Parsa Yadollahi - 260869949
Nicholas Nikas - 260870980

ECSE 420 - Assignment 1 Report

Question 1:

1.1)

In this question we implemented a method that multiplies two *NxN* matrices sequentially and stores it in a new matrix. This algorithm does this in three nested loops meaning it runs quite slow with a runtime of $O(n^3)$. We also verify in this method if both matrices have the same dimensions and if not, throw an exception.

1.2)

Our parallel algorithm applies matrix multiplication the same way as sequentially. But, in this case, each innermost iteration creates a new thread where they are given a set of rows to apply the dot product on. This allows for the matrix to be divided into the number of available threads without any overlap. We are using a fixed thread pool and the maximum number of threads is dictated by the number of processor threads on the host machine.

1.3)

In java, there exists a built-in timing function that calculates the time in milliseconds. We simply nest both of the matrix multiplication methods between two of these timing functions and return the difference between the two to calculate the time the method takes.

1.4)

Number of threads	Time (millisecond)
1	50069
2	36382
3	29979
4	28292
5	28465
6	26645
7	24653
8	22774
9	53974

Table 1: Table representing execution time vs. Number of threads

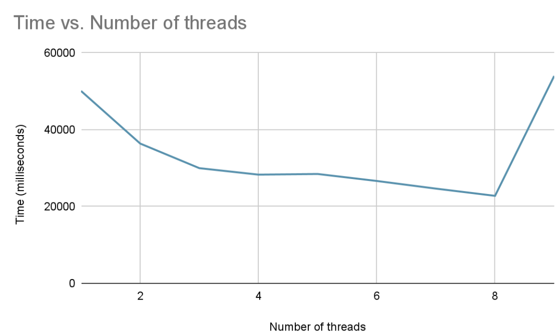

Figure 1: Graph representing execution time vs. Number of threads

From table 1 and figure 1 above we can determine that there is a clear advantage in processing time when increasing the number of threads we use in the thread pool. But we also notice that the more threads there are than available, or the host machine can handle diminishes the results. We reason that this is caused by the extra overhead when needing to create the threads, the extra strain on the thread scheduler, and the potential deadlocks.

1.5)

Matrix Size	Sequential multiplication (ms)	Parallel multiplication (ms)
100x100	10	52
200x200	34	83
500x500	464	359
1000x1000	9761	3278
2000x2000	113276	30349
4000x4000	1041900	388098

Table 2: Execution time vs. Matrix size

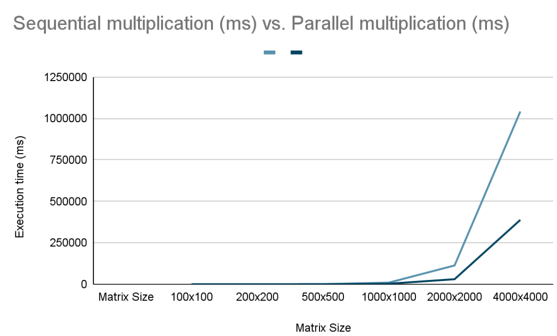

Figure 2: Execution time vs. Matrix size

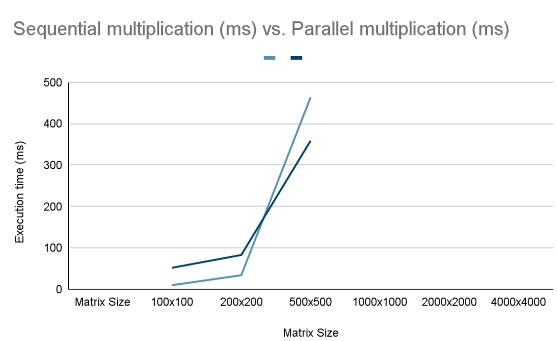

Figure 3: Execution time vs. Matrix size

For this case we decided to use 8 threads which was the best execution time shown in the graph in part 1.4 and figure 1. Figure 2 above demonstrates the difference in execution time between sequential and parallel matrix multiplication. The graph underneath it (figure 3) is simply a zoomed-in version of the graph focusing on matrix dimensions 100x100 to 500x500 to have a more in-depth demonstration of when parallel execution becomes beneficial over sequential. We notice that between 200x200 to 500x500 matrix, the parallel execution has better execution.

