Parallel Tasks

Timothy Lee, John Aquino, Michael Merabi

May 17, 2019

1 Introduction

This project includes optimizations using OpenMP and Java Streams. Both the Gaussian Elimination and Bilinear Interpolation was taken from Rosetta code to be optimized. Benchmarks of the runtimes were taken to provide proof of optimizations.

ALL BENCHMARKS WERE RECORDED WITH AN i5 6600k

2 Bilinear Interpolation

Bilinear Interpolation is an extension of linear interpolation that functions by scaling each pixel of the original image to a certain position. Bilinear Interpolation uses values of the nearest 4 pixels in diagonal directions to find the RGB value of that specific pixel. A problem that occurs during this process is that while the image is being scaled, certain pixels are not completely filled (i.e, holes). In order to fill these "holes", interpolation is used.

2.1 Optimizing (OpenMP)

To optimize the code in OpenMP for Bilinear Interpolation, we analyzed the serial run time and then we looked for the block of code with the longest runtime. After concluding that the scale method took the longest, we proceeded to optimize it by replacing the block with openMP. After testing different combinations of different work sharing constructs, we concluded that using the single work-sharing construct with tasks was the best way to improve the program and our changes significantly improved the runtime of the program.

```c
#include <stdint.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

```

```
#include <omp.h>
double ctime1, ctime2;

typedef struct {
    uint32_t *pixels; // pointer to array of pixels
    unsigned int w;
    unsigned int h;
} image_t;

#define getByte(value, n) (value >> (n*8) & 0xFF)

uint32_t getpixel(image_t *image, unsigned int x, unsigned int y){
    return image->pixels[(y*image->w)+x];
}
float lerp(float s, float e, float t){return s+(e-s)*t;}

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

void putpixel(image_t *image, unsigned int x, unsigned int y, uint32_t color){
    image->pixels[(y*image->w) + x] = color;
}

void scale(image_t *src, image_t *dst, float scalex, float scaley){
    int newWidth = (int)src->w*scalex;
    int newHeight= (int)src->h*scaley;
    int x, y;

    ctime1 = omp_get_wtime();
    int nthreads, tid;

    #pragma omp parallel //shared(nthreads) private(x, y, tid)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();

        //#pragma omp single private(tid, nthreads)
        #pragma omp single
        // #pragma omp for
        for(x= 0, y=0; y < newHeight; x++){
            if(x > newWidth){
                x = 0; y++;
            }

            float gx = x / (float)(newWidth) * (src->w-1);
            float gy = y / (float)(newHeight) * (src->h-1);
            int gxi = (int)gx;
            int gyi = (int)gy;
            uint32_t result=0;
            uint32_t c00 = getpixel(src, gxi, gyi);
            uint32_t c10 = getpixel(src, gxi+1, gyi);
            uint32_t c01 = getpixel(src, gxi, gyi+1);
            uint32_t c11 = getpixel(src, gxi+1, gyi+1);
            uint8_t i;

            //printf("Thread num is : %d\n", tid);
            //printf("Num threads: %d\n", nthreads);

            #pragma omp task
            // #pragma omp critical
            for(i = 0; i < 3; i++){
                //((uint8_t*)&result)[i] = blerp( ((uint8_t*)&c00)[i], ((uint8_t*)&c10)[i],
                result |= (uint8_t)blerp(getByte(c00, i), getByte(c10, i), getByte(c01, i),


            }
            // #pragma omp critical
            #pragma omp task
            putpixel(dst,x, y, result);
        }

    }
    ctime2 = omp_get_wtime();

    printf("Time for parallel region is: %f\n", ctime2 - ctime1);

}
//XGA (1024×768)

//pixels = malloc(picHeight * sizeof(*pixels)*3);
int main() {
    // clock timer
    double time_spent = 0.0;
    double totaltime = 0.0;
//////////////////////////////////////////////////

    int i ;
// array = calloc(n, sizeof(int));

    image_t *imagein = malloc(sizeof(image_t));
    image_t *imageout = malloc(sizeof(image_t));

    imagein->pixels = malloc(sizeof(uint32_t) * sizeof(*imagein));

    imageout->pixels = malloc(sizeof(uint32_t) * sizeof(*imagein));

    clock_t begin = clock();
    scale(imagein, imageout, 1.6f, 1.6f);
    clock_t end = clock();

    totaltime =  (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Total time: %lf\n", totaltime);

    // printf("imagein->pixels: %u \n", imagein->pixels);
    // printf("imagein:  %u \n", imagein);
    // printf("imageout->pixels: %u \n", imageout->pixels);

    free(imagein);
    free(imageout);


    return 0;
}
```

