COP5618/CIS4930 Concurrent Programming

Assignment 3

Due Wednesday, December 7 at 11:59pm.
Accepted until Friday, December 9 at 11:59pm with 1% late penalty.
Submissions will be closed on December 9 at 11:59pm.

This assignment involves

1. implementing parallelism using the Java fork/join framework
2. Java 8 stream processing
3. determining a bound on speedup using Ahmdahl's law
4. Instrumenting code and performing experiments

The setting will be some transformations on images using Java's BufferedImage and related classes.

For this assignment, the following facts about

To read an image from a file using `javax.imageio.ImageIO`, and `java.io.File`:

```
File sourceFile = new File(sourceImageFilename);
sourceImage = ImageIO.read(sourceFile);
```

To write an image to a file

```
File outputFile = new File(filename);
ImageIO.write(newImage, "jpg", outputFile);
```

Images are viewed as a 2d array of pixels, with (0,0) in the top left corner. The horizontal axis is typically referred to as x, the vertical by y.

In order to manipulate pixels, which can be stored in many ways, you want them in the default RGB color model, where each pixel has four 8-bit components, representing alpha, red, green, and blue components of the pixel's color, packed into a single integer. Alpha is the transparency. In this assignment, we won't do anything with the alpha component, but if you need to specify it, use 255.

The getRGB and setRGB methods defined in BufferedImage get and set either a single pixel, or a 1d array of pixels. If necessary, these methods transform pixels to/from the sRGB ColorModel.

A ColorModel instance encapsulates the pixel representation and provides methods to extract the individual components. For pixels that have been obtained with the getRGB method, you can use the ColorModel returned from the ColorModel.getRGBdefault() method. Then use the getRed, getGreen, getBlue methods to obtain the values of those color components from the pixel. To pack individual color components into an int, create an int array containing the colors and invoke getDataElement.

```
ColorModel colorModel = ColorModel.getRGBdefault();
int red  = colorModel.getRed(pixel);
…
int[] components = {red,green,blue,alpha};
int pixel2 = colorModel.getDataElement(components,0);
```

Alternatively, since we know that we are in the default RGB Color Model, we can just create the pixel directly and avoid having to create the int array:

```
static int makeRGBPixel(int red, int green, int blue) {
       int pixel = ((255 & 0xFF) << 24) /* alpha component */
                     | ((red & 0xFF) << 16) /* red component */
                     | ((green & 0xFF) << 8) /* green component */
                     | ((blue & 0xFF) << 0) /* blue component */;
       return pixel;
}
```

In this assignment, we will generally read an image from a file, get its pixels, compute new pixel values, create a new image and set its pixels to the new values, then write the new image to a file. We will use Java 8 Streams to compute the new pixels. An example which converts a color image to grayscale is given below.

```
public static void gray_SS(BufferedImage image, BufferedImage newImage) {
       ColorModel colorModel = ColorModel.getRGBdefault();
       int w = image.getWidth();
       int h = image.getHeight();

       //Get pixels from source image
       int[] sourcePixelArray = image.getRGB(0, 0, w, h, new int[w*h], 0, w);

       int[] grayPixelArray =
              //convert array of pixels to a stream
              Arrays.stream(sourcePixelArray)
              //combine red, green, and blue values to obtain gray value
              .map(pixel -> (int) ((colorModel.getRed(pixel) * .299) +
                    (colorModel.getGreen(pixel) * .587) +
                    (colorModel.getBlue(pixel) * .114)))
              //make new pixel where all three colors have the same gray value
              .map(grayVal -> makeRGBPixel(grayVal, grayVal, grayVal))
              //convert stream to array
              .toArray();

       // set pixels of newImage to the grayPixelArray
       newImage.setRGB(0, 0, w, h, grayPixelArray, 0, w);
}
```

The caller read an image source from a file, and created a matching newImage:

```
BufferedImage newImage = new BufferedImage(source.getWidth(),
source.getHeight(), source.getType());
```

There are three significant parts to this routine: getRGB, the stream computation, and setRGB.

We want to instrument the code and use Ahmdahl's law to determine how much speedup we could expect by parallelizing any of these three parts of the code.

First, instrument the code to measure the time taken by each of these parts. For the sake of uniformity, a class called Timer is provided for you. This takes a list of Strings that serve as labels for various durations in the code. Then we insert calls to the now() method to record the time. If there are N labels, then an array of N+1 longs holds results from System.nanoTime(). The toString method returns a String with the total time between the first and last call to now. Also, label[0] and the elapsed time in milliseconds between the first and second calls to now(), label[1] with the elapsed time between second and third calls to now(), etc. are printed along with the percent of the total time for each duration.

