}$$

$$= \begin{bmatrix} 58 & 46 \\ 77 & 44 \end{bmatrix}$$

1.2) Matrix multiplication in Parallel

```
------------------------Parallel---------------------------
188.0 225.0 229.0 336.0 148.0 213.0 291.0 324.0 288.0 216.0
125.0 105.0 95.0 199.0 52.0 110.0 148.0 188.0 202.0 165.0
203.0 197.0 179.0 279.0 133.0 134.0 237.0 227.0 281.0 174.0
219.0 195.0 203.0 307.0 227.0 189.0 217.0 316.0 193.0 167.0
220.0 209.0 194.0 317.0 154.0 190.0 241.0 317.0 267.0 202.0
253.0 253.0 205.0 334.0 201.0 189.0 239.0 308.0 247.0 211.0
198.0 186.0 169.0 290.0 143.0 127.0 217.0 261.0 282.0 157.0
127.0 142.0 146.0 224.0 100.0 130.0 211.0 220.0 221.0 125.0
206.0 136.0 133.0 243.0 128.0 153.0 216.0 291.0 216.0 163.0
95.0 80.0 73.0 139.0 48.0 47.0 108.0 140.0 151.0 85.0
------------------------Sequential-------------------------
188.0 225.0 229.0 336.0 148.0 213.0 291.0 324.0 288.0 216.0
125.0 105.0 95.0 199.0 52.0 110.0 148.0 188.0 202.0 165.0
203.0 197.0 179.0 279.0 133.0 134.0 237.0 227.0 281.0 174.0
219.0 195.0 203.0 307.0 227.0 189.0 217.0 316.0 193.0 167.0
220.0 209.0 194.0 317.0 154.0 190.0 241.0 317.0 267.0 202.0
253.0 253.0 205.0 334.0 201.0 189.0 239.0 308.0 247.0 211.0
198.0 186.0 169.0 290.0 143.0 127.0 217.0 261.0 282.0 157.0
127.0 142.0 146.0 224.0 100.0 130.0 211.0 220.0 221.0 125.0
206.0 136.0 133.0 243.0 128.0 153.0 216.0 291.0 216.0 163.0
95.0 80.0 73.0 139.0 48.0 47.0 108.0 140.0 151.0 85.0
```

We implemented the parallelMultiplyMatrix(double[][] a, double[][] b, int numThreads) method using an ExecutorService with a fixed thread pool as shown in *A.1.2: Parallel Matrix Multiplication Function*. Each thread is assigned a row of the result matrix, calculating all the elements in that row by performing the dot product of the corresponding row from matrix a and all columns of matrix b.

Since each thread only writes to its assigned row, there is no need for synchronization, ensuring thread safety. After all tasks are completed and the executor is shut down and the final result matrix is returned. The correctness of the parallel method is validated by comparing the parallel results with those from the sequential method.

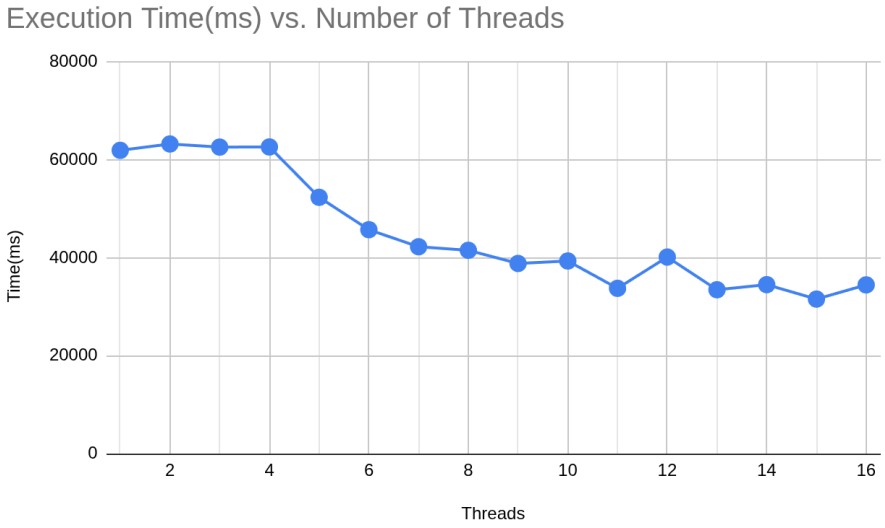
1.3) Execution time for both sequential and parallel matrix multiplication

