Spiro Mavroidakos - 260689391
Jastaj Virdee - 260689027

ECSE 420- Assignment 1

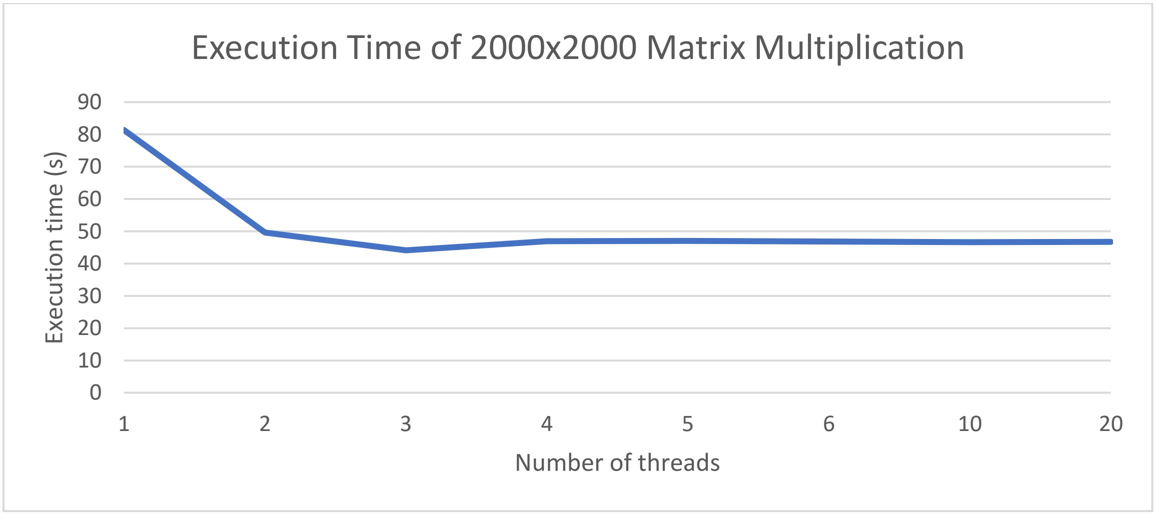
Q1

1. Question 1.1 was implemented using a basic matrix multiplication algorithm. The method starts at line 32 in the MatrixMultiplication.java file, see Appendix, Question 1.1 for details.

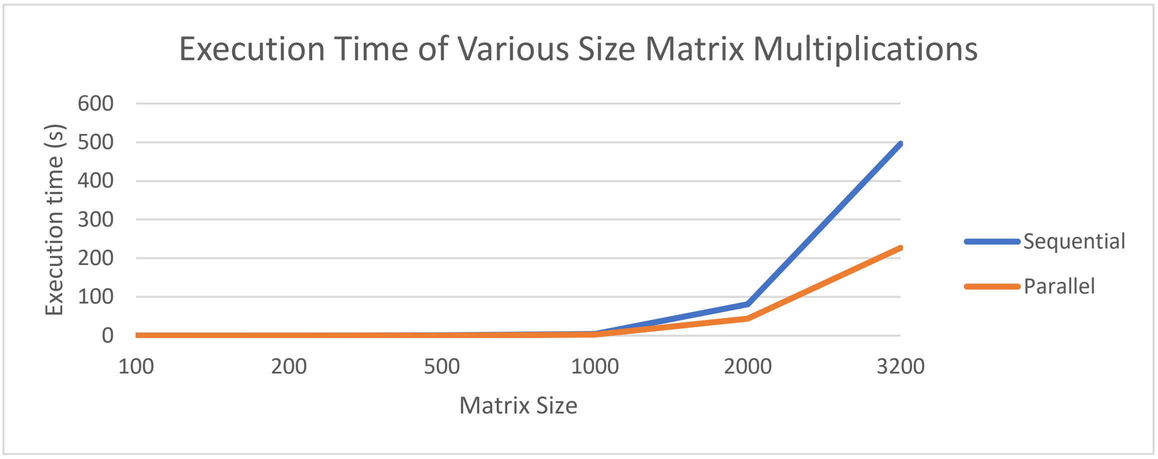
2. The tasks that were defined in the code that work in parallel are matrix multiplication algorithms that only produce one row of the product matrix. In other words if matrix A is multiplied with matrix B to produce matrix C, each of the tasks will produce one row of C. Hence after all the tasks are completed, the entire product matrix will be produced. The method and task are at lines 63 and 119 respectively in the MatrixMultiplication.java file. See Appendix, Question 1.2 for details.

