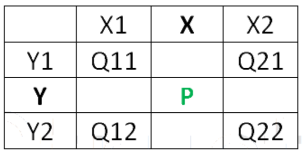
Project 3: Java Parallel Tasks

Tristin Greenstein   Anthony M   Vladimir S   Jefferson Perez Diaz

April 24, 2021

1 Bilinear Interpolation:

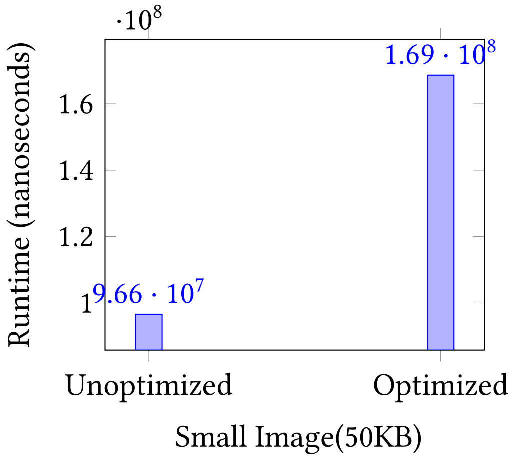
Bilinear Interpolation is an algorithm to determine the value of a function if the four corners of the object is known. It is performed by first doing Linear Interpolation in one direction, then again in the reverse direction. The equation is shown below:

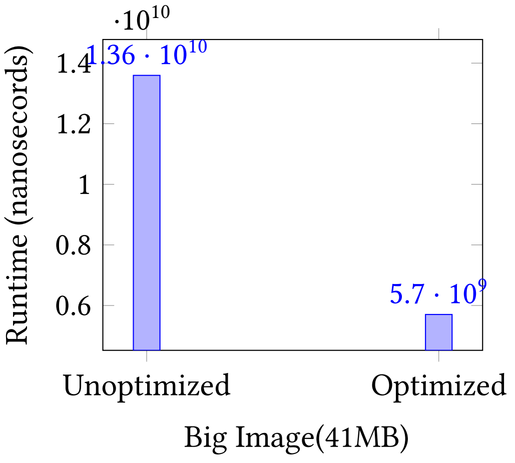


	X1	**X**	X2
Y1	Q11		Q21
Y		P	
Y2	Q12		Q22

$$P \approx \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} Q_{11} + \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} Q_{21} + \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} Q_{12} + \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} Q_{22}$$

We programmed the Bilinear Interpolation in Java. We then created a copy of the code to function in parallel by implementing Java Streams. We used the streams to run the algorithm in parallel to find out whether or not this optimization saved us run time. We used first a 50KB image and then a 41MB image to test our code. The results are seen in the graph below.





Our conclusion was that with smaller images, running the Bilinear Interpolation code in parallel was slower then the base code. However when testing bigger and bigger images, we found out that the parallel approach became faster and soon surpassed the unoptimized version of the code.

Unoptimized:

```
for (int x = 0; x < newWidth; ++x) {
    for (int y = 0; y < newHeight; ++y) {
        float gx = ((float) x) / newWidth * (self.getWidth() - 1);
        float gy = ((float) y) / newHeight * (self.getHeight() - 1);
        int gxi = (int) gx;
        int gyi = (int) gy;
        int rgb = 0;
        int c00 = self.getRGB(gxi, gyi);
        int c10 = self.getRGB(gxi + 1, gyi);
        int c01 = self.getRGB(gxi, gyi + 1);
        int c11 = self.getRGB(gxi + 1, gyi + 1);
        for (int i = 0; i <= 2; ++i) {
            float b00 = get(c00, i);
            float b10 = get(c10, i);
            float b01 = get(c01, i);
            float b11 = get(c11, i);
            int ble = ((int) blerp(b00, b10, b01, b11, gx - gxi,
            gy - gyi)) << (8 * i);
            rgb = rgb | ble;
        }
        newImage.setRGB(x, y, rgb);
    }
}
```

