Alexander Harris - 260688155
Abbas Yadollahi - 260680343

ECSE 420
October 10, 2018

ECSE 420 - Assignment 1 Report

Q1:

1.1

Implemented a method for calculating the product of two matrices sequentially using 3 nested for loops. This algorithm is quite slow at $O(n^3)$. This method also checks if both matrices have valid dimensions for multiplication, and throws an exception if not.

1.2

Our parallel method employs the same algorithm as the sequential method, however for the innermost for loop we create a new thread for each iteration. This means that each dot product operation takes place in its own thread. We are using a fixed thread pool for this case and the maximum number of threads is by default set to the number of available processor threads on the host machine.

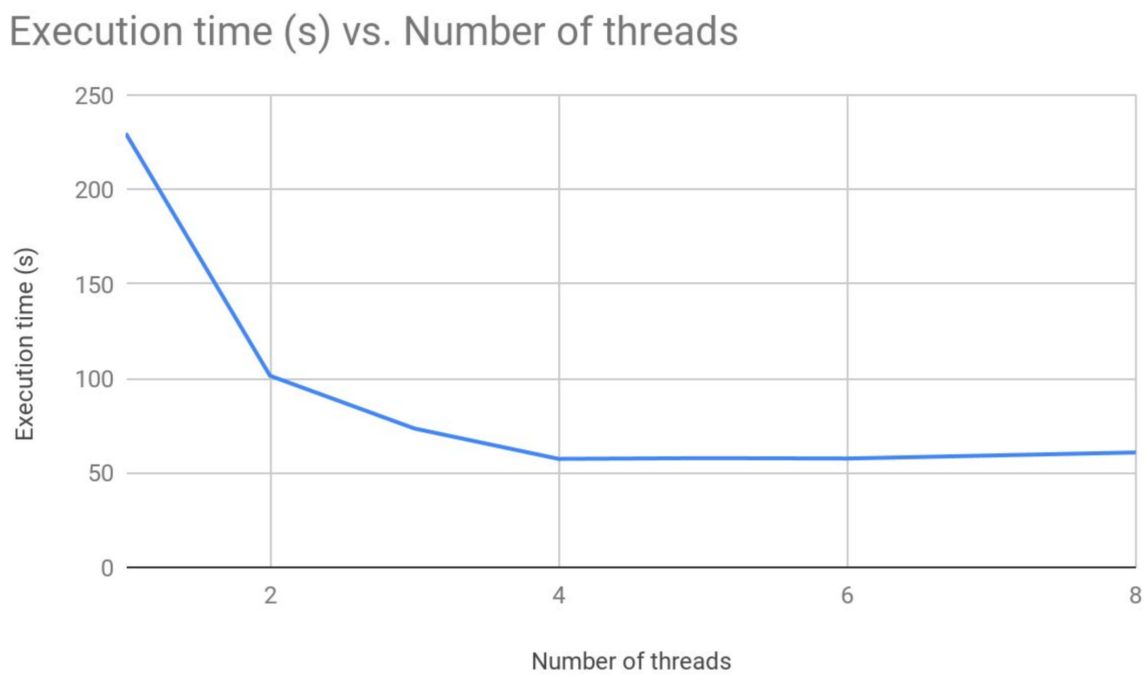
1.3

Here we get the current system time in milliseconds before and after executing the method. We then take the difference of the two to get the method's execution time.

1.4

For a matrix size of 2000 x 2000 and a host machine with 4 processor threads:

Number of threads	**Execution time (s)**
1	229.7
2	101.4
3	73.5
4	57.4
5	57.8
6	57.6
8	60.8