The example shown earlier could be instrumented as follows:

```
public static String[] grayLabels = {"getRGB", "toGrayPixaleArray", "setRGB"};

public static Timer gray_SS(BufferedImage image, BufferedImage newImage) {
       Timer timer = new Timer(grayLabels);
       ColorModel colorModel = ColorModel.getRGBdefault();
       int w = image.getWidth();
       int h = image.getHeight();
       timer.now();
       int[] sourcePixelArray = image.getRGB(0, 0, w, h, new int[w*h], 0, w);
       timer.now(); //label = getRGB
       int[] grayPixelArray =
                     //get array of pixels from image and convert to stream
                     Arrays.stream(sourcePixelArray)
                     //combine red, green, and blue values to obtain gray value
                     .map(pixel -> (int) ((colorModel.getRed(pixel) * .299) +
                          (colorModel.getGreen(pixel) * .587) +
                          (colorModel.getBlue(pixel) * .114)))
                     //make new pixel, all components have the same gray value
                     .map(grayVal -> makeRGBPixel(grayVal, grayVal, grayVal))
                     //convert stream to array
                     .toArray();
       timer.now(); // label = toGrayPixaleArray
       //create a new Buffered image and set its pixels to the gray pixel array
       newImage.setRGB(0, 0, w, h, grayPixelArray, 0, w);
       timer.now(); //label = setRGB
```

```java
            return time;
    }
```

1. Execute this method several times and print the results. What do you notice about the times? What does this tell you about timing programs written in Java? You can use the serial test case in the provided HW3TestGray class to do this, just uncomment the print statement.

You should have noticed that the first few times this is executed take longer. In general, several things can happen while the JVM is getting "warmed up". This may include loading classes, which may not happen until the first time it is needed. Also, the Hot Spot compiler in the JVM may do dynamic optimization based what happens at runtime. So the first couple of times through the code may not be optimized yet.

2. Run your test when with as little as possible happening on your computer. Then try again while you are doing other activities, say watching a youtube video. What happens to the timings?

Times will go up as you share system resources (CPUs, caches, bus, etc.) with other applications.

3. Consider this loop where timerData is never again referenced.

```java
for (int rep = 0; rep < WARMUPREPS; rep++) {
     Timer timerData = GrayScale.gray_SS(source, newImage);
}
```

4. What concerns should we generally have about a loop like this in a benchmark? Is it a problem in this particular case? Why not?

A compiler might determine that the loop body is dead code and elide the loop. In this particular case, it probably won't happen since GrayScale.gray_SS is a procedure call that, as far as the compiler knows, may have side effects. Nevertheless, it is important when benchmarking code to ensure that you are measuring what you think you are.

5. Use Amdahl's law to compute an upper bound on the speedup you could expect from
   a. parallelizing only the stream processing portion of the code
   b. parallelizing only the setRGB and getRGB portions of the code
   c. parallelizing both stream processing and setRGB and getRGB portions of the code.

getRGB mean duration=316.1 msecs, stddev=30.8, mean percent of total=34
The following is data collected from the serial version. (Your data will not be exactly the same)

stream processing mean duration=254.2 msecs, stddev=20.3, mean percent of total=27
setRGB mean duration=361.8 msecs, stddev=39.5, mean percent of total=39
total mean duration=932 msecs, stddev=37.5

Ahmdahl's law says that the max speedup one can expect is $\frac{1}{\left(\gamma+\frac{1-\gamma}{P}\right)}$ where γ is the serial fraction and P is the number of processors. Or taking the limit as P → ∞, we get $\text{Sbound} = \frac{1}{\gamma}$.

a. Only parallelizing stream processing: Serial fraction = 1-.27 = .73, Thus with 4 procs the max speedup for parallelizing the stream processing part only would be 1.25. Or with an unbounded number of procs, 1.36.

b. Only parallelizing setRGB and getRGB: Serial fraction = 1-.39 = .61. With an unbounded number of procs, the max speedup is 1.63.

c. Parallelizing stream processing and set/getRGG: Serial fraction = 1 = .27 - .39 = .24. Max speedup is 2.94.