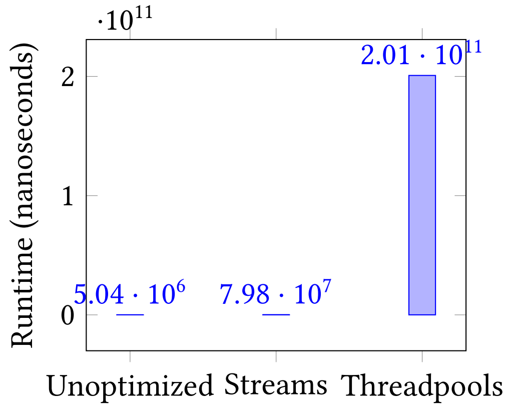
Optimized:

```
IntStream.range(0, newWidth).parallel().forEach(x -> {
    IntStream.range(0, newHeight).parallel().forEach(y -> {
        float gx = ((float) x) / newWidth * (self.getWidth() - 1);
        float gy = ((float) y) / newHeight * (self.getHeight() - 1);
        int gxi = (int) gx;
        int gyi = (int) gy;
        int c00 = self.getRGB(gxi, gyi);
        int c10 = self.getRGB(gxi + 1, gyi);
        int c01 = self.getRGB(gxi, gyi + 1);
        int c11 = self.getRGB(gxi + 1, gyi + 1);
        int rgb = IntStream.rangeClosed(0, 2).map(i -> {
            float b00 = get(c00, i);
            float b10 = get(c10, i);
            float b01 = get(c01, i);
            float b11 = get(c11, i);
            int ble = ((int) blerp(b00, b10, b01, b11, gx - gxi,
            gy - gyi)) << (8 * i);
            return ble;
        }).reduce(0, (a, b) -> a | b);
        newImage.setRGB(x, y, rgb);
    });
});
```

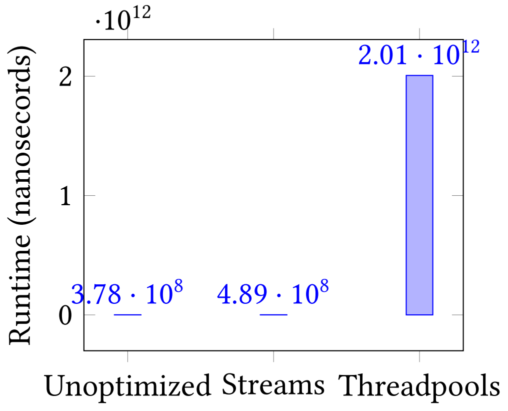
2 Gaussian Elimination:

The Gaussian Elimination algorithm, also known as row reduction algorithm, is used to solve systems of linear equations. It is performed on matrix's to find its rank, determinant, and or inverse. The algorithm is not expressed as a mathematical equation but as a series of operations performed on a matrix of coefficients.

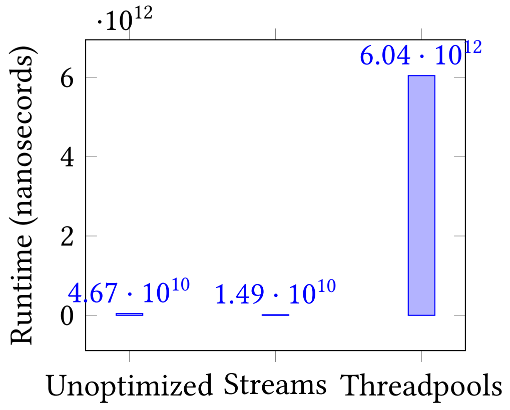
We programmed the row reduction formula in java. We then created two copies of the code and implemented both Java Streams and Java Thread pools to try to optimize the run time for various size matrix's. The results are shown below:



Matrix[100][100]



Matrix[1000][1000]



Matrix[5000][5000]

Our conclusion was that in Threadpools in every case of increasing matrix size the Threadpools performed the worse in runtime. This result may have been caused because we needed to create an object that would store three different variables to allow us to update the matrices in time. Unfortunately we were trading performance by using this methodology as the code would have to call an external class every time it run the object. We used two runnable methods since the code has two outer loops , that would not allow us to run both segments of the code in the same loop, so we implemented a class name "ImplementSecondLoop" to run the second segment of the code. For when we ran Streams we discovered that once you reach a certain size matrix, it begins to run faster than both the unoptimized and thread pools.

Unoptimized:

