Project 3

Kevin Siraki

April 2021

1 Introduction

In this project, we essentially started out by running the original code for Bilinear Interpolation and Gaussian Elimination that was provided to us on Rosetta Code and adding a bit of code using System.NanoTime() in Java to record the total runtime of each program. Next, we proceeded to convert every for loop in each program to parallelized for-each loops by utilizing IntStreams. Overall, our results were quite easy to extrapolate prior to even running the code, but the in-depth details and code analysis will provide more reasoning as to why the outcomes were not entirely shocking to us.

2 Bilinear Interpolation Serial:

We first do linear interpolation in the x-direction. This yields

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

We proceed by interpolating in the y-direction to obtain the desired estimate:

$$\begin{aligned} f(x, y) &\approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \\ &= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \left(f(Q_{11})(x_2 - x)(y_2 - y) + f(Q_{21})(x - x_1)(y_2 - y) + f(Q_{12})(x_2 - x)(y - y_1) + f(Q_{22})(x - x_1)(y - y_1) \right) \\ &= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}. \end{aligned}$$

Note that we will arrive at the same result if the interpolation is done first along the y direction and then along the x direction.[1]

```
C:\Users\kevsi\Desktop\Project 3>javac BilinearInterpolation.java

C:\Users\kevsi\Desktop\Project 3>java BilinearInterpolation
Original approach of Bilinear Interpolation took: 550307400 NS.

C:\Users\kevsi\Desktop\Project 3>javac bpParallel.java

C:\Users\kevsi\Desktop\Project 3>java bpParallel
Parallelized approach of Bilinear Interpolation took: 244006900 NS.
```

To summarize, Bilinear Interpolation utilizes linear interpolation across 2 directions in order to enlarge an image on a 2D plane. The algorithm is mathematically quite advanced and would appear benefit immensely from being run in parallel as it is a multifaceted procedure. Before converting this code to parallel, one thing we observed is that there is a triple-nested for loop present. The two outer loops are for the width and the height of the image while the innermost loop handles the RGB aspects of the image. The runtime

of the original code took 550307400 nanoseconds to execute on the stock Lenna100.jpg image from Rosetta Code. The following is the original code we were provided from Rosetta Code with a minor addition of some runtime-recording code:

```java
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

class BilinearInterpolation {
    /* gets the 'n'th byte of a 4-byte integer */
    private static int get(int self, int n) {
        return (self >> (n * 8)) & 0xFF;
    }

    private static float lerp(float s, float e, float t) {
        return s + (e - s) * t;
    }

    private static float blerp(final Float c00, float c10, float c01, float c11, float tx, float ty) {
        return lerp(lerp(c00, c10, tx), lerp(c01, c11, tx), ty);
    }

    private static BufferedImage scale(BufferedImage self, float scaleX, float scaleY) {
        int newWidth = (int) (self.getWidth() * scaleX);
        int newHeight = (int) (self.getHeight() * scaleY);
        BufferedImage newImage = new BufferedImage(newWidth, newHeight, self.getType());
        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);
            }
        }
        return newImage;
    }

    public static void main(String[] args) throws IOException {
        long start = System.nanoTime(); //runtime analysis
```

```
        long end,dt;
        File lenna = new File("Lenna100.jpg");
        BufferedImage image = ImageIO.read(lenna);
                //BufferedImage image2 = scale(image, 3.0f, 3.0f);
        BufferedImage image2 = scale(image, 1.6f, 1.6f);
        File lenna2 = new File("Lenna100_larger.jpg");
        ImageIO.write(image2, "jpg", lenna2);
        end = System.nanoTime();
        dt = end - start;
        System.out.println("Original approach of Bilinear Interpolation took: " + dt + " NS.");
    }
}
```