3. To measure the execution time of the methods, the time was recorded at the beginning of the methods and saved in a variable. This was done again at the end of the methods. These two variables were then passed to a method that took the difference between the times and divided by 1000 (to get the elapsed time in seconds). The method starts at line 113 in the MatrixMultiplication.java file, see Appendix, Question 1.3 for details.

4.



5.



6. As seen in the graph in 1.4, the execution time decreases after the first two increments of the number of threads, however it then levels off. The reason for this is that the CPU being used only has two cores, so adding more than two threads will not aid in decreasing the execution time, which is why the time levels out.

Q2

The code for this question can be found in the Appendix under Q2.

DeadLock in the execution of the program is shown by the following output:

```
Thread with id = 12 acquiring Lock 1
Thread with id = 13 Acquiring Lock 2
Thread with id = 13 Holding Lock 2
Thread with id = 12 Holding Lock 1
Thread with id = 13 Acquiring Lock 1
Thread with id = 12 Acquiring Lock 2
```

Here it is clear that deadlock has occurred as thread with id 12 has lock 1 but wants to acquire lock 2 while thread with id 13 is holding lock 2 and wants to acquire lock 1 but lock one is held already as stated prior.

1. Deadlock can occur when 2 or more threads need to acquire the locks on several shared objects. Deadlock happens when each thread holds a lock that the other needs and therefore leads to all the threads waiting for a lock that that they cannot get.

   There are 4 conditions for deadlock to occur:

   1. Mutual exclusion
   2. Hold and wait: A thread is holding a resource while waiting to get another.
   3. No pre-emption: A thread will not give up a resource until it finishes using it.
   4. Circular wait: Each thread holds a resource that another needs.

Conditions 1,2 and 3 are necessary but not enough for deadlock to occur. All 4 conditions are needed.

2. One solution would be resource ordering. In this solution, an order is assigned to each object whose lock needs to be acquired and ensure that each thread acquires the locks in that order.

   Another solution would be to evaluate if granting the lock will cause the system to deadlock. For this to happen, every thread would need to send a list of all the shared resources that it needs. The system will also keep track of how much of a specific resource is available. The system will then receive requests for resources and deem if granting this request will lead to deadlock. If found that the request will make it so that no other thread can terminate due to being inability of getting a resource, then the request gets denied and received more requests, granting only those that are safe. This is known as Banker's algorithm.

Q3

The code for this question can be found in the Appendix under Q3.1, Q3.2-3.3.

1. In the source code for the DiningPhilosophersDeadlocked.java, deadlocked is displayed in the following output for n = 5;

```
Philosopher 14 is hungry. Attempting to acquire chopsticks.
Philosopher 12 is hungry. Attempting to acquire chopsticks.
Philosopher 15 is hungry. Attempting to acquire chopsticks.
Philosopher 16 is hungry. Attempting to acquire chopsticks.
Philosopher 13 is hungry. Attempting to acquire chopsticks.
Philosopher 12 picked up left chopstick.
Philosopher 14 picked up left chopstick.
Philosopher 16 picked up left chopstick.
Philosopher 15 picked up left chopstick.
Philosopher 13 picked up left chopstick.
```

   Each philosopher is holding the chopstick to their left and therefore all the chopsticks are being held by another philosopher. The philosophers are therefore deadlocked as no one can pick up the chopstick to their right which is why "Philosopher # picked up right chopstick" is not printed to the output.

2. To avoid deadlock, the solution is simple. Create one lock (re-entrant lock) to act as a moderator. When a philosopher gets hungry, they will ask the moderator if they can acquire the chopsticks (acquire the lock before the locks on the chopsticks). If the philosopher can acquire the lock then they have access to the chopsticks. The only issue with this approach is that only one philosopher will be able to eat at a time but that also means that deadlock will not occur.

3. Using the solution for 3.2, we can simply set the reenetrant lock to implement fairness by setting it (i.e. lock = new ReentranLock(true)). As seen in class, fair locks prevent starvations as this ensures that every thread has fair access to the shared resources. Fairness implies that every thread will be able to access the resources they need.