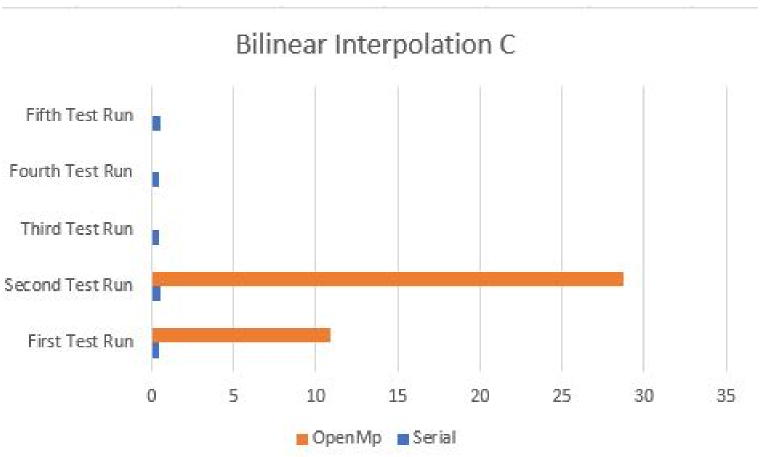

Figure 1: Benchmark for Bilinear Interpolation (C)

2.2 Optimizing (Java Streams)

To optimize the code in parallel streams for Bilinear Interpolation, we first took the approach of analyzing where in the code it would take the longest to run. We found that because the bulk of the work is being done in the scale method (which does things serially), with nested loops, we decided to optimize the loops with parallel streams. Using the knowledge we learned in class, we used the approach of creating a nested IntStream with the ranges of the loops, then we finished with a forEach to traverse each element. We found that trying to optimize the inner loop (3rd loop) faced us with many issues due to it having many dependencies. Even though we could not completely integrate the streams, we found great improvement in the run time of our code. Benchmarks provided below.

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

// imports for streams
import java.util.*;
import java.util.stream.*;

public 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());

        IntStream.range(0, newWidth).parallel().forEach(x -> {
                IntStream.range(0, newHeight).parallel().forEach(y -> {

```

```
        //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);
                //IntStream.range(0, 2).parallel().forEach(i -> {
                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 {
        File google = new File("google_image.jpg");
        BufferedImage image = ImageIO.read(google);
        BufferedImage image2 = scale(image, 1.6f, 1.6f);
        File google2 = new File("google_image_larger.jpg");
        ImageIO.write(image2, "jpg", google2);

        long endTime = System.nanoTime();

        long totalTime = endTime - startTime;
        System.out.println(totalTime + " ns");