6. Implement the method gray_PS. This should do the stream processing step in parallel.

```
int[] grayPixelArray = Arrays.stream(sourcePixelArray)
.parallel() //THIS IS THE ONLY PART THAT IS DIFFERENT
.map(pixel -> (int) ((colorModel.getRed(pixel) * .299)
                + (colorModel.getGreen(pixel) * .587)
                + (colorModel.getBlue(pixel) * .114)))
.map(grayVal ->HW3Utils.makeRGBPixel(grayVal, grayVal, grayVal))
.toArray();
```

7. Consider parallelizing setRGB and getRGB. This cannot be done using stream processing. Which restrictions imposed by the Stream processing framework are violated by these methods?

Usually, functions involved in stream processing should be functional, and not have side effects.

8. Parallelize the setRGB and getRGB methods using a divide and conquer approach and Java's fork/join framework. To do this, fill in the missing methods in the given FJBufferedImage file. This class extends BufferedImage and overrides the setRGB and getRGB methods. Thus the only thing that needs to be done to change from serial to parallel is to create a FJBufferedImage instead of a BufferedImage.

Some things to think about:

a. How should the 2d image be partitioned in the divide phase? By rows? By columns? Something else? Does the presence of caches affect your answer?

Generally, we want data to be accessed in cache-friendly ways, which means that different threads (as much as possible) avoid simultaneously accessing data, even if logically different, that are on the same cache line. Since the data is stored by row, it makes sense to partition by row rather than by column, or even block.

b. What would be a sensible default threshold? Should it be a constant number of pixels? A constant number of rows or columns? Be a function of the number of threads (which can be obtained from the ForkJoinPool *pool*.getParallelism() method)? Experiment and choose one.

Several approaches might make sense as a starting point. A simple one is to divide the rows equally among the available threads.

A partial solution is given below. Only the getRBG is given, setRGB is similar.

```java
public class FJBufferedImage_solution extends BufferedImage {

	/**
	 * The singleton ForkJoinPool used by all instances of this class
	 */
	static ForkJoinPool pool = new ForkJoinPool();

	Various things omitted

	@Override
	public int[] getRGB(int xStart, int yStart, int w, int h, int[] rgbArray, int 
offset, int scansize){
		return getRGB(xStart,  yStart, w, h, rgbArray, offset, scansize, 
taskPerThread);
	}

	public int[] getRGB(int xStart, int yStart, int w, int h, int[] rgbArray, int 
offset, int scansize, Function<BufferedImage,Integer> thresholdCalc){
		FJBufferedImage_solution.ForkGetRGB fs = this.new 
ForkGetRGB(xStart,yStart,w,h,rgbArray,offset,scansize, thresholdCalc.apply(this));
		pool.invoke(fs);
		return rgbArray;
	}


	@SuppressWarnings("serial")
	class ForkGetRGB extends RecursiveAction{
		final int xStart;
		final int yStart;
		final int w;
		final int h;
		final int[] rgbArray;
		final int offset;
		final int scansize;
		final int threshold;

		ForkGetRGB(int xStart, int yStart, int w, int h, int[] rgbArray, int 
offset, int scansize){
			this.xStart = xStart;
			this.yStart = yStart;
			this.w = w;
			this.h = h;
			this.rgbArray = rgbArray;
			this.offset = offset;
			this.scansize = scansize;
			this.threshold = 
taskPerThread.apply(FJBufferedImage_solution.this);
		}


		ForkGetRGB(int xStart, int yStart, int w, int h, int[] rgbArray, int 
offset, int scansize, int threshold){
			this.xStart = xStart;
			this.yStart = yStart;
```

```
			this.w = w;
			this.h = h;
			this.rgbArray = rgbArray;
			this.offset = offset;
			this.scansize = scansize;
			this.threshold = threshold;
		}


		@Override
		protected void compute() {
			if (h <= threshold){
				
	FJBufferedImage_solution.super.getRGB(xStart,yStart,w,h,rgbArray,offset,scansi
ze);
			}
			else {
				int splitH= h/2;
				ForkGetRGB t1 = new ForkGetRGB(xStart, yStart, w, splitH, 
rgbArray, offset, scansize, threshold);
				ForkGetRGB t2 = new ForkGetRGB(xStart, yStart+splitH, w, 
h-splitH, rgbArray, offset + splitH*scansize, scansize, threshold);
				invokeAll(t1,t2);
			}


	}
	}

	}
```

9. Run tests to determine the speedup with parallel setRGB and getRGB and serial stream processing (gray_SS_FJ(…)) and parallel setRGB and getRGB, and parallel stream processing (gray_PS_FJ(…)).