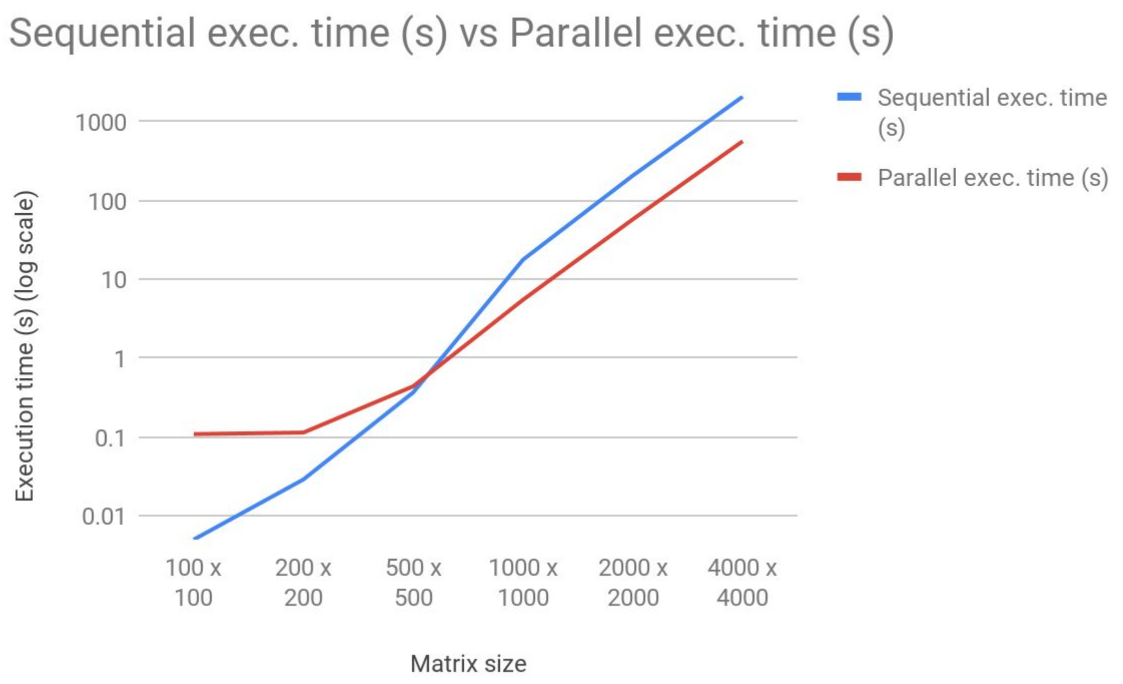


As we can see, there is a clear advantage to increasing the number of threads the pool can use. However, creating more threads than the host machine's processor can handle (exceeding the numbers of processor cores/threads) can lead to diminishing returns or even performance degradation.

1.5

Using 4 threads:

Matrix size	**Sequential exec. time (s)**	**Parallel exec. time (s)**
100 x 100	0.005	0.109
200 x 200	0.029	0.114
500 x 500	0.369	0.441
1000 x 1000	17.8	5.52
2000 x 2000	207.7	57.9
4000 x 4000	2081.7	566.5



1.6

For 1.4, we can see that improvement in execution begins to plateau as we approach the number of physical processor threads on the host machine, for our case 4. We can also see that as we continue to increase the number of threads beyond this the execution actually begins to increase. We believe this is because it gets more complicated for the host OS scheduler to process the increased number of threads.

As for 1.5, we can see that for small matrix sizes the sequential method is actually much faster than the parallel method. This is because of the time needed to create the threads for the parallel method. However, once we get to matrix sizes above 500x500, the parallel method starts to become significantly faster than the sequential method since the overhead caused by creating threads becomes negligible compared to the overall computation time.

Q2:

2.1

Deadlock can occur only if all the following conditions are met:

- Mutual exclusion: prevents simultaneous access to a shared resource.
- Hold and wait: A process is holding one resource while requesting other resources which are held by other processes.
- No preemption: Resources can only be released by the process holding it.
- Circular wait: Each process is waiting for a resource held by another process in the list, with the last process waiting for resources from the first.

2.2

There are multiple different strategies to avoiding deadlock. For example:

- Resource ordering: Each object that needs to be locked is assigned an order and we ensure that all threads can only acquire the locks in that order. This helps to avoid the circular wait condition for deadlock.
- Deadlock detection: Detect deadlocks when they happen and either terminate the offending process or preempt the resource allocated to break the deadlock.
- Require processes to request all resources needed before executing. However, it is difficult to predict what the process will need and this can often lead to starvation.