We added a method to measure the execution time of both sequential and parallel matrix multiplication. The method calculates the time taken for each approach by recording the start and end times using System.currentTimeMillis() for both methods, and returns the differences in an array, as shown in *A.1.5: Performance Measuring Helper Functions*.

We tested the method, and the execution times matched the expected behavior. The parallel method consistently showed faster performance for larger matrices, as it divides the work across multiple threads. The results were validated by comparing the parallel output with the sequential output, ensuring both methods produced the same result.

1.4) Speed-up obtained vs. the number of threads used

We created a function that is iteratively called parallelMultiplyMatrix(double[][] a, double[][] b, int numThreads) using an increasing number of threads (up to the max number of available processors on the system), and recorded the results in a csv file. Due to limitations in available system memory, we were only able to test matrices up to a size of 3000x3000 instead of the specified 4000x4000. The program encountered heap memory issues during runtime when attempting to handle larger matrices.



The results for matrix sizes up to 3000x3000, as shown in the data, indicate a clear trend: as the number of threads increases, the execution time decreases, showcasing the benefits of parallelization. We found that the execution reached its lowest point at 15 threads. Initially, when the number of threads were low, the execution time was significantly higher. However, after a certain point, the improvement in execution time slowed down, suggesting diminishing returns as the number of threads increased beyond a certain threshold.

Although the test was limited to 3000x3000 matrices, it is likely that larger matrices (e.g., 4000x4000) would follow a similar pattern, with a potential for further heap memory issues unless optimizations are applied.

1.5) Vary the matrices size

In this task, we varied the matrix sizes as specified (100x100, 200x200, 500x500, 1000x1000, 2000x2000, 3000x3000) to analyze the execution time of both the sequential and parallel matrix multiplication methods. We used 15 threads for parallel matrix multiplication since it achieved the smallest execution time from Q1.4.

The plot indicates that as matrix size increases, the time taken for both sequential and parallel methods also increases, with parallelization showing significant speed-up in comparison to the sequential method. However, for smaller matrices (e.g., 100x100, 200x200), the parallel method does not offer a significant advantage in terms of execution time compared to the sequential method.

Although the parallel method seems to provide an advantage in terms of execution time over the sequential method, even its speed-up starts to diminish as matrix size increases.

Due to memory constraints, the test was conducted up to matrix size 3000x3000, and larger matrix sizes of 4000x4000 could not be tested because of heap memory limitations.

1.6) Shape and possible reasons for the observed behavior

Graph 1: Time vs. Number of Threads (1.4)
In the first graph, the execution time significantly decreases as the number of threads increases, with a noticeable drop between 4 and 5 threads. This is expected in parallel computing, where initially, each thread can handle separate tasks, reducing the overall execution time. However, beyond a certain threshold (around 11 threads in this case), the reduction in time becomes minimal, and the execution time stabilizes. This behavior is likely due to the overhead involved in managing threads, such as synchronization and communication between threads. Eventually, the system reaches a point where the cost of managing additional threads outweighs the performance benefits gained from parallelism.

Graph 2: Execution Time as a Function of Matrix Size (1.5)
In the second graph, we observe that both sequential and parallel execution times increase with matrix size. The sequential execution time grows exponentially as the matrix size increases, as expected. However, the parallel method shows a much slower rate of increase in comparison, especially at larger matrix sizes. This demonstrates the efficiency of parallelization, where larger matrices benefit more from parallel execution due to better utilization of available threads.

For smaller matrices (up to 500x500), the difference between sequential and parallel execution is less significant, as the overhead of thread management becomes more apparent. However, as the matrix size increases (1000x1000 and beyond), the parallel execution shows a substantial advantage, confirming that larger problems are more suited for parallel processing.