1.6)

For the graph in section 1.4 (figure 1), we notice that as we increase the number of threads, the performance also increased until we hit the number of total processors on the host machine (that being 8 in our scenario). Once we start using more threads than this, the performance begins to decrease. As mentioned in part 1.4 under figure 1, this is likely caused by many factors, for example, the overhead of creating more threads, the extra strain on the thread scheduler, and the potential deadlocks.

For the graph in 1.5 (figure 2 and figure 3), we notice that using extra threads really only starts benefitting the execution time once we start computing more complex matrix multiplication. At first, sequential multiplication performs better than parallel but once we start multiplying matrices of dimension 500x500 and higher, increasing the number of threads proves to have better execution time. This is because the time taken to create the threads is neglectable compared to the total computing time.

Question 2:

2.1)

Deadlock occurs when two different threads try to access resources that another thread is holding. The possible consequence is that both threads halt and cannot continue their execution since they don't have the resources to do so. More specifically deadlock can occur only if all the following conditions have been met:

1. Mutual exclusion: When there is at least one resource in the system that can only be used by one process at a time
2. Hold and wait: When there is a process holding onto a resource while waiting for another resource held by another process to become available for use.
3. No preemption: When there is a process holding a resource that can only be released by that process alone.
4. Circular wait: When each process waits for a resource in a cyclic manner where the last process waits for the resource held by the first process.

2.2)

To avoid deadlock we simply must eliminate the possibility of any of the four conditions listed above from occurring. Note that some of the conditions above are inherently impossible to avoid. We could also avoid a deadlock using the Banker's Algorithm, ensuring a safe state to allocate the resources.

There are multiple ways of avoiding a deadlock from occurring:

1. Resource ordering: When each of the objects that need to be locked is assigned an ordering where we ensure that all threads can only acquire the locks in that order.
2. Deadlock detection: When a deadlock occurs simply terminate the process which is holding the resources or to preempt the resource allocated to break the deadlock.
3. We could also ensure that each process requests access to the resources before starting execution. This solution comes with the possibility of causing starvation.

Question 3:

3.1)

The reason there is a deadlock in the DiningPhilosophers program is that there is a point where every Philosopher picks up one chopstick on their left hand while waiting for the right chopstick to become available to pick it up. In other words, every process (philosopher) is waiting on a resource (chopstick) to become available for use causing the deadlock.

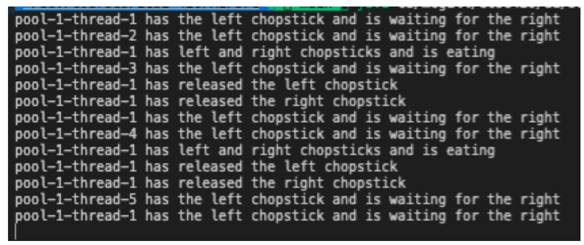

Figure 4: Deadlock

As we can see in figure 4 in the last line of execution, the program halts its execution, where all philosophers are holding a chopstick in their left hand and not releasing it and waiting for a chopstick from another philosopher to add to their right hand to eat.

3.2)

To avoid deadlock we simply ensured that for the last philosopher to eat, they can only pick up their chopsticks if they pick up the right chopstick first instead of the left. Doing so removes condition 4 in question 2.1 which removes any circular wait and hence removes any chance of having a deadlock since not all four conditions are met.

3.3)

To prevent starvation, we used ReentrantLock with a fair trade policy set to true instead of using the synchronized blocks like previously done. This is because the OS schedules thread execution in priority of whichever thread was waiting for their resources the longest. We also ensure that if a philosopher picks up their left chopstick and the right one is unavailable, then that philosopher drops their left chopstick to allow another philosopher to pick it up and eat which will later allow both chopsticks to be available.

Question 4:

Please have a look at the next page

Question 4:

4.1)

$$S = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{1-0.6+\frac{0.6}{n}} = \frac{1}{0.4+\frac{0.6}{n}} \text{ where n is the number of cores}$$

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

4.2)