Q3:

3.1

To cause deadlock in our Dining Philosopher program, every philosopher needs to grab the chopstick to his right (without releasing it until after eating), then grab the chopstick to his left. It will inevitably cause a deadlock since at some point all philosophers will be holding the chopstick to their right, while waiting for the chopstick to their left which will never be available.

3.2

To solve the imminent deadlock from the initial Dining Philosopher program, we forced at least one philosopher (the first in our case) to wait for the left chopstick to be available before grabbing the right chopstick. What this does is eliminate the circular wait condition that is a key element in causing deadlock.

3.3

We used ReentrantLock with the fair policy set to True instead of the synchronized() block in order to prevent starvation. This method always gives priority to the thread that has been waiting the longest so no thread ever starves. At the same time, when a philosopher picks up the right chopstick but can't pick up the left chopstick (due to it already being in use), we force him to drop his right chopstick to allow other philosophers to dine.

```
Connected to the target VM, address: '127.0.0.1:41897', transport: 'socket'
Philosopher 4 ate 438 times and waited for 1.102393728 seconds
Philosopher 8 ate 438 times and waited for 0.956193565 seconds
Philosopher 21 ate 416 times and waited for 0.949622594 seconds
Philosopher 2 ate 464 times and waited for 1.014146057 seconds
Philosopher 11 ate 442 times and waited for 0.967855131 seconds
Philosopher 3 ate 441 times and waited for 0.970082827 seconds
Philosopher 28 ate 424 times and waited for 0.972241621 seconds
Philosopher 13 ate 465 times and waited for 1.003296633 seconds
Philosopher 12 ate 432 times and waited for 1.018043156 seconds
Philosopher 1 ate 474 times and waited for 1.034164701 seconds
Philosopher 7 ate 456 times and waited for 1.053293582 seconds
Philosopher 31 ate 454 times and waited for 1.034014694 seconds
Philosopher 33 ate 441 times and waited for 1.005967611 seconds
Philosopher 16 ate 478 times and waited for 1.114366652 seconds
Philosopher 44 ate 455 times and waited for 1.044088844 seconds
Philosopher 20 ate 475 times and waited for 1.070861408 seconds
Philosopher 10 ate 433 times and waited for 1.021526106 seconds
Philosopher 42 ate 413 times and waited for 0.961041856 seconds
Philosopher 46 ate 425 times and waited for 0.926664192 seconds
Philosopher 5 ate 470 times and waited for 1.144000738 seconds
Philosopher 41 ate 448 times and waited for 1.014932565 seconds
Philosopher 35 ate 454 times and waited for 1.011922048 seconds
Philosopher 6 ate 468 times and waited for 1.057921819 seconds
Philosopher 15 ate 446 times and waited for 1.080866654 seconds
Philosopher 19 ate 457 times and waited for 1.057144552 seconds
Philosopher 24 ate 460 times and waited for 1.048198898 seconds
Philosopher 25 ate 458 times and waited for 1.073394802 seconds
Philosopher 18 ate 456 times and waited for 1.06706051 seconds
Philosopher 36 ate 447 times and waited for 0.999623797 seconds
Philosopher 22 ate 470 times and waited for 1.069131804 seconds
Philosopher 29 ate 442 times and waited for 1.084731084 seconds
Philosopher 37 ate 438 times and waited for 0.951853966 seconds
Philosopher 23 ate 451 times and waited for 0.99577943 seconds
Philosopher 14 ate 446 times and waited for 1.067163046 seconds
Philosopher 49 ate 418 times and waited for 0.940420896 seconds
Philosopher 48 ate 457 times and waited for 1.01744061 seconds
Philosopher 26 ate 451 times and waited for 1.079880601 seconds
Philosopher 43 ate 461 times and waited for 1.084158762 seconds
Philosopher 17 ate 469 times and waited for 1.066266842 seconds
Philosopher 45 ate 477 times and waited for 1.066670421 seconds
Philosopher 47 ate 457 times and waited for 0.998829588 seconds
Philosopher 30 ate 471 times and waited for 1.140055683 seconds
Philosopher 34 ate 456 times and waited for 1.078883875 seconds
Philosopher 39 ate 471 times and waited for 1.120646313 seconds
Philosopher 50 ate 454 times and waited for 1.101858918 seconds