Q4

1. $s = 0.4, p = 1 - s = 1 - 0.4 = 0.6$

$$Speedup = \frac{1}{s + p/n} = \frac{1}{0.4 + 0.6/n}$$
$$\lim_{n\to\infty} \frac{1}{0.4 + 0.6/n} = \frac{1}{0.4} = 2.5$$

2. $sn' = 2sn$

$$\frac{1}{s' + \frac{1 - s'}{n}} = \frac{2}{s + \frac{1 - s}{n}} = \frac{2}{0.2 + \frac{1 - 0.2}{n}} = \frac{2n}{0.2n + 0.8} = \frac{10n}{n + 4}$$
$$\frac{n}{s'n - s' + 1} = \frac{10n}{n + 4}$$
$$n + 4 = 10s'n - 10s' + 10$$
$$n - 6 = 10s'(n - 1)$$
$$s' = \frac{n - 6}{10(n - 1)} = sk = 0.2k \to k = \frac{n - 6}{2(n - 1)}$$

3. $\frac{2}{s+\frac{1-s}{n}} = \frac{1}{\frac{s}{3}+\frac{1-s/3}{n}}$

$$\frac{2n}{sn - s + 1} = \frac{1}{sn + 3 - s/3n} = \frac{3n}{sn - s + 3}$$
$$\frac{2n}{sn - s + 1} = \frac{3n}{sn - s + 3}$$
$$2sn - 2s + 6 = 3sn - 3s + 3$$
$$3 = sn - s = s(n - 1)$$
$$s = \frac{3}{n - 1}$$

Appendix

Question 1.1

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

Question 1.2

```
/**
 * 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) {
	double[][] c = new double[MATRIX_SIZE][MATRIX_SIZE];
	ExecutorService executor =Executors.newFixedThreadPool(NUMBER_THREADS);
	for (int i = 0; i < MATRIX_SIZE; i++) {
		executor.execute(new OneRowMultiplication(a,b,c,i,MATRIX_SIZE));
	}
	executor.shutdown();
	return c;
}

// task class: matrix multiplication on only 1 row
class OneRowMultiplication implements Runnable {
	private double[][] a, b, c;
	private int i, MATRIX_SIZE;
	public OneRowMultiplication(double[][] m1, double[][] m2,
	                              double[][] m3, int index, int m_size) {
		a = m1; b = m2; c = m3;
		i = index; MATRIX_SIZE = m_size;
	}
	@Override
	public void run() {
		for (int j = 0; j < MATRIX_SIZE; j++) {
			for (int k = 0; k < MATRIX_SIZE; k++) {
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

```
                        }
                }
        }
}
```

Question 1.3

```
public static double[][] sequentialMultiplyMatrix(double[][] a, double[][] b)
{
      double start = System.currentTimeMillis();
      // rest of the method, see Appendix A, Question 1.1
      double end = System.currentTimeMillis();
      System.out.println("Sequential: " + measureTime(start,end) + "s");
      }

public static double[][] parallelMultiplyMatrix(double[][] a, double[][] b) {
      double start = System.currentTimeMillis();
      // rest of the method, see Appendix A, Question 1.2
      try {
            executor.awaitTermination(30, TimeUnit.MINUTES);
      } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
      }
      double end = System.currentTimeMillis();
      System.out.println("Parallel: " + measureTime(start,end) + "s");

}

/**
 * Measure the execution time of a method
 * @param start is the start time
 * @param end is the end time
 * @return end - start time of the method
 * */
public static double measureTime(double start, double end) {
      return (end - start)/1000;
}
```

Question 2

```
public class DeadLock {
  public static ReentrantLock lock1 = new ReentrantLock();

  public static ReentrantLock lock2 = new ReentrantLock();

  public static void main(String arg[]) {
    DeadLockThread dead = new DeadLockThread();
    DeadLockThread2 dead2 = new DeadLockThread2();

    // dead will acquire lock1 and then attempt to acquire lock2
    dead.start();
    // dead2 will acquire lock2 and then attempt to acquire lock1
    dead2.start();

    // Due to the acquisition order of the locks, this will always lead
    // to a deadlock situation.
  }
  private static class DeadLockThread extends Thread {
    public DeadLockThread() {

    }

    public void run() {
      System.out.println("Thread with id = " + this.getId() + " acquiring
                  Lock 1");
      lock1.lock();
      System.out.println("Thread with id = " + this.getId() + " Holding Lock
                  1");

      System.out.println("Thread with id = " + this.getId() + " Acquiring
                  Lock 2");
      lock2.lock();
      System.out.println("Thread with id = " + this.getId() + " Holding Lock
                  2");

      System.out.println("Thread with id = " + this.getId() + " Releasing
                  locks");
      lock2.unlock();
      lock1.unlock();

      System.out.println("Thread with id = " + this.getId() + " Done thread
                  execution");
    }
  }

  private static class DeadLockThread2 extends Thread {
    public DeadLockThread2() {

    }

    public void run() {
      System.out.println("Thread with id = " + this.getId() + " Acquiring
                  Lock 2");
      lock2.lock();
      System.out.println("Thread with id = " + this.getId() + " Holding Lock
                  2");