3 Bilinear Interpolation Parallel:

The changes we made to make the code run in parallel were not too drastic: we converted the 3 for loops to IntStreams and changed "int rgb;" into an AtomicInteger in order to allow us to modify it within our lambdas. The execution of this parallel code took 244006900 nanoseconds, which is less than half of the runtime of the original code. To further investigate, when we increased the image size by 300 percent instead of 60 percent (1.6f to 3.0f), the run-time of the parallel approach took 156911400 NS while the serial approach took 331715900 NS. Furthermore, we tested this algorithm on a larger image (1000x1000) to see if this would still incur a benefit for the parallelized version and we were faced with the same behaviour where the parallel approach was vastly superior. This was something we expected initially, as it seems to indicate that the serial method for Bilinear Interpolation is much slower regardless of parameter size because the algorithm apparently is quite complex and seems to only run faster when the loops are converted to parallel streams.

```
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.stream.IntStream;
import java.util.concurrent.atomic.AtomicInteger; //trick lambda
class BilinearInterpolation2 {
    /* gets the 'n'th byte of a 4-byte integer */
    private static int get(int self, int n) {
        return (self >> (n * 8)) & 0xFF;
    }
    private static float lerp(float s, float e, float t) {
        return s + (e - s) * t;
    }
    private static float blerp(final Float c00, float c10, float c01, float c11, float tx, float ty) {
        return lerp(lerp(c00, c10, tx), lerp(c01, c11, tx), ty);
    }
    private static BufferedImage scale(BufferedImage self, float scaleX, float scaleY) {
        int newWidth = (int)(self.getWidth() * scaleX);
        int newHeight = (int)(self.getHeight() * scaleY);
        BufferedImage newImage = new BufferedImage(newWidth, newHeight, self.getType());
        IntStream.range(0, newWidth).parallel().forEach(x -> { //parallelized for-each loop.
            IntStream.range(0, newHeight).parallel().forEach(y -> { //nested parallelized for-each loop
                float gx = ((float) x) / newWidth * (self.getWidth() - 1);
                float gy = ((float) y) / newHeight * (self.getHeight() - 1);
                int gxi = (int) gx;
```

```
                int gyi = (int) gy;
                AtomicInteger rgb = new AtomicInteger(0); //trick lambda
                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);
                IntStream.rangeClosed(0, 2).parallel().forEach(i -> { // nested nested parallelized for
                    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.set( rgb.get() | ble );
                });
                newImage.setRGB(x, y, rgb.get());
            });
        });
        return newImage;
    }
    public static void main(String[] args) throws IOException {
        long start = System.nanoTime(); //runtime analysis
        long end, dt;
        File lenna = new File("Lenna100.jpg");
        BufferedImage image = ImageIO.read(lenna);
        //BufferedImage image2 = scale(image, 3.0f, 3.0f);
        BufferedImage image2 = scale(image, 1.6f, 1.6f);
        File lenna2 = new File("Lenna100_larger.jpg");
        ImageIO.write(image2, "jpg", lenna2);
        end = System.nanoTime();
        dt = end - start;
        System.out.println("Parallelized approach of Bilinear Interpolation took: " + dt + " NS.");
    }
}
```

4 Gaussian Elimination Serial:

$$\begin{aligned} 2x + y - z &= 8 && (L_1) \\ -3x - y + 2z &= -11 && (L_2) \\ -2x + y + 2z &= -3 && (L_3) \end{aligned}$$

System of equations	Row operations	Augmented matrix
$\begin{aligned} 2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$		$\left[\begin{array}{rrr\|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$
$\begin{aligned} 2x + y - z &= 8 \\ \tfrac{1}{2}y + \tfrac{1}{2}z &= 1 \\ 2y + z &= 5 \end{aligned}$	$\begin{aligned} L_2 + \tfrac{3}{2}L_1 &\to L_2 \\ L_3 + L_1 &\to L_3 \end{aligned}$	$\left[\begin{array}{rrr\|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 2 & 1 & 5 \end{array}\right]$
$\begin{aligned} 2x + y - z &= 8 \\ \tfrac{1}{2}y + \tfrac{1}{2}z &= 1 \\ -z &= 1 \end{aligned}$	$L_3 + -4L_2 \to L_3$	$\left[\begin{array}{rrr\|r} 2 & 1 & -1 & 8 \\ 0 & \frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & -1 & 1 \end{array}\right]$
The matrix is now in echelon form (also called triangular form)		
$\begin{aligned} 2x + y &= 7 \\ \tfrac{1}{2}y &= \tfrac{3}{2} \\ -z &= 1 \end{aligned}$	$\begin{aligned} L_2 + \tfrac{1}{2}L_3 &\to L_2 \\ L_1 - L_3 &\to L_1 \end{aligned}$	$\left[\begin{array}{rrr\|r} 2 & 1 & 0 & 7 \\ 0 & \frac{1}{2} & 0 & \frac{3}{2} \\ 0 & 0 & -1 & 1 \end{array}\right]$
$\begin{aligned} 2x + y &= 7 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} 2L_2 &\to L_2 \\ -L_3 &\to L_3 \end{aligned}$	$\left[\begin{array}{rrr\|r} 2 & 1 & 0 & 7 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$
$\begin{aligned} x &= 2 \\ y &= 3 \\ z &= -1 \end{aligned}$	$\begin{aligned} L_1 - L_2 &\to L_1 \\ \tfrac{1}{2}L_1 &\to L_1 \end{aligned}$	$\left[\begin{array}{rrr\|r} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & -1 \end{array}\right]$