Conclusion:
Both graphs clearly show the effectiveness of parallel computing, but also highlight the limitations when too many threads are used, and thread management overhead begins to dominate.

2) Deadlock

2.1) Conditions and Consequences

```
Thread 1: Locked Resource A
Thread 2: Locked Resource B
Thread 2: Waiting for Resource A
Thread 1: Waiting for Resource B
```

Conditions for Deadlock:

- Deadlock occurs when the following four conditions are met:
- Mutual Exclusion: Resources are only held by one thread at a time.
- Hold and Wait: A thread holds one resource and waits for another.
- No Preemption: Resources cannot be forcibly taken from a thread.
- Circular Wait: A cycle of threads waiting for each other exists.

Consequences of Deadlock:

- Program Stalling: Threads are blocked and the program freezes.
- Resource Wastage: Resources are held but not used.
- System Inefficiency: Deadlock reduces overall system performance.

Example Program:
We created a class called DeadlockExample in which we demonstrate how deadlock can occur when accessing shared resources, as shown in *B.1.0: Deadlock Example Source Code.* The program creates the following scenario:

- Thread 1 locks resourceA and waits for resourceB.
- Thread 2 locks resourceB and waits for resourceA.

This creates a circular wait, where both threads are stuck waiting for each other, resulting in a deadlock.

2.2) Solutions

General Design Solutions:

- Resource Ordering: Ensure that all threads acquire resources in a consistent order. For example, always acquire resourceA before resourceB. This prevents circular wait, as threads will never wait for each other in a cycle.
- Timeouts: Implement timeouts when acquiring locks. If a thread cannot acquire all required resources within a specified time, it releases any resources it holds and retries. This prevents threads from being indefinitely blocked.

- Deadlock Detection and Recovery: Continuously monitor the system for deadlocks. If one is detected, take corrective action, such as aborting or rolling back transactions. This is more complex but can resolve deadlocks dynamically.
- Lock Hierarchy: Use a predefined hierarchy of resources, where threads always request resources in the same order based on priority. This eliminates the possibility of a circular wait.

Solution to the Example Program:
The Resource Ordering solution can be applied to avoid deadlock in this example. By ensuring that both threads acquire the locks in the same order (e.g., always lock resourceA first, then resourceB), the circular wait condition is eliminated. Thus, this solution would prevent deadlock in the given code.

3) Dining Philosophers

3.1) Deadlocked Philosophers

In the Dining Philosophers problem, the philosophers are represented by threads that compete for the same chopsticks. The key challenge is to ensure that two philosophers do not hold the same chopstick simultaneously and prevent deadlock.

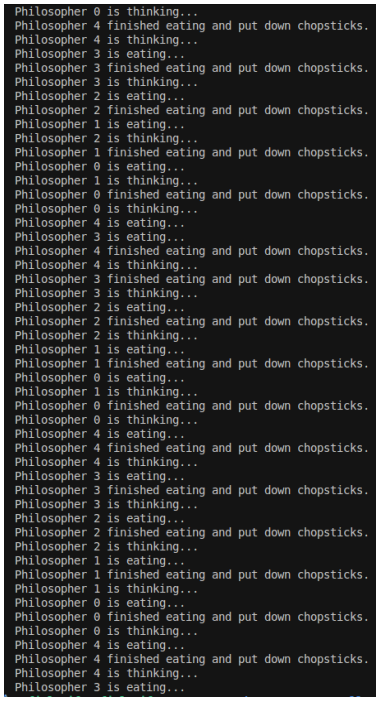
Ensuring Philosophers Cannot Hold the Same Chopstick:
- Each chopstick is controlled by a semaphore (chopsticks[i]), ensuring that only one philosopher can pick it up at a time.
- The philosophers must acquire two semaphores to begin eating: the one for their left chopstick (chopsticks[id]) and the one for their right chopstick (chopsticks[(id + 1) % N]).
- By using semaphores, the system guarantees mutual exclusion, meaning that only one philosopher can hold a chopstick at any given time.

