Parsa Yadollahi - 260869949
Nichola Nikas -

ECSE 420 - Assignment 1 Report

Question 1:

1.1)

In this question we implemented a method which 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)

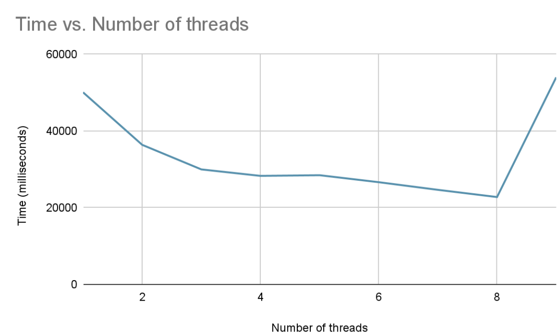
In java, there exists a built-in timing function which calculates time in milliseconds. We simply nest both of the matrix multiplication methods between two of these timing functions and return the difference between the two to calculate the time the method takes.

1.4)

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

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

Figure 2: Graph representing execution time vs. Number of threads



From the graph above we can determine that there is a clear advantage to increasing the number of threads we can 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. 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)

Figure 3: Execution time vs. Matrix size

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

Figure 4: Execution time vs. Matrix size

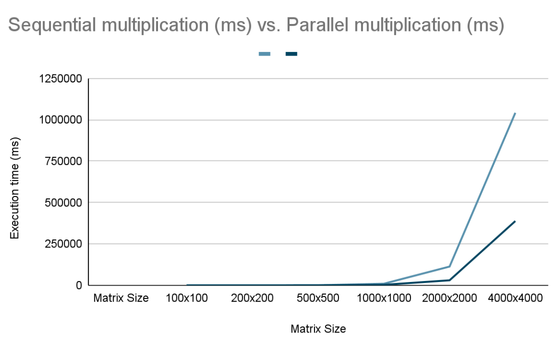
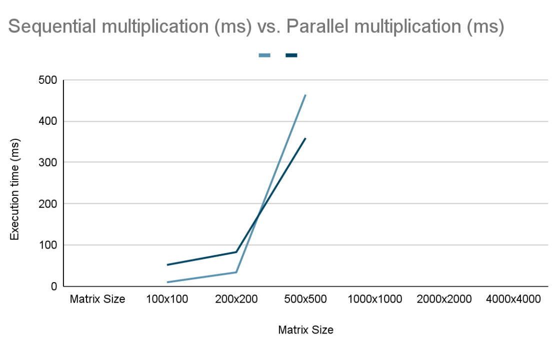


Figure 5: Execution time vs. Matrix size



For this case we decided to use 8 threads which was the best execution time shown in the graph in part 1.4. The graph at the top demonstrates the difference in execution time between sequential and parallel matrix multiplication. The graph underneath it is simply a zoomed in version of the graph focusing on matrix dimensions 100x100 to 500x500 to demonstrate when parallel execution becomes beneficial over sequential.

1.6)

For the graph in 1.4, we notice that the more we increase the number of threads the better the performance until we hit the number of total processors on the host machine (that being 8). Once we start using more threads than this, the performance begins to decrease. Like mentioned in part 1.4 under the graph, 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, 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: Process holding a resource can only be released by that process alone.
4. Circular wait: 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 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:

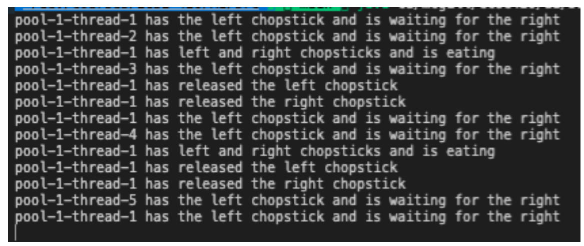
1. Resource ordering: Each of the objects that need to be locked are assigned an ordering where we ensure that all threads can only acquire the locks in that order.
2. Deadlock detection: When a deadlock occurs simply terminate the process which is holding the resources or preempt the resource allocated to break the deadlock.
3. Ensure each process requests access to the resources before starting execution. The issue with this solution is that it may cause starvation

Question 3:

3.1)

The reason there is a deadlock in the DiningPhilosophers program is because 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.

Figure 6: Deadlock



```
pool-1-thread-1 has the left chopstick and is waiting for the right
pool-1-thread-2 has the left chopstick and is waiting for the right
pool-1-thread-1 has left and right chopsticks and is eating
pool-1-thread-3 has the left chopstick and is waiting for the right
pool-1-thread-1 has released the left chopstick
pool-1-thread-1 has released the right chopstick
pool-1-thread-1 has the left chopstick and is waiting for the right
pool-1-thread-4 has the left chopstick and is waiting for the right
pool-1-thread-1 has left and right chopsticks and is eating
pool-1-thread-1 has released the left chopstick
pool-1-thread-1 has released the right chopstick
pool-1-thread-5 has the left chopstick and is waiting for the right
pool-1-thread-1 has the left chopstick and is waiting for the right
```

As we can see the program hits a point where all philosophers are holding a chopstick and not releasing it.