Gaussian Elimination is a very simple algorithm that we learned in Linear Algebra (also known as row reduction). This algorithm is mainly used to calculate the unknowns in a set of linear equations and is quite elementary. Therefore, we were not very inclined to think this code would benefit in any way from being run in parallel with the sample matrix provided by Rosetta Code. The original approach took 4088200 NS to execute on a 5x5 matrix and is as follows:

```
import java.util.Locale;

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) {
      long start = System.nanoTime(); //runtime analysis
        long end,dt;
        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}};

        double[] x = {39.0 / 400.0,
                      11.0 / 100.0,
                      31.0 / 240.0,
                      37.0 / 300.0,
                      71.0 / 400.0};
                /*
                //6x6 matrix
                double[][] a = new double[][] {
        {1.00, 0.00, 0.00, 0.00, 0.00, 0.00},
        {1.00, 0.63, 0.39, 0.25, 0.16, 0.10},
        {1.00, 1.26, 1.58, 1.98, 2.49, 3.13},
        {1.00, 1.88, 3.55, 6.70, 12.62, 23.80},
        {1.00, 2.51, 6.32, 15.88, 39.90, 100.28},
        {1.00, 3.14, 9.87, 31.01, 97.41, 306.02}
                };

        double[][] b = new double[][] {{-0.01}, {0.61}, {0.91}, {0.99}, {0.60}, {0.02}};

        double[] x = {-0.01, 1.602790394502114, -1.6132030599055613,
            1.2454941213714368, -0.4909897195846576, 0.065760696175232};
                */
        System.out.println("det: " + solve(a, b));

                //for (int i = 0; i < 6; i++) {
        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]);
        }
        end = System.nanoTime();
        dt = end - start;
        System.out.println("Original approach of Guassian Elimination took: " + dt + " NS.");
    }
}
```