How Deadlock Occurs in This Program:
- Deadlock can occur if all philosophers acquire their left chopstick before attempting to acquire the right chopstick. This creates a situation where each philosopher holds one chopstick and waits for the other, causing all threads to be blocked indefinitely.

3.2) Deadlock-Free Philosophers

To avoid deadlock in the Dining Philosophers problem, we introduce a queue that limits the number of philosophers able to pick up chopsticks. By limiting the number of philosophers that can attempt to eat simultaneously (using a queue semaphore), we prevent the circular wait condition that leads to deadlock. As shown in *C.1.0: Dining Philosopher Deadlock and Starvation Free Code,* we use a semaphore queue containing N-1 locks to ensure there are only ever N-1 philosophers attempting to hold N chopsticks.

Is Starvation Possible?
Starvation refers to a situation where certain are never able to execute due to other threads cutting ahead. In terms of the dining philosophers, starvation would mean a philosopher never gets a chance to eat because other philosophers keep eating. However, in this modified program, starvation is unlikely. Since the semaphore queue also enforces FIFO, it ensures a fair ordering where all philosophers are eventually allowed to pick up chopsticks, preventing indefinite waiting. Thus, starvation is avoided in this system through Resource Ordering.



4) Amdahl's Law

4.1)

According to Amdahl's Law, the overall speedup of a program on n processors depends on the fraction of the program that is sequential versus parallelizable. In this case, 40% of the program is sequential (f=0.40), meaning that 60% of the program can be parallelized (1−f=0.60). The expression for speedup is:

$S(n)= 1 / (f+ ((1-f) / n)) = 1 / (0.4 + 0.6/n)$

This formula shows how the performance improvement depends on the number of processors. As the number of processors grows very large, the term 0.6 / n approaches zero, and the speedup approaches:

$\text{Lim } (n \rightarrow \infty) S (n) = 1 / 0.4 = 2.5$

Therefore, the maximum theoretical speedup achievable for this program is 2.5x, regardless of how many processors are added. This result highlights the limiting effect of the sequential portion of the program: even with unlimited processors, the execution time cannot be reduced beyond this bound.

4.2)

$$\frac{1}{\frac{0.3}{K} + \frac{1 - \frac{0.3}{K}}{n}} > 2 \times \frac{1}{0.3 + \frac{0.7}{n}}$$

$$0.3 + \frac{0.7}{n} > 2 \times \left(\frac{0.3}{K} + \frac{1 - \frac{0.3}{K}}{n} \right)$$

$$0.3 + \frac{0.7}{n} > \frac{0.6}{K} + \frac{2}{n} - \frac{0.6}{Kn}$$

$$0.3 - \frac{1.3}{n} > \frac{1}{K} \left(0.6 - \frac{0.6}{n} \right)$$

$$K > \frac{0.6 - \frac{0.6}{n}}{0.3 - \frac{1.3}{n}} = \frac{6n - 6}{3n - 13}$$

4.3)

$$\frac{1}{\frac{1-p}{3}+\frac{1-\frac{1-p}{3}}{n}} = 2 \times \frac{1}{1-p+\frac{p}{n}}$$

$$\frac{1}{2\left(\frac{1-p}{3}+\frac{1-\frac{1-p}{3}}{n}\right)} = \frac{1}{1-p+\frac{p}{n}}$$