3.2)

To avoid deadlock we simply ensured that the last philosopher to eat can only pick up his chopsticks if he picks up the right chopstick first instead of the left. Doing so removes condition 4 in question 2.1 removing any circular wait.

3.3)

To prevent starvation, we used ReentrantLock with fair trade policy set to true instead of using the synchronized blocks like previously. 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.

Question 4:
4.1)

$$S = \frac{1}{1-p+\frac{p}{n}} = \frac{1}{1 - 0.6 + \frac{0.6}{n}} = \frac{1}{0.4 + \frac{0.6}{n}}$$

$$where\ n\ is\ the\ number\ of\ cores$$

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

4.2)

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

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

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

$$\Rightarrow \frac{0.3 + \frac{0.7}{n}}{0.6+\frac{1.4}{n}} \Rightarrow \frac{0.3n + 0.7}{0.6n+1.4} \Rightarrow \frac{0.6n + 1.4}{0.3n+0.7} > k$$

4.3)

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

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

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

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

$$\Rightarrow sn - s = 3$$

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

Appendix:

Matrix multiplication:

```java
You, 3 days ago | 1 author (You)
package ca.mcgill.ecse420.a1;

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

You, 3 days ago | 1 author (You)
public class MatrixMultiplication {

  private static final int NUMBER_THREADS = 4;
  private static final int MATRIX_SIZE = 2000;

  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);

    long begin = System.currentTimeMillis();
    sequentialMultiplyMatrix(a, b);
    long end = System.currentTimeMillis();
    System.out.println("Time sequentialMultiplyMatrix: " + (end - begin));

    begin = System.currentTimeMillis();
    parallelMultiplyMatrix(a, b);
    end = System.currentTimeMillis();
    System.out.println("Time parallelMultiplyMatrix: " + (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");
    }

    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("Multiplication succeessfully terminated: " + executorService.isTerminated());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return c;
  }

  You, 3 days 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];
      }
    }
  }

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

}
```

Deadlock:

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

You, 3 days ago | 1 author (You)
public class Deadlock {
  public static String Lock1 = "lock 1";      You, 3 days ago • A1 – Q2
  public static String Lock2 = "lock 2";
  public static String Thread1 = "Thread 1";
  public static String Thread2 = "Thread 2";

  Run | Debug
  public static void main(String[] args) {
    DeadlockThread thread1 = new DeadlockThread(Lock1, Lock2, Thread1);
    DeadlockThread thread2 = new DeadlockThread(Lock2, Lock1, Thread2);
    thread1.start();
    thread2.start();
  }

  You, 3 days ago | 1 author (You)
  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(){
      synchronized(lock1) {
        System.out.println(threadNumber + ": Holding " + lock1);

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

    }
  }
}
```

DiningPhilosopher with deadlock:

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

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

You, 19 hours 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();
    }
```

You, 19 hours ago | 1 author (You)

```java
    public static class Philosopher implements Runnable {
        private final Object rightChopstick;
        private final Object leftChopstick;

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

        @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");
                        Thread.sleep(10);

                        // 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");
                            Thread.sleep(10);
                        }
                        // 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:

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

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

You, 19 hours 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++) {
      leftIndex = i;
      if (i == numberOfPhilosophers) {
        rightIndex = leftIndex;
        leftIndex = 0;
      } else {
        rightIndex = 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, 19 hours ago | 1 author (You)

```java
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;
  }

  @Override
  public void run() {
    // Keep iterating until we hit a deadlock
    String name = Thread.currentThread().getName();
    for (int i = 0; i < 100; i++) {      You, 19 hours ago • Uncommitted changes
      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");
          Thread.sleep(10);

          // 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");
            Thread.sleep(10);
          }
          // 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();
      }
    }
    System.out.println("\n\nPhilosopher " + name.substring(name.length() - 1) + " has eaten " + numberEaten + " times.\n\n");
  }
}
}
```

DiningPhilosopher without starvation:

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

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

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
    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();
  }

  public static class Chopstick {
    private ReentrantLock lock = new ReentrantLock(true);

    public Chopstick() {}

    public boolean grabChopstick(){
      return lock.tryLock();

    }

    public void dropChopstick(){
      lock.unlock();
    }
  }

```

```java
    public static class Philosopher implements Runnable {
        private final Chopstick rightChopstick;
        private final Chopstick leftChopstick;
        private int numberEaten = 0;
          You, 16 minutes ago • A1 - Q3
        public Philosopher(Chopstick rightChopstick, Chopstick leftChopstick) {
            this.rightChopstick = rightChopstick;
            this.leftChopstick = leftChopstick;
        }

        @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");
                        Thread.sleep(10);

                        // 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");
                            Thread.sleep(10);
                            numberEaten++;
                            rightChopstick.dropChopstick();
                        }
                        // Release the right chopstick
                        System.out.println(name + " has released the left chopstick");
                        leftChopstick.dropChopstick();
                        Thread.sleep(10);
                    }
                    Thread.sleep(10);
                    // Release the left chopstick
                    System.out.println(name + " has released the right chopstick");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("\n\nPhilosopher " + name.substring(name.length() - 1) + " has eaten " + numberEaten + " times.\n\n");
        }
    }
}
```