(NOTE: all code is posted since major changes in all areas)

```java
public class GaussianElimination {
    public static double solve(double[][] a, double[][] b) {
        if (a == null || b == null || a.length == 0 || b.length == 0) {
            throw new IllegalArgumentException("Invalid dimensions");
        }
        int n = b.length, p = b[0].length;
        if (a.length != n || a[0].length != n) {
            throw new IllegalArgumentException("Invalid dimensions");
        }
        double det = 1.0;
        for (int i = 0; i < n - 1; i++) {
            int k = i;
            for (int j = i + 1; j < n; j++) {
                if (Math.abs(a[j][i]) > Math.abs(a[k][i])) {
                    k = j;
                }
            }
            if (k != i) {
                det = -det;
                for (int j = i; j < n; j++) {
                    double s = a[i][j];
                    a[i][j] = a[k][j];
                    a[k][j] = s;
                }
                for (int j = 0; j < p; j++) {
                    double s = b[i][j];
                    b[i][j] = b[k][j];
                    b[k][j] = s;
                }
            }
            for (int j = i + 1; j < n; j++) {
                double s = a[j][i] / a[i][i];
                for (k = i + 1; k < n; k++) {
                    a[j][k] -= s * a[i][k];
                }
                for (k = 0; k < p; k++) {
                    b[j][k] -= s * b[i][k];
                }
            }
        }
        for (int i = n - 1; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                double s = a[i][j];
                for (int k = 0; k < p; k++) {
                    b[i][k] -= s * b[j][k];
                }
            }
            double s = a[i][i];
            det *= s;
            for (int k = 0; k < p; k++) {
                b[i][k] /= s;
            }
        }
        return det;
    }
    public static void main(String[] args) {
        int size = 100;
        double[][] a = new double[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (i < j) {
                    a[i][j] = 0;
                } else {
                    a[i][j] = 45;
                }
            }
        }
        double[][] b = new double[size][1];
        for (int i = 0; i < size; i++) {
            b[i][0] = (double) i / (double) i + 1;
        }
        long startTime = System.nanoTime();
        System.out.println("det: " + solve(a, b));
        long endTime = System.nanoTime();
        long duration = (endTime - startTime);
        System.out.println(duration + " milliseconds (not parallel)");
    }
}
```

Streams

```java
import java.util.Collections;
import java.util.stream.IntStream;
public class GaussianEliminationParallel {

    public static double det = 1.0;
    public static int k;

    public static double solve(double[][] a, double[][] b) {
        if (a == null || b == null || a.length == 0 || b.length == 0) {
            throw new IllegalArgumentException("Invalid dimensions");
        }
        int n = b.length, p = b[0].length;
        if (a.length != n || a[0].length != n) {
            throw new IllegalArgumentException("Invalid dimensions");
        }
        IntStream.range(0, n - 1).forEach(i -> {
            k = i;
            IntStream.range(i + 1, n).forEach(j -> {
                if (Math.abs(a[j][i]) > Math.abs(a[k][i])) {
                    k = j;
                }
            });
            IntStream.range(i + 1, n).forEach(j -> {
                if (Math.abs(a[j][i]) > Math.abs(a[k][i])) {
                    k = j;
                }
            });
            if (k != i) {
                det = -det;

                IntStream.range(i, n).parallel().forEach(j -> {
                        double s = a[i][j];
                        a[i][j] = a[k][j];
                        a[k][j] = s;
                    // });
                });
                IntStream.range(0, p).parallel().forEach(j -> {
                    double s = b[i][j];
                    b[i][j] = b[k][j];
                    b[k][j] = s;
                });
            }
            IntStream.range(i + 1, n).parallel().forEach(j -> {
                double s = a[j][i] / a[i][i];
                IntStream.range(i + 1, n).forEach(k -> {
                    a[j][k] -= s * a[i][k];
                });
                IntStream.range(0, p).forEach(k -> {
                    b[j][k] -= s * b[i][k];
                });
            });
        });
        IntStream.rangeClosed(0, n -
            1).boxed().sorted(Collections.reverseOrder()).forEach(i -> {
            IntStream.range(i + 1, n).parallel().forEach(j -> {
                double s = a[i][j];
                IntStream.range(0, p).forEach(k -> {
                    b[i][k] -= s * b[j][k];
                });
            });
            double s = a[i][i];
            det *= s;
            IntStream.range(0, p).forEach(k -> {
                b[i][k] /= s;
            });
        });
        return det;
    }
    public static void main(String[] args) {
        int size = 2000;
        double[][] a = new double[size][size];
```

```
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (i < j) {
                    a[i][j] = 0;
                } else {
                    a[i][j] = 1.1;
                }
            }
        }
        double[][] b = new double[size][1];
        for (int i = 0; i < size; i++) {
            b[i][0] = (double) i / (double) i + 1;
        }
        long startTime = System.nanoTime();
        System.out.println("det: " + solve(a, b));
        long endTime = System.nanoTime();
        long duration = (endTime - startTime);
        System.out.println(duration + " nanoseconds (parallel)");
    }
```