      System.out.println("Thread with id = " + this.getId() + " Acquiring
                  Lock 1");
      lock1.lock();
      System.out.println("Thread with id = " + this.getId() + " Holding Lock
                  1");

      System.out.println("Thread with id = " + this.getId() + " Releasing
                  locks");
      lock2.unlock();
      lock1.unlock();

      System.out.println("Thread with id = " + this.getId() + " Done thread
                  execution");
    }
  }
}
```

Question 3.1

```
public class DiningPhilosophersDeadlocked {
    public static void main(String[] args) {
    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Object[] chopsticks = new Object[numberOfPhilosophers];

    // Initialize the chopsticks
    for (int i = 0; i < chopsticks.length; i++) {
      chopsticks[i] = new Object();
    }

    // Initialize each philosopher by setting their chopsticks
    for (int i = 0; i < philosophers.length; i++) {
      philosophers[i] = new Philosopher(chopsticks[i], chopsticks[(i + 1) %
            chopsticks.length]);

      Thread philosophThread = new Thread(philosophers[i]);
      philosophThread.start();
    }
  }

  public static class Philosopher implements Runnable {
    private Object LeftChopstick;

    private Object RightChopstick;

    public int eatingAmount = 0;

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

    @Override
    public void run() {
      while (true) {
        try {
          System.out.println(
              "Philosopher " + Thread.currentThread().getId() + " is hungry.
                  Attempting to acquire chopsticks.");

          synchronized (this.LeftChopstick) {
            System.out.println("Philosopher " +
              Thread.currentThread().getId() + " picked up left chopstick.");

            synchronized (this.RightChopstick) {
              System.out
                  .println("Philosopher " + Thread.currentThread().getId() +
                  " picked up Right chopstick. Eating now.");

              // Simulate eating
              Thread.sleep(((int) (Math.random() * 1000)));
            }
          }
          System.out.println("Philosopher " + Thread.currentThread().getId()
              + " finished using the chopsticks. Going back to thinking.");

          // Simulate thinking.
          Thread.sleep(((int) (Math.random() * 1000)));
        } catch (InterruptedException e) {
          System.out.println(e.getMessage());
          return;
        }
      }
    }
  }
}
```

Question 3.2-3.3

```
public class DiningPhilosophers {
  // Lock to avoid deadlock and starvation
  // Avoids starvation as the reetrant lock is fair
  // Solution for 3.2 and 3.3
  private static ReentrantLock lock = new ReentrantLock(true);

  public static void main(String[] args) {
    int numberOfPhilosophers = 5;
    Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];
    Object[] chopsticks = new Object[numberOfPhilosophers];

    // Initialize the chopsticks
    for (int i = 0; i < chopsticks.length; i++) {
      chopsticks[i] = new Object();
    }

    // Initialize each philosopher by setting their chopsticks
    for (int i = 0; i < philosophers.length; i++) {
      philosophers[i] = new Philosopher(chopsticks[i], chopsticks[(i + 1) %
            chopsticks.length]);

      Thread philosophThread = new Thread(philosophers[i]);
      philosophThread.start();
    }
  }

  public static class Philosopher implements Runnable {
    private Object LeftChopstick;

    private Object RightChopstick;

    public int eatingAmount = 0;

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

    @Override
    public void run() {
      while (true) {
        try {
          System.out.println(
              "Philosopher " + Thread.currentThread().getId() + " is hungry.
                        Attempting to acquire chopsticks.");

          // Add the lock to avoid Deadlock and starvation!
          // Solution for 2 and 3
          lock.lock();

          synchronized (this.LeftChopstick) {
            System.out.println("Philosopher " +
            Thread.currentThread().getId() + " picked up left chopstick.");

            synchronized (this.RightChopstick) {
              System.out
                  .println("Philosopher " + Thread.currentThread().getId() +
                  " picked up Right chopstick. Eating now.");

              // Simulate eating
              Thread.sleep(((int) (Math.random() * 1000)));
            }
          }
          System.out.println("Philosopher " + Thread.currentThread().getId()
              + " finished using the chopsticks. Going back to thinking.");

          // Simulate thinking.
          Thread.sleep(((int) (Math.random() * 1000)));
        } catch (InterruptedException e) {
          System.out.println(e.getMessage());
          return;
        } finally {
          // Release the lock so that no one else will be blocked if thread
          //dies
          // Solution for 3.2 and 3.3
          lock.unlock();
        }
      }
    }
  }
}
```