Philosopher 38 ate 459 times and waited for 1.039274262 seconds
Philosopher 9 ate 497 times and waited for 1.179101449 seconds
Philosopher 40 ate 450 times and waited for 1.05958375 seconds
Philosopher 27 ate 494 times and waited for 1.176325962 seconds
Disconnected from the target VM, address: '127.0.0.1:41897', transport: 'socket'
Philosopher 32 ate 510 times and waited for 1.168613117 seconds

Process finished with exit code 0
```

```
Connected to the target VM, address: '127.0.0.1:42683', transport: 'socket'
Philosopher 50 ate 1000 times and waited for 2.936154593 seconds
Philosopher 49 ate 1000 times and waited for 4.232693206 seconds
Philosopher 48 ate 1000 times and waited for 5.166449549 seconds
Philosopher 47 ate 1000 times and waited for 5.884278745 seconds
Philosopher 46 ate 1000 times and waited for 6.636597814 seconds
Philosopher 45 ate 1000 times and waited for 7.11264779 seconds
Philosopher 44 ate 1000 times and waited for 7.69285394 seconds
Philosopher 43 ate 1000 times and waited for 7.735811561 seconds
Philosopher 42 ate 1000 times and waited for 8.134206404 seconds
Philosopher 41 ate 1000 times and waited for 8.566088324 seconds
Philosopher 40 ate 1000 times and waited for 8.883048842 seconds
Philosopher 38 ate 1000 times and waited for 9.3429495 seconds
Philosopher 39 ate 1000 times and waited for 9.218039274 seconds
Philosopher 37 ate 1000 times and waited for 9.68769104 seconds
Philosopher 36 ate 1000 times and waited for 9.828706847 seconds
Philosopher 35 ate 1000 times and waited for 10.032723296 seconds
Philosopher 34 ate 1000 times and waited for 10.319163922 seconds
Philosopher 33 ate 1000 times and waited for 10.580865951 seconds
Philosopher 32 ate 1000 times and waited for 10.832351929 seconds
Philosopher 31 ate 1000 times and waited for 11.160958025 seconds
Philosopher 30 ate 1000 times and waited for 11.474042842 seconds
Philosopher 29 ate 1000 times and waited for 11.815972303 seconds
Philosopher 28 ate 1000 times and waited for 11.928248321 seconds
Philosopher 27 ate 1000 times and waited for 12.104392127 seconds
Philosopher 26 ate 1000 times and waited for 12.361550142 seconds
Philosopher 25 ate 1000 times and waited for 12.736897299 seconds
Philosopher 24 ate 1000 times and waited for 12.716953485 seconds
Philosopher 23 ate 1000 times and waited for 12.831713292 seconds
Philosopher 22 ate 1000 times and waited for 13.037909587 seconds
Philosopher 21 ate 1000 times and waited for 13.231296464 seconds
Philosopher 20 ate 1000 times and waited for 13.568926357 seconds
Philosopher 19 ate 1000 times and waited for 13.665305361 seconds
Philosopher 18 ate 1000 times and waited for 13.881167538 seconds
Philosopher 16 ate 1000 times and waited for 13.960874889 seconds
Philosopher 17 ate 1000 times and waited for 14.026335089 seconds
Philosopher 15 ate 1000 times and waited for 14.282090815 seconds
Philosopher 14 ate 1000 times and waited for 14.351347619 seconds
Philosopher 13 ate 1000 times and waited for 14.645081748 seconds
Philosopher 12 ate 1000 times and waited for 14.918615861 seconds
Philosopher 11 ate 1000 times and waited for 15.193292407 seconds
Philosopher 10 ate 1000 times and waited for 15.496815679 seconds
Philosopher 9 ate 1000 times and waited for 15.696839283 seconds
Philosopher 8 ate 1000 times and waited for 15.964747567 seconds
Philosopher 7 ate 1000 times and waited for 16.228303843 seconds
Philosopher 6 ate 1000 times and waited for 16.260371561 seconds
Philosopher 3 ate 1000 times and waited for 16.577915953 seconds
Philosopher 1 ate 1000 times and waited for 16.490269841 seconds
Philosopher 4 ate 1000 times and waited for 16.60856369 seconds
Philosopher 5 ate 1000 times and waited for 16.614252267 seconds
Disconnected from the target VM, address: '127.0.0.1:42683', transport: 'socket'
Philosopher 2 ate 1000 times and waited for 17.280595567 seconds

Process finished with exit code 0
```

We can see from the above images that the program using ReentrantLocks (on the left) has much lower and more consistent total wait times for each philosopher than the synchronized block method on the right.