$$\frac{2-2p}{3} + \frac{2-\frac{2-2p}{3}}{n} = 1-p+\frac{p}{n}$$

$$\frac{2-2p}{3} + \frac{2}{n} - \frac{2-2p}{3n} = 1-p+\frac{p}{n}$$

$$2-2p+\frac{6}{n}-\frac{2-2p}{n} = 3-3p+\frac{3p}{n}$$

$$\frac{6}{n}-\frac{2-2p}{n} = 1-p+\frac{3p}{n}$$

$$\frac{4}{n}+\frac{2p}{n} = 1-p+\frac{3p}{n}$$

$$\frac{2p}{n}+p-\frac{3p}{n} = 1-\frac{4}{n}$$

$$p\left(1-\frac{1}{n}\right) = 1-4n$$

$$p = \frac{n-4}{n-1} \; ; \; n \geq 4$$

k as $1-p$

$$k = 1-p = 1-\frac{n-4}{n-1} = \frac{3}{n-1} \; ; \; n \geq 4$$

Appendix A: Matrix Multiplication Source Code

A.1.0 Matrix Multiplication Class Declaration

```java
package ca.mcgill.ecse420.a1;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.io.FileWriter;
import java.io.IOException;

public class MatrixMultiplication {

   private static final int NUMBER_THREADS = 13;
   private static final int MATRIX_SIZE = 2000;

       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);
       System.out.println("-------------------------Parallel-----------------------------");
       printMatrix(parallelMultiplyMatrix(a, b, MATRIX_SIZE, NUMBER_THREADS));
       System.out.println("-------------------------Sequential---------------------------");
       printMatrix(sequentialMultiplyMatrix(a, b, MATRIX_SIZE));

       System.out.println("Sequential time: " + measurePerformance(MATRIX_SIZE)[0]);
       System.out.println("Parallel time: " + measurePerformance(MATRIX_SIZE)[1]);

       createMatrixMultiplicationCSV(new int[] {100, 200, 500, 1000, 2000});
       createThreadPerformanceCSV(Runtime.getRuntime().availableProcessors());
   }
```

A.1.1 Sequential Matrix Multiplication Function

```java
   /**
    * 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, int size) {
       double c[][] = new double[size][size];
       for (int i = 0; i < size; i++) {
           for (int j = 0; j < size; j++) {
               for (int k = 0; k < size; k++)
                   c[i][j] += a[i][k] * b[k][j];
           }
       }
       return c;
   }
```

A.1.2 Parallel Matrix Multiplication Function

```
   /**
    * 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, int size,
int threads) {
           double[][] c = new double[size][size];
           ExecutorService executor = Executors.newFixedThreadPool(threads);
           for (int i = 0; i < size; i++) {
           for (int j = 0; j < size; j++) {
               // Each element calculation is a task
               DotProduct task = new DotProduct(a, b, c, i, j, size);
               executor.execute(task);
           }
       }

       executor.shutdown();
           try {
               // Wait for all tasks to finish (timeout after 1 minute)
               executor.awaitTermination(1, java.util.concurrent.TimeUnit.MINUTES);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
       return c;
   }
```

A.1.3 Random Matrix Generator Function

```
       /**
        * 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;
   }
```

A.1.4 Dot Product Helper Class

```
   public static class DotProduct implements Runnable {
       private double[][] a;
       private double[][] b;
       private double[][] c;
       private int row;
       private int col;
       private int size;

       public DotProduct(double[][] a, double[][] b, double[][] c, int row, int col, int size)
{
           this.a = a;
           this.b = b;
           this.c = c;
           this.row = row;
           this.col = col;
           this.size = size;
       }

       @Override
       public void run() {
           // Calculate the dot product for one element
           double sum = 0;
           for (int k = 0; k < size; k++) {
               sum += a[row][k] * b[k][col];
           }
           c[row][col] = sum;
       }

   }

   static void printMatrix(double M[][])
   {
       for (int i = 0; i < M[0].length; i++) {
           for (int j = 0; j < M[0].length; j++)
               System.out.print(M[i][j] + " ");

           System.out.println();
       }
   }
```