        // first test run no optimization
        // 337587138 NS = 0.33758138 seconds
        // first test run with parallel streams
        // 6552543 NS = 0.006552543 seconds
        //  faster by 0.331028837 seconds
        // second test run with paralllel streams
        // 6409054 NS = 0.006409054 seconds
        // CONCLUSION : parallel streams made the program much faster by
    }
}
```

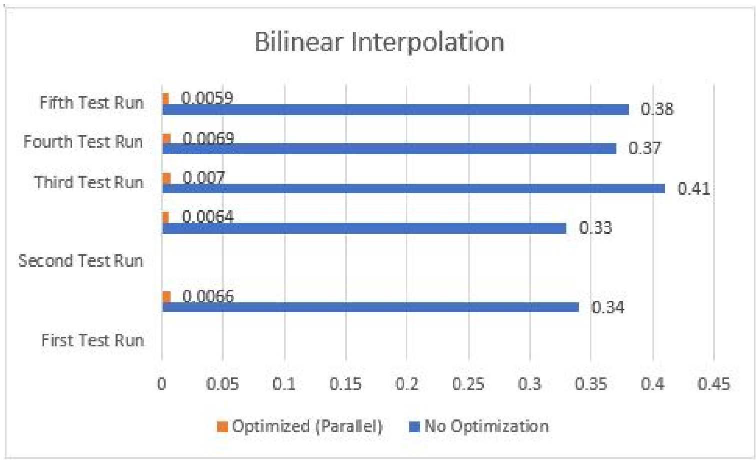

Figure 2: Benchmark for Bilinear Interpolation (Java)

3 Gaussian Elimination (Java Only)

Gaussian Elimination is an algorithm for solving systems of linear equations in linear algebra. The work is done by using elementary row operations on the augmented matrix formed from the linear equations. The row operations include, swapping rows, multiplying rows by a non-zero scalar, and adding a row to another row. In computer science, Gaussian Elimination has a Big O complexity of (n^3).

3.1 Optimizing

With the complex nature of Gaussian Elimination, trying to get it running faster is difficult to say the least. Making the program run in parallel is close to impossible because of how the equations interact with each other while calculating operations. So we determined the fastest method is actually having the program run serially but through a stream so that each operation can be done automatically on the reduction of all vectors. This approach allows for a much cleaner view of the code that eliminates lots of repetitive cycling in favor of purer functionality.

3.2 Optimizing (Java)

```java
// imports for streams
import java.util.*;
import java.util.stream.*;

// translated code from C# to java
class Vector
{
    private double[] b;
    public final int rows;

    public Vector(int rows)
    {
        this.rows = rows;
        b = new double[rows];
    }

    public Vector(double[] initArray)
    {
        b = (double[])initArray.clone();
        rows = b.length;
    }

    public Vector Clone()
    {
        Vector v = new Vector(b);
        return v;
    }

    public double get(int row)
    {
        return b[row];
    }
    public void set(int row, double value)
    {
        b[row] = value;
    }

    public void SwapRows(int r1, int r2)
    {
        if (r1 == r2)
        {
            return;
        }
        double tmp = b[r1];
        b[r1] = b[r2];
        b[r2] = tmp;
    }

    private double norm(double[] weights)
    {
        double sum = 0;
        for (int i = 0; i < rows; i++)
        {
            double d = b[i] * weights[i];
            sum += d * d;
        }
        return Math.sqrt(sum);
    }

    public void print()
    {
        for (int i = 0; i < rows; i++)
        {
            System.out.println(b[i]);
        }
        System.out.println();
    }

    private static Vector Subtract(Vector lhs, Vector rhs)
    {
        Vector v = new Vector(lhs.rows);
        for (int i = 0; i < lhs.rows; i++)
        {
            v.set(i, lhs.get(i) - rhs.get(i));
        }
        return v;
    }
}
class Matrix
{
    private double[][] b;
    private final int rows, cols;

    public Matrix(int rows, int cols)
    {
```

```java
        this.rows = rows;
        this.cols = cols;
        double[][] b = new double[rows][cols];
    }

    public Matrix(int size)
    {
        this.rows = size;
        this.cols = size;
        double[][] b = new double[rows][cols];
        for (int i = 0; i < size; i++)
        {
            this.setValue(i, i, 1);
        }
    }

    public Matrix(int rows, int cols, double[][] initArray)
    {
        this.rows = rows;
        this.cols = cols;
        b = (double[][])initArray.clone();
        if (b.length != rows || b.length !=cols)
        {
            throw new RuntimeException("bad array");
        }
    }

    private double getValue(int row, int col)
    {
        return b[row][col];
    }

    private void setValue(int row, int col, double value)
    {
        b[row][col] = value;
    }