Q4:

4.1

$$S = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{1-0.6+\frac{0.6}{n}} = \frac{1}{0.4+\frac{0.6}{n}},\ where\ n = \#\ of\ cores$$

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

4.2

$$S_n = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{1-0.8+\frac{0.8}{n}} = \frac{1}{0.2+\frac{0.8}{n}}$$

$$S_n{}' > 2 * S_n \Rightarrow \frac{1}{0.2k+\frac{1-0.2k}{n}} > 2 * \frac{1}{0.2+\frac{0.8}{n}}$$

$$\frac{1}{0.2kn+1-0.2k} > 2 * \frac{1}{0.2n+0.8}$$

$$0.2n + 0.8 > 0.4kn + 2 - 0.4k$$

$$k < \frac{0.5n-3}{n-1}$$

4.3

$$S_n = \frac{1}{s+\frac{1-s}{n}}$$

$$S_n{}' = 0.5 * S_n = \frac{1}{\frac{s}{3}+\frac{1-\frac{s}{3}}{n}} = 0.5 * \frac{1}{s+\frac{1-s}{n}}$$

$$\frac{sn}{3} + 1 - \frac{s}{3} = \frac{sn+1-s}{2}$$

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

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

Appendix:

Matrix multiplication:

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

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

public class MatrixMultiplication {

  //  private static final int NUMBER_THREADS =
Runtime.getRuntime().availableProcessors();
  private static final int NUMBER_THREADS = 4;
  private static final int MATRIX_SIZE = 2000;
  private static final int NUM_ITERATIONS = 1;

  public static void main(String[] args) {

    boolean success = true;
    for (int i = 0; i < NUM_ITERATIONS; i++) {

      System.out.println("\n==== ITERATION " + (i + 1) + "/" + NUM_ITERATIONS +
" ====\n");

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

      System.out.println("###### Sequential Multiplication ######");
      start = System.currentTimeMillis();
      double[][] seqResult = sequentialMultiplyMatrix(a, b);
      stop = System.currentTimeMillis();
      System.out.println("Sequential exec time(ms): " + (stop - start));

      System.out.println("###### Parallel Multiplication ######");
      start = System.currentTimeMillis();
      double[][] paraResult = parallelMultiplyMatrix(a, b);
      stop = System.currentTimeMillis();
      System.out.println("Parallel exec time(ms): " + (stop - start));

      // Check if results from parallel and sequential are the same
      if (!Arrays.deepEquals(seqResult, paraResult)) {
        success = false;
        break;
      }
```

```java
    }
    if (!success) {
      System.out.println("Sequential and parallel results did not match");
    }

  }

  /**
   * 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");
    }

    // Naive O(n^3) method for matrix multiplication
    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;
  }

  /**
   * 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) {
```

```java
    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");
    }

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

      for (int i = 0; i < aRows; i++) {
        for (int j = 0; j < bColumns; j++) {
          // Spawn each row operation in its own thread
          executor.execute(new ParallelMultiply(i, j, a, b, c));
        }
      }

      executor.shutdown();

      // Wait for all threads to finish before continuing
      executor.awaitTermination(MATRIX_SIZE, TimeUnit.SECONDS);
      System.out.println("Terminated successfully: " + executor.isTerminated());

    } catch (Exception e) {
      e.printStackTrace();
    }

    return c;
  }

  static class ParallelMultiply implements Runnable {

    private int row;
    private int column;
    private double[][] a;
    private double[][] b;
    private double[][] c;