A.1.5 Performance Measuring Helper Functions

```
static String[] measurePerformance(int size) {
    double[][] a = generateRandomMatrix(size, size);
    double[][] b = generateRandomMatrix(size, size);

    long startSeq = System.currentTimeMillis();
    sequentialMultiplyMatrix(a, b, size);
    long endSeq = System.currentTimeMillis();

    long startPar = System.currentTimeMillis();
    parallelMultiplyMatrix(a, b, size, NUMBER_THREADS);
    long endPar = System.currentTimeMillis();

    String seqTime = String.valueOf(endSeq - startSeq);
    String parTime = String.valueOf(endPar - startPar);

    return new String[] {seqTime, parTime};
}

static void createMatrixMultiplicationCSV(int[] sizes) {
    String csvFile = "matrix_multiplication_performance.csv";
    try (FileWriter writer = new FileWriter(csvFile, false)) {
        writer.append("Matrix Size,Sequential Time(ms),Parallel Time(ms)\n");
        writer.flush();
        for (int size : sizes) {
            String[] times = measurePerformance(size);
            writer.append(size + "," + times[0] + "," + times[1] + "\n");
            writer.flush();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

static void createThreadPerformanceCSV(int threads) {
    String csvFile = "thread_performance.csv";
    try (FileWriter writer = new FileWriter(csvFile)) {
        writer.append("Threads,Time(ms)\n");
        writer.flush();
        for (int i = 1; i < (threads+1); i++) {
            double[][] a = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
            double[][] b = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
            long startPar = System.currentTimeMillis();
            parallelMultiplyMatrix(a, b, MATRIX_SIZE, i);
            long endPar = System.currentTimeMillis();
            writer.append(i + "," + (endPar - startPar) + "\n");
            writer.flush();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
    }
}
```

Appendix B: Deadlock Example Source Code

B.1.0 Deadlock Example Source Code

```
package ca.mcgill.ecse420.a1;
public class DeadlockExample {
   private static final Object resourceA = new Object();
   private static final Object resourceB = new Object();

   public static void main(String[] args) {
       // Thread 1: Locks resourceA then tries to lock resourceB
       Thread thread1 = new Thread(() -> {
           synchronized (resourceA) {
               System.out.println("Thread 1: Locked Resource A");
               try {
                   // Introduce a small delay to allow Thread 2 to acquire its lock
                   Thread.sleep(50);
               } catch (InterruptedException e) {
                   e.printStackTrace();
               }
               System.out.println("Thread 1: Waiting for Resource B");
               synchronized (resourceB) {
                   System.out.println("Thread 1: Locked Resource B");
               }
           }
       });

       // Thread 2: Locks resourceB then tries to lock resourceA
       Thread thread2 = new Thread(() -> {
           synchronized (resourceB) {
               System.out.println("Thread 2: Locked Resource B");
               try {
                   // Introduce a small delay to ensure Thread 1 has acquired its lock
                   Thread.sleep(50);
               } catch (InterruptedException e) {
                   e.printStackTrace();
               }
               System.out.println("Thread 2: Waiting for Resource A");
               synchronized (resourceA) {
                   System.out.println("Thread 2: Locked Resource A");
               }
           }
       });

       thread1.start();
       thread2.start();
   }
}
```

Appendix C: Dining Philosophers

C.1.0 Dining Philosopher Deadlock and Starvation Free Code

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

import java.util.concurrent.Semaphore;

public class DiningPhilosophers {
   private static final int N = 5;  // Number of philosophers
   private static final Semaphore[] chopsticks = new Semaphore[N];
   private static final Semaphore queue = new Semaphore(N - 1, true);

   public static void main(String[] args) {
       for (int i = 0; i < N; i++) {
           chopsticks[i] = new Semaphore(1);
       }
       for (int i = 0; i < N; i++) {
           new Thread(new Philosopher(i)).start();
       }
   }

   public static class Philosopher implements Runnable {
       private final int id;

       Philosopher(int id) {
           this.id = id;
       }

       @Override
       public void run() {
           while (true) {
               System.out.println("Philosopher " + id + " is thinking...");

               try {
                   // Request to enter the queue (FIFO)
                   queue.acquire();

                   // Pick up left chopstick
                   chopsticks[id].acquire();

                   // Pick up right chopstick
                   chopsticks[(id + 1) % N].acquire();

                   System.out.println("Philosopher " + id + " is eating...");

                   // Simulate eating
                   Thread.sleep(100);

                   // Put down left chopstick
                   chopsticks[id].release();
```

```
                    // Put down right chopstick
                    chopsticks[(id + 1) % N].release();

                    // Leave the queue
                    queue.release();

                    System.out.println("Philosopher " + id + " finished eating and put down
chopsticks.");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```