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

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

$$\Rightarrow \frac{1}{0.3k+\frac{1-0.3k}{n}} > 2 * \frac{1}{0.3+\frac{0.7}{n}} = 0.3 + \frac{0.7}{n} > 0.6k + \frac{2-0.6k}{n}$$

$$\Rightarrow 0.3n + 0.7 > 0.6kn + 2 - 0.6k \Rightarrow$$

$$0.3n - 1.3 > 0.6k(n-1)$$

$$\Rightarrow \frac{0.3n - 1.3}{0.6(n-1)} > k$$

4.3)

$$S_n = \frac{1}{1-p+\frac{1-s}{n}} \Rightarrow S'_n = 0.5 * S_n = \frac{1}{\frac{s}{3}+\frac{1-s/3}{n}} = 2 * \frac{1}{s+\frac{1-s}{n}}$$

$$\Rightarrow (s + \frac{1-s}{n}) = 2(\frac{s}{3} + \frac{1-s/3}{n}) \Rightarrow \frac{sn+1-s}{n} = 2 * \frac{sn-3-s}{3n}$$

$$\Rightarrow 3sn + 3 - 3s = 2sn - 6 - 2s$$

$$\Rightarrow sn - s = 3$$

$$\Rightarrow s = \frac{3}{n-1}$$

Appendix:

For the appendix, please have a look at the following page.

Appendix:

Matrix multiplication:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a1/MatrixMultiplication.java
- java ca/mcgill/ecse420/a1/MatrixMultiplication

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

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

You, a minute ago | 1 author (You)
public class MatrixMultiplication {

  private static final int NUMBER_THREADS = 4;
  private static final int MATRIX_SIZE = 500;

  Run | Debug
  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("Starting sequentialMultiplyMatrix:");
    long begin = System.currentTimeMillis();
    sequentialMultiplyMatrix(a, b);
    long end = System.currentTimeMillis();
    System.out.println("Done sequentialMultiplyMatrix - time = " + (end - begin));

    System.out.println();
    System.out.println("Starting parallelMultiplyMatrix:");
    begin = System.currentTimeMillis();
    parallelMultiplyMatrix(a, b);
    end = System.currentTimeMillis();
    System.out.println("Done parallelMultiplyMatrix - time = " + (end - begin));
  }

  /**
   * 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 aRows = a.length;
    int bRows = b.length;
    int bColumns = b[0].length;
    int aColumns = a[0].length;
    double[][] c = new double[aRows][bColumns];

    // Throw exception if matrix dimensions are invalid
    if (aColumns != bRows) {
      throw new ArithmeticException("Invalid matrix dimensions");
    }

    for (int i = 0; i < aRows; i++) {
      for (int j = 0; j < bColumns; j++) {
        for (int k = 0; k < aColumns; k++){
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }
    return c;
  }

```

```java
/**
 * 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 aRows = a.length;
  int bRows = b.length;
  int bColumns = b[0].length;
  int aColumns = a[0].length;
  double[][] c = new double[aRows][bColumns];

  // Throw exception if matrix dimensions are invalid
  if (aColumns != bRows) {
    throw new ArithmeticException("Invalid matrix dimensions");      You, 2 weeks ago • Cleaned q1
  }

  try {
    // Create a thread pool
    ExecutorService executorService = Executors.newFixedThreadPool(NUMBER_THREADS);

    for (int i = 0; i < aRows; i++) {
      for (int j = 0; j < bColumns; j++) {
        executorService.execute(new ParallelMultiply(i, j, a, b, c));
      }
    }

    // No other threads accept tasks
    executorService.shutdown();

    // Wait for threads to finish
    executorService.awaitTermination(MATRIX_SIZE, TimeUnit.SECONDS);
    System.out.println("Parallel multiplication successfully terminated: " + executorService.isTerminated());
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  return c;
}

You, 2 weeks ago | 1 author (You)
static class ParallelMultiply implements Runnable {
  private int row;
  private  int col;
  private double[][] a;
  private double[][] b;
  private double[][] c;

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

  public void run() {
    for (int k = 0; k < MATRIX_SIZE; k++) {
      c[row][col] += a[row][k] * b[k][col];
    }
  }
}
```

```java
    /**
     * 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:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a1/Deadlock.java
- java ca/mcgill/ecse420/a1/Deadlock

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

public class Deadlock {
  public static String Lock1 = "lock 1";
  public static String Lock2 = "lock 2";
  public static String Thread1 = "Thread 1";
  public static String Thread2 = "Thread 2";

  public static void main(String[] args) {
    // Initialize threads
    DeadlockThread thread1 = new DeadlockThread(Lock1, Lock2, Thread1);
    DeadlockThread thread2 = new DeadlockThread(Lock2, Lock1, Thread2);
    // Start executing both threads
    thread1.start();
    thread2.start();
  }

  public static class DeadlockThread extends Thread {
    private String lock1;
    private String lock2;
    private String threadNumber;