    ParallelMultiply(final int row, final int column, final double[][] a, final
double[][] b, final double[][] c) {
      this.row = row;
      this.column = column;
      this.a = a;
      this.b = b;
      this.c = c;
```

```
    }

    public void run() {
      for (int k = 0; k < a.length; k++) {
        c[row][column] += a[row][k] * b[k][column];
      }
    }
  }

  /**
   * 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:

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

public class Deadlock {

  public static Object lock1 = new Object();
  public static Object lock2 = new Object();

  public static void main(String[] args) {
    DeadlockThread thread1 = new DeadlockThread(lock1, lock2);
    DeadlockThread thread2 = new DeadlockThread(lock2, lock1);
    thread1.start();
    thread2.start();
  }

  public static class DeadlockThread extends Thread {
    final Object lock1;
    final Object lock2;
```

```
    public DeadlockThread(Object lock1, Object lock2) {
      this.lock1 = lock1;
      this.lock2 = lock2;
    }

    public void run() {
      synchronized (lock1) {
        System.out.println("Lock 1");
        try {
          // Gives time for other thread to acquire lock on its first resource
          Thread.sleep(100);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
        synchronized (lock2) {
          System.out.println("Lock 2");
        }
      }
    }
  }
}
```

Dining philosophers (with deadlock):

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

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

public class DiningPhilosophersDeadlock {

  public static void main(String[] args) {

    int numberOfPhilosophers = 50;
    Object[] chopsticks = new Object[numberOfPhilosophers];
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    ExecutorService executor =
Executors.newFixedThreadPool(numberOfPhilosophers);

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

    // Initialize Philosophers and execute the Thread
    for (int i = 0; i < numberOfPhilosophers; i++) {
      philosophers[i] = new Philosopher(chopsticks, i, numberOfPhilosophers);
      executor.execute((philosophers[i]));
      try {
```

```java
        Thread.sleep((long) (Math.random() * 10));
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    executor.shutdown();
  }

  public static class Philosopher implements Runnable {

    final Object rightChopstick;
    final Object leftChopstick;

    public Philosopher(Object[] chopsticks, int position, int
numberOfPhilosophers) {
      this.rightChopstick = chopsticks[(position + 1) % numberOfPhilosophers];
      this.leftChopstick = chopsticks[position];
    }

    @Override
    public void run() {
      while (true) {
        String name = Thread.currentThread().getName();

        try {
          // Lock the philosopher's right chopstick
          // If chopstick is already locked, wait until available
          synchronized (rightChopstick) {
            System.out.println(name + " - Holding Right Chopstick");
            Thread.sleep((long) (Math.random() * 5));

            // Lock the philosopher's left chopstick and eat
            // If chopstick is already locked, wait until available
            synchronized (leftChopstick) {
              System.out.println(name + " - Holding Left Chopstick");
              System.out.println(name + " - Eating");
              Thread.sleep((long) (Math.random() * 5));
            }
            System.out.println(name + " - Released Left Chopstick");
          }
          System.out.println(name + " - Released Right Chopstick");
          Thread.sleep((long) (Math.random() * 10));
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
}
```

Dining philosophers (no deadlock):

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

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

public class DiningPhilosophersNoDeadlock {

  public static void main(String[] args) {

    int numberOfPhilosophers = 50;
    Object[] chopsticks = new Object[numberOfPhilosophers];
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    ExecutorService executor =
Executors.newFixedThreadPool(numberOfPhilosophers);

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

    // Initialize Philosophers and execute the Thread
    for (int i = 0; i < numberOfPhilosophers; i++) {
      philosophers[i] = new Philosopher(chopsticks, i, numberOfPhilosophers);
      executor.execute((philosophers[i]));
      try {
        Thread.sleep((long) (Math.random() * 10));
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    executor.shutdown();
  }

  public static class Philosopher implements Runnable {

    int ate = 0;
    final int id;
    final Object rightChopstick;
    final Object leftChopstick;

    Philosopher(Object[] chopsticks, int position, int numberOfPhilosophers) {
      this.id = position + 1;

      // Force the first philosopher to grab the chopstick to his left first
      // As a result, we end up in a circular wait situation where each
philosopher
      // is holding on to one chopstick
      if (position != 0) {
        this.rightChopstick = chopsticks[position];
```