5 Gaussian Elimination Parallel:

This parallelized approach features similar changes to the Bilinear Interpolation code, but there is an additional class we created called AtomicFloat in order to allow for modification of floating point variables within the scope of our IntStreams. Moreover, as we had initially expected, this code did not benefit from being run in parallel with the original 5x5 matrix. On the contrary, it actually had a significantly increased run-time of 50538600 NS due to the added overhead of parallelization. As stated earlier, Gaussian Elimination is quite simple and all of the for-loops present in this code performed relatively simple computations per iteration. To further investigate, we converted the 5x5 matrix to a 6x6, the serial approach took 51222700 NS while the parallel approach took 4477400 NS, showing that the parallel approach is only a worth-wile option when we are presented with a relatively large parameter size in the case of Gaussian Elimination.

```java
import java.util.Locale;
import java.util.stream.IntStream;
import java.util.concurrent.atomic.AtomicInteger;
import static java.lang.Float.*;

class AtomicFloat extends Number { //allows for atomic floating point numbers
    private AtomicInteger bits;
    public AtomicFloat() {
        this(0f);
    }
    public AtomicFloat(float initialValue) {
        bits = new AtomicInteger(floatToIntBits(initialValue));
    }
    public final boolean compareAndSet(float expect, float update) {
        return bits.compareAndSet(floatToIntBits(expect),
                                  floatToIntBits(update));
    }
    public final void set(float newValue) {
        bits.set(floatToIntBits(newValue));
    }
    public final float get() {
        return intBitsToFloat(bits.get());
    }
    public float floatValue() {
        return get();
    }
    public final float getAndSet(float newValue) {
        return intBitsToFloat(bits.getAndSet(floatToIntBits(newValue)));
    }
    public final boolean weakCompareAndSet(float expect, float update) {
        return bits.weakCompareAndSet(floatToIntBits(expect),
                                      floatToIntBits(update));
    }
    public double doubleValue() { return (double) floatValue(); }
    public int intValue()       { return (int) get();           }
    public long longValue()     { return (long) get();          }
}

class GaussianElimination2 {
    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");
        }
        AtomicFloat det = new AtomicFloat(1.0f); //trick lambda
        IntStream.range(0, n - 1).forEach(i -> {  //nested parallalized for each loop
            AtomicInteger k = new AtomicInteger(i); //trick lambda
            IntStream.range(i + 1, n).parallel().forEach(j -> { //nested parallalized for each loop
                if (Math.abs(a[j][i]) > Math.abs(a[k.get()][i])) k.set(j);
            });
```

```
            if (k.get() != i) {
                det.set(-det.get());
                IntStream.range(i, n).parallel().forEach(j -> { //nested parallalized for each loop
                    double s = a[i][j];
                    a[i][j] = a[k.get()][j];
                    a[k.get()][j] = s;
                });
                IntStream.range(0, p).parallel().forEach(j -> { //nested parallalized for each loop
                    double s = b[i][j];
                    b[i][j] = b[k.get()][j];
                    b[k.get()][j] = s;
                });
            }
            IntStream.range(i + 1, n).parallel().forEach(j -> { //nested parallalized for each loop
                double s = a[j][i] / a[i][i];
                AtomicInteger t = new AtomicInteger(i+1); //trick lambda
                IntStream.range(t.get(), n).parallel().forEach(w -> { //nested parallalized for each lo
                    a[j][t.get()] -= s * a[i][t.get()];
                    t.set(t.get()+1);
                });
                IntStream.range(0, p).parallel().forEach(w -> { //nested parallalized for each loop
                    b[j][w] -= s * b[i][w];
                });
            });
        });
        int w = -n;
        IntStream.range(w + 1, 1).forEach(i -> { //nested parallalized for each loop
            i = -i;
            AtomicInteger t = new AtomicInteger(i); //trick lambda
            IntStream.range(i + 1, n).parallel().forEach(j -> { //nested parallalized for each loop
                double s = a[t.get()][j];
                IntStream.range(0, p).parallel().forEach(k -> { //nested nested parallalized for each l
                    b[t.get()][k] -= s * b[j][k];
                });
            });
            double s = a[t.get()][t.get()];
            det.set(det.get() * (float)s);
            IntStream.range(0, p).parallel().forEach(k -> { //nested parallalized for each loop
                b[t.get()][k] /= s;
            });
        });
        return det.doubleValue();
    }
    public static void main(String[] args) {
        long start = System.nanoTime(); //runtime analysis
        long end,dt;
                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}};

        double[] x = {39.0 / 400.0,
                      11.0 / 100.0,
                      31.0 / 240.0,
                      37.0 / 300.0,
                      71.0 / 400.0};
        /*
                //6x6 matrix
                double[][] a = new double[][] {
        {1.00, 0.00, 0.00, 0.00, 0.00, 0.00},
        {1.00, 0.63, 0.39, 0.25, 0.16, 0.10},
        {1.00, 1.26, 1.58, 1.98, 2.49, 3.13},
        {1.00, 1.88, 3.55, 6.70, 12.62, 23.80},
        {1.00, 2.51, 6.32, 15.88, 39.90, 100.28},
        {1.00, 3.14, 9.87, 31.01, 97.41, 306.02}
                };

        double[][] b = new double[][] {{-0.01}, {0.61}, {0.91}, {0.99}, {0.60}, {0.02}};

        double[] x = {-0.01, 1.602790394502114, -1.6132030599055613,
            1.2454941213714368, -0.4909897195846576, 0.065760696175232};
                */
        System.out.println("det: " + solve(a, b));
                //IntStream.range(0, 6).parallel().forEachOrdered( i -> { //parallelize this simple loo
        IntStream.range(0, 5).parallel().forEachOrdered( i -> { //parallelize this simple loop for good
            System.out.printf(Locale.US, "%12.8f %12.4e%n", b[i][0], b[i][0] - x[i]);
        });
        end = System.nanoTime();
        dt = end - start;
        System.out.println("Parallelized approach of Gaussian Elimination took: " + dt + " NS.");
    }
}
```

6 Conclusion:

To conclude, after completing this project, we have learned that a parallel approach is usually a feasible option when writing code to solve large problems, but is not practical when the input parameters are small because the added overhead may actually increase run-time. With Bilinear Interpolation, the run-time was always immensely faster when we used parallel IntStreams and even performed twice as fast when the parallel approach was used on a very small image. Lastly, we observed that Gaussian Elimination was considerably slower when ran in parallel with a small input matrix, but performed faster when a larger 6x6 matrix was used as the input parameter.