    public DeadlockThread(String lock1, String lock2, String threadNumber) {
      this.lock1 = lock1;
      this.lock2 = lock2;
      this.threadNumber = threadNumber;
    }

    public void run(){
      // Lock the first lock
      synchronized(lock1) {
        System.out.println(threadNumber + ": Holding " + lock1);

        try {
          Thread.sleep(100);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        System.out.println(threadNumber + ": waiting for " + lock2);
        // Lock the second lock
        synchronized (lock2) {
          System.out.println(threadNumber + ": Holding lock 1 & 2");
        }
        // Release the second lock
      }
      // Release the first lock
    }
  }
}
```

DiningPhilosopher with deadlock:

To run this file, simply run the following commands from the src folder

- javac ca/mcgill/ecse420/a1/DiningPhilosophers.java
- java ca/mcgill/ecse420/a1/DiningPhilosophers

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

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

You, 4 minutes ago | 1 author (You)
public class DiningPhilosophers {

    Run | Debug
    public static void main(String[] args) {

        int numberOfPhilosophers = 5;
        Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
        Object[] chopsticks = new Object[numberOfPhilosophers];
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfPhilosophers);
        int leftIndex;
        int rightIndex;


        // Initialize shared objects (chopsticks)
        for (int i = 0; i < numberOfPhilosophers; i++) {
            chopsticks[i] = new Object();
        }

        // Initialize the threads (Philosopher) and execute the threads
        for (int i = 0; i < numberOfPhilosophers; i++) {
            leftIndex = i;
            rightIndex = (i == numberOfPhilosophers - 1) ? 0 : i+1;
            philosophers[i] = new Philosopher(chopsticks[leftIndex], chopsticks[rightIndex]);

            executorService.execute(philosophers[i]);
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        executorService.shutdown();
    }
```

```java
You, 4 minutes ago | 1 author (You)
public static class Philosopher implements Runnable {
    private final Object rightChopstick;
    private final Object leftChopstick;       You, a week ago • A1 – Q3.1

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

    private static void think_wait_or_eat() throws InterruptedException{
        Thread.sleep(10);
    }

    @Override
    public void run() {
        // Keep iterating until we hit a deadlock
        while (true) {
            String name = Thread.currentThread().getName();
            try {
                // Lock the Philosopher's left chopstick
                // If the chopstick is already locked they must wait for it to be available
                synchronized(leftChopstick) {
                    System.out.println(name + " has the left chopstick and is waiting for the right");
                    think_wait_or_eat();

                    // Lock the Philosopher's right chopstick
                    // If the chopstick is already locked they must wait for it to be available
                    synchronized(rightChopstick) {
                        System.out.println(name + " has left and right chopsticks and is eating");
                        think_wait_or_eat();
                    }
                    // Release the right chopstick
                    System.out.println(name + " has released the left chopstick");
                }
                // Release the left chopstick
                System.out.println(name + " has released the right chopstick");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
}
```

DiningPhilosopher without deadlock:

To run this file, simply run the following commands from the src folder

- javacca/mcgill/ecse420/a1/DiningPhilosophersNoDeadlock.java
- java ca/mcgill/ecse420/a1/DiningPhilosophersNoDeadlock

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

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

You, 4 minutes ago | 1 author (You)
public class DiningPhilosophersNoDeadlock {

  Run | Debug
  public static void main(String[] args) {

    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Object[] chopsticks = new Object[numberOfPhilosophers];
    ExecutorService executorService = Executors.newFixedThreadPool(numberOfPhilosophers);
    int leftIndex;
    int rightIndex;


    // Initialize shared objects (chopsticks)
    for (int i = 0; i < numberOfPhilosophers; i++) {
      chopsticks[i] = new Object();
    }

    // Initialize the threads (Philosopher) and execute the threads
    for (int i = 0; i < numberOfPhilosophers; i++) {
      // Force the last philosopher to pick up his fist chopstick
      //  with the right hand.
      // This removes the circular wait which prevents
      // the deadlock from occuring
      if (i == numberOfPhilosophers - 1) {
        leftIndex = 0;
        rightIndex = i;
      } else {
        leftIndex = i;
        rightIndex = (i + 1) % numberOfPhilosophers;
      }
      philosophers[i] = new Philosopher(chopsticks[leftIndex], chopsticks[rightIndex]);

      executorService.execute(philosophers[i]);

      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    executorService.shutdown();
  }

```