Threadpools:

```
import java.util.ArrayList;
import java.util.Locale;
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
public class GaussElimThreadPools implements Runnable {
   private double[][] a;
   private double[][] b;
   private double det;
   private int p;
   private int n;
   private int i;
   public DoubleHolder temp;
    public GaussElimThreadPools( DoubleHolder temp,int n, int p,int i) {
      this.a = temp.getArray1();
      this.b =temp.getArray2();
      this.n = n;
      this.p =p;
      this.i = i;
      this.det= temp.getDet();
      this.temp = temp;
   }
   public static double solve(double[][] a, double[][] b) throws InterruptedException,
       ExecutionException {
      if (a == null || b == null || a.length == 0 || b.length == 0) {
```

```java
            throw new IllegalArgumentException("Invalid dimensions");
        }
        int n = b.length, p = b[0].length;
        if (a.length != n || a[0].length != n) {
            throw new IllegalArgumentException("Invalid dimensions");
        }
        double det = 1.0;
        DoubleHolder temp = new DoubleHolder(a,b ,det);
        ExecutorService thread =
            Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        for (int i = 0; i < n - 1; i++) {
         thread.execute(new GaussElimThreadPools(temp, n, p, i));
         Thread.sleep(1000);
        }
       det =  temp.getDet();
       for (int i = n - 1; i >= 0; i--) {
         thread.execute(new ImplementSeconLoopPart(temp, n, p, i));
         Thread.sleep(1000);
       }
       thread.shutdown();
       thread.awaitTermination(1, TimeUnit.SECONDS);
      return temp.getDet();
   }
    public static void main(String[] args) throws InterruptedException,
        ExecutionException {
        double[][] a = new double[][] { { 4.0, 1.0, 0.0, 0.0, 0.0 }, { 1.0, 4.0, 1.0,
            0.0, 0.0 },
           { 0.0, 1.0, 4.0, 1.0, 0.0 }, { 0.0, 0.0, 1.0, 4.0, 1.0 }, { 0.0, 0.0, 0.0,
               1.0, 4.0 } };
           double[][] b = new double[][] { { 1.0 / 2.0 }, { 2.0 / 3.0 }, { 3.0 / 4.0 },
               { 4.0 / 5.0 }, { 5.0 / 6.0 } };
           int size =5000;
          double [][]matrix = new double[size][size];
          for(int i=0;i<size;i++){
           for(int j =0; j < size; j++) {
              matrix[i][j] = (double)new Random().nextInt(5);
           }
             }
          double [][]whatever = new double[size][1];
          for(int i=0;i<size;i++){
              whatever[i][0] = (double) i / (double) i+1;
          }
        double[] x = { 39.0 / 400.0, 11.0 / 100.0, 31.0 / 240.0, 37.0 / 300.0, 71.0 /
            400.0 };
        long startTime = System.nanoTime();
        System.out.println("det:  " + solve(a,b));
        long endTime = System.nanoTime();
        for (int i = 0; i < 5; i++) {
            System.out.printf(Locale.US, "%12.8f %12.4e\n", b[i][0], b[i][0] - x[i]);
        }
        long duration = (endTime - startTime);
      System.out.println(duration + " nanoseconds ( parallel)");
  }

 private static class ImplementSeconLoopPart implements Runnable {
   private double[][] a2;
   private double[][] b2;
   private double det;
   private int p;
   private int n;
   private int i;
   private DoubleHolder temp;
   ImplementSeconLoopPart( DoubleHolder temp ,int n, int p, int i) {
      this.a2 = temp.getArray1();
      this.b2 =temp.getArray2();
      this.det = temp.getDet();
      this.p =p;
      this.i =i;
      this.n =n;
      this.temp = temp;
   }
   @Override
   public void run() {
        for (int j = i + 1; j < n; j++) {
            double s = a2[i][j];
            for (int k = 0; k < p; k++) {
                b2[i][k] -= s * b2[j][k];
            }
         }
         double s = a2[i][i];
         det *= s;
         for (int k = 0; k < p; k++) {
            b2[i][k] /= s;
         }
        temp.setArrays(new DoubleHolder(a2,b2,det));
   }
 }
 @Override
 public void run() {
   int k = i;
     for (int j = i + 1; j < n; j++) {
         if (Math.abs(a[j][i]) > Math.abs(a[k][i])) {
             k = j;
         }
     }
     if (k != i) {
         det = -det;
         for (int j = i; j < n; j++) {
             double s = a[i][j];
             a[i][j] = a[k][j];
             a[k][j] = s;
         }
```

```
            for (int j = 0; j < p; j++) {
                double s = b[i][j];
                b[i][j] = b[k][j];
                b[k][j] = s;
            }
        }
        for (int j = i + 1; j < n; j++) {
            double s = a[j][i] / a[i][i];
            for (k = i + 1; k < n; k++) {
                a[j][k] -= s * a[i][k];
            }
            for (k = 0; k < p; k++) {
                b[j][k] -= s * b[i][k];
            }
        }

        temp.setArrays(new DoubleHolder(a,b,det));
    }
}
```