    private static Vector Multiply(Matrix lhs, Vector rhs)
    {
        long startTime = System.nanoTime();


        if (lhs.cols != rhs.rows)
        {
            throw new RuntimeException("Can't multiply matrix by a vector");
        }
        Vector v = new Vector(lhs.rows);
        for (int i = 0; i < lhs.rows; i++)
        {
            double sum = 0;
            for (int j = 0; j < rhs.rows; j++)
            {
                sum += lhs.getValue(i,j) * rhs.get(j);
            }
            v.set(i, sum);
        }

        long endTime = System.nanoTime();
        long totalTime = endTime - startTime;
        System.out.println("Time for Multiply" + totalTime + " ns");

        return v;
    }
        private void SwapRows(int r1, int r2)
    {
        if (r1 == r2)
        {
            return;
        }
        // IntStream.range(0, cols).parallel().forEach(x ->
        for (int i = 0; i < cols; i++)
        {
            double tmp = b[r1][i];
            b[r1][i] = b[r2][i];
            b[r2][i] = tmp;
        }
        // )};
    }

    //with partial pivot
    public final void ElimPartial(Vector B)
    {

            long startTime = System.nanoTime();

        IntStream.range(0, rows).forEach(diag -> {
        // for (int diag = 0; diag < rows; diag++)
        // {
                int max_row = diag;
                double max_val = Math.abs(b[diag][diag]);
            double d;

```

```java
            // IntStream.range(diag+1, rows).parallel().forEach(row -> {
                for (int row = diag + 1; row < rows; row++)
                {
                    if ((d = Math.abs(b[row][diag])) > max_val)
                    {
                        max_row = row;
                        max_val = d;
                    }
            // });
            }
                SwapRows(diag, max_row);

            B.SwapRows(diag, max_row);

                double invd = 1 / b[diag][diag];

            for (int col = diag; col < cols; col++)
                {
                    b[diag][col] *= invd;
                }

            B.set(diag,B.get(diag)*invd);

                for (int row = 0; row < rows; row++)
                {
                    d = b[row][diag];
                    if (row != diag)
                    {
                        for (int col = diag; col < cols; col++)
                        {
                            b[row][col] -= d * b[diag][col];
                        }
                        B.set(row,(B.get(row) - (d * B.get(diag))));
                    }
            }
        });
        // }

        long endTime = System.nanoTime();
        long totalTime = endTime - startTime;
        System.out.println(totalTime / 1_000_000_000.0 + " sec");
    }

    public final void print()
    {
        IntStream.range(0, rows).forEach(i ->{
            IntStream.range(0, cols).forEach( j-> {
        // for (int i = 0; i < rows; i++)
        // {
        //     for (int j = 0; j < cols; j++)
        //     {
        //     }
        //     System.out.println();
        // }
                System.out.println();
            });
        });
    }
}

class GaussianElimination{
        public static void main(String[] args){

                long startTime = System.nanoTime();

            double[][]a={{1.1, 0.12, 0.13, 0.12, 0.14, -0.12},
                                            {1.21, 0.63, 0.39, 0.25, 0.16, 0.1},
                                            {1.03, 1.26, 1.58, 1.98, 2.49, 3.13},
                                            {1.06, 1.88, 3.55, 6.7, 12.62, 23.8},
                                            {1.12, 2.51, 6.32, 15.88, 39.9, 100.28},
                                            {1.16, 3.14, 9.87, 31.01, 97.41, 306.02}
                        };
                Matrix A = new Matrix(6, 6, a);
                Vector B = new Vector(new double[] {-0.01, 0.61, 0.91, 0.99, 0.60, 0.02});
                A.ElimPartial(B);
                B.print();

            long endTime = System.nanoTime();
        long totalTime = endTime - startTime;
        System.out.println(totalTime / 1_000_000_000.0 + " sec");
    }
}
// // {output}
// // -0.05973910275019 76
// // 1.850189667262 78
// // -1.972783301811 63
// // 1.469758775065 1
// // -0.553874184782179
// // 0.0723048745759396
```