```java
You, 4 minutes ago | 1 author (You)
public static class Philosopher implements Runnable {
  private final Object rightChopstick;
  private final Object leftChopstick;
  private int numberEaten = 0;

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

  private static void think_wait_or_eat() throws InterruptedException{
    Thread.sleep(10);
  }

  @Override
  public void run() {
    // Keep iterating until we hit a deadlock
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 100; i++) {
      try {
        // Lock the Philosopher's left chopstick
        // If the chopstick is already locked they must wait for it to be available
        synchronized(leftChopstick) {
          System.out.println(name + " has the left left chopstick and is waiting for the right");
          think_wait_or_eat();

          // Lock the Philosopher's right chopstick
          // If the chopstick is already locked they must wait for it to be available
          synchronized(rightChopstick) {
            System.out.println(name + " has left and right chopsticks and is eating");
            numberEaten ++;
            think_wait_or_eat();
          }
          // Release the right chopstick
          System.out.println(name + " has released the left chopstick");
        }
        // Release the left chopstick
        System.out.println(name + " has released the right chopstick");
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    // Display number of times each philospher has eaten      You, 4 minutes ago • Done project 1
    System.out.println("\n\nPhilosopher " + name.substring(name.length() - 1) + " has eaten " + numberEaten + " times.\n\n");
  }
}
}
```

DiningPhilosopher without starvation:

To run this file, simply run the following commands from the src folder

- javacca/mcgill/ecse420/a1/DiningPhilosophersNoStarvation.java
- java ca/mcgill/ecse420/a1/DiningPhilosophersNoStarvation

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

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

You, 5 minutes ago | 1 author (You)
public class DiningPhilosophersNoStarvation {

  Run | Debug
  public static void main(String[] args) {

    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Chopstick[] chopsticks = new Chopstick[numberOfPhilosophers];
    ExecutorService executorService = Executors.newFixedThreadPool(numberOfPhilosophers);
    int leftIndex;
    int rightIndex;


    // Initialize shared objects (chopsticks)
    for (int i = 0; i < numberOfPhilosophers; i++) {
      chopsticks[i] = new Chopstick();
    }

    // Initialize the threads (Philosopher) and execute the threads      You, a week ago • A
    for (int i = 0; i < numberOfPhilosophers; i++) {
      leftIndex = i;
      rightIndex = (i == numberOfPhilosophers - 1) ? 0 : i+1;
      philosophers[i] = new Philosopher(chopsticks[leftIndex], chopsticks[rightIndex]);

      executorService.execute(philosophers[i]);
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    executorService.shutdown();
  }

  You, 5 minutes ago | 1 author (You)
  public static class Chopstick {
    private ReentrantLock lock = new ReentrantLock(true);

    public Chopstick() {}

    // Function attempting to grab the chopstick
    // In proper terms to lock the ressource
    public boolean grabChopstick(){
      return lock.tryLock();
    }

    // Function attempting to drop the chopstick
    // In proper terms to unlock the ressource
    public void dropChopstick(){
      lock.unlock();
    }
  }

```

You, 5 minutes ago | 1 author (You)

```java
public static class Philosopher implements Runnable {
  private final Chopstick rightChopstick;
  private final Chopstick leftChopstick;
  private int numberEaten = 0;

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

  private static void think_wait_or_eat() throws InterruptedException{
    Thread.sleep(10);
  }

  @Override
  public void run() {
    // Keep iterating until we hit a deadlock
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 100; i++) {
      try {
        // Lock the Philosopher's left chopstick
        // If the chopstick is already locked they must wait for it to be available
        if (leftChopstick.grabChopstick()) {
          System.out.println(name + " has the left chopstick and is waiting for the right");
          think_wait_or_eat();

          // Lock the Philosopher's right chopstick
          // If the chopstick is already locked they must wait for it to be available
          if (rightChopstick.grabChopstick()) {
            System.out.println(name + " has left and right chopsticks and is eating");
            think_wait_or_eat();
            numberEaten++;
            rightChopstick.dropChopstick();
          }
          // Release the right chopstick
          System.out.println(name + " has released the left chopstick");
          leftChopstick.dropChopstick();
          think_wait_or_eat();
        }
        think_wait_or_eat();
        // Release the left chopstick
        System.out.println(name + " has released the right chopstick");
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    // Display number of times each philospher has eaten
    System.out.println("\n\nPhilosopher " + name.substring(name.length() - 1) + " has eaten " + numberEaten + " times.\n\n");
  }
}
}
```