```
        this.leftChopstick = chopsticks[(position + 1) % numberOfPhilosophers];
      } else {
        this.rightChopstick = chopsticks[(position + 1) % numberOfPhilosophers];
        this.leftChopstick = chopsticks[position];
      }
    }

    @Override
    public void run() {
      long start;
      long totalWait = 0;

      for (int x = 0; x < 1000; x++) {
        start = System.nanoTime();

        try {
          // Lock the philosopher's right chopstick
          // If chopstick is already locked, wait until available
          synchronized (rightChopstick) {
//            System.out.println(id + " - Holding Right Chopstick");
            Thread.sleep((long) (Math.random() * 5));

            // Lock the philosopher's left chopstick and eat
            // If chopstick is already locked, wait until available
            synchronized (leftChopstick) {
//              System.out.println(id + " - Holding Left Chopstick");
//              System.out.println(id + " - Eating");
              totalWait += System.nanoTime() - start;
              ate++;
              Thread.sleep((long) (Math.random() * 5));
            }
//            System.out.println(id + " - Released Left Chopstick");
          }
//          System.out.println(id + " - Released Right Chopstick");
          Thread.sleep((long) (Math.random() * 10));
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
      // To see # of times each philosopher ate, comment out the other
System.out.println() lines
      System.out.println("Philosopher " + id + " ate " + ate + " times and
waited for " + totalWait/1000000000.0 + " seconds");
    }
  }
}
```

Dining philosophers (no starvation):

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

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

public class DiningPhilosophersNoStarvation {

  public static void main(String[] args) {

    int numberOfPhilosophers = 50;
    Chopstick[] chopsticks = new Chopstick[numberOfPhilosophers];
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    ExecutorService executor =
Executors.newFixedThreadPool(numberOfPhilosophers);

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

    // Initialize Philosophers and execute the Thread
    for (int i = 0; i < numberOfPhilosophers; i++) {
      philosophers[i] = new Philosopher(i, i > 0 ? chopsticks[i - 1] :
chopsticks[chopsticks.length - 1], chopsticks[i]);
      executor.execute((philosophers[i]));
      try {
        Thread.sleep((long) (Math.random() * 10));
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    executor.shutdown();
  }

  public static class Chopstick {

    private ReentrantLock reLock = new ReentrantLock(true);

    public Chopstick() {
    }

    // Attempt to pick up chopstick, if successful lock resource
    public boolean grabStick() {
      return reLock.tryLock();
    }

    // Release the lock on the chopstick
```

```java
    public void dropStick() {
      reLock.unlock();
    }
  }

  public static class Philosopher implements Runnable {

    int ate = 0;
    final int id;
    final Chopstick rightChopstick;
    final Chopstick leftChopstick;

    public Philosopher(int position, Chopstick rightChopstick, Chopstick
leftChopstick) {
      this.id = position + 1;
      this.rightChopstick = rightChopstick;
      this.leftChopstick = leftChopstick;
    }

    @Override
    public void run() {
      long start;
      long totalWait = 0;

      for (int x = 0; x < 1000; x++) {
        start = System.nanoTime();

        try {
          if (rightChopstick.grabStick()) {
//            System.out.println(id + " - Holding Right Chopstick");
            Thread.sleep((long) (Math.random() * 5));

            if (leftChopstick.grabStick()) {
//              System.out.println(id + " - Holding Left Chopstick");
//              System.out.println(id + " - Eating");
              totalWait += System.nanoTime() - start;
              ate++;
              Thread.sleep((long) (Math.random() * 5));
              leftChopstick.dropStick();
//              System.out.println(id + " - Released Left Chopstick");
            }

            rightChopstick.dropStick();
//            System.out.println(id + " - Released Right Chopstick");
          }
          Thread.sleep((long) (Math.random() * 10));
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
```

```
      // To see # of times each philosopher ate, comment out the other System.out.println() lines
      System.out.println("Philosopher " + id + " ate " + ate + " times and waited for " + totalWait/1000000000.0 + " seconds");
    }
  }
}
```