COP5618 Students only

Implement a class similar to Gray.java except that it performs (almost) serial and parallel histogram equalization on an image. In some images, this improves the contrast. As an example, see LotsaGators3.jpg and colorHisEq_LotsaGators3.jpg.

Fill in the missing methods in ColorHistEq.java.

The transformation has the following steps:

1. Get RGB pixel array from the image
2. Convert the pixels into HSB format and save into a new array. In HSB format, each pixel is represented by a float array with 3 elements for hue, saturation, and brightness. Use the java.awt.Color.RGBtoHSB method.
3. Create a histogram of the brightness, which is a value between 0 and 1. Divide the space into a given number of bins, then count how many times a

brightness value falls in each bin. For example, if you had 4 bins, and 6 pixels with brightness values {.1,.2,.27,.80,.90,.2} the four bins hold [3,1,0,2] respectively.

4. Compute cumulative probability of a value falling this or an earlier bin. You can do this by computing the prefix sum of the bins, and dividing by the number of samples. For the previous example, the prefix sum is [3,4,4,6] and the cumulative probability is [3,4,4,6] [.5 , .66 , .66 ,1].
5. Create a new pixel array by replacing the brightness value in the HSB pixel you found earlier with the value in the cumulative probability array and convert to RGB The brightness values in our original pixel array (from step 3) become [.5, .5, .66, 1,1,.5]. Use the Color.HSBtoRGB method.
6. Create a new image with your new pixels.

Give an (almost) serial version and a parallel version.

Implementation notes:

- Call your methods colorHistEq_serial and colorHistEq_parallel.
- Use streams to convert to HSB format
- To create the histogram, take a stream of HSB pixels, extract the brightness element, use collect and Collectors.groupingBy. When you apply the stream map function, note that if the type converts from a primitive type to an Object, you need to use mapToObj, or from and Object to a primitive type, the mapToInt, or mapToDouble, etc. (When I did this, I used integers from [0,numBins) and let the brightness value belong to bin (int)brightness*numBins. Then the brightness level belonging to a bin is binNum/numBins.
- You can use Arrays.parallelPrefix to compute prefix sums in both your solutions.
- Use a stream to create the equalized pixel array in step 5.
- Set the pixels in a new image.
- The provided Timer class helps capture timing information. Pass an array of Strings containing labels of things that will be measured. Then instrument the code with labels.length+1 calls to the Timer’s now() method. Other methods should be self explanatory.

TURN IN: Gray.java, ColorHistEq.java, FJBufferedImage.java

You do not need to turn in the answers to the questions. However, some of them WILL appear on the final, so make sure to take them seriously.

Here’s the parallel version. The serial version has been omitted.

```
public class ColorHistEq {


      static String[] labels = { "getRGB", "convert to HSB", "create brightness
map", "probability array",
                     "parallel prefix", "equalize pixels", "setRGB" };
```

```java
	static Timer colorHistEq_parallel(FJBufferedImage image, FJBufferedImage
newImage) {
		ColorModel colorModel = ColorModel.getRGBdefault();
		Timer times = new Timer(labels);
		int numLevels = 256;
		int w = image.getWidth();
		int h = image.getHeight();
		int numPixels = w * h;
		times.now();
		int[] sourcePixelArray = image.getRGB(0, 0, w, h, new int[w * h], 0, w);
		times.now();
		// Obtain image's pixels and convert to HSB format. Save in array to
		// allow two traversals
		// In HSB format, each pixel is a float array with three elements
		// representing hue, saturation, and brightness.
		Object[] hsbPixels =
Arrays.stream(sourcePixelArray).parallel().mapToObj(pixel -> Color
				.RGBtoHSB(colorModel.getRed(pixel),
colorModel.getGreen(pixel), colorModel.getBlue(pixel), null))
				.toArray();
		times.now();

		/*
		 * Creates a map from brightness to number of times that value for
		 * brightness occurs
		 *
		 * First map the normalize value (i.e. in [0,1)) to an int value in
		 * [0,numLevels) by multiplying by numLevels-1 and truncating to int.
		 * (This is the floor because all values are positive)
		 *
		 * Then count the number with each value.
		 */
		Map<Integer, Long> frequencyMap = Arrays.stream(hsbPixels).parallel()
				.mapToInt(hsvPixel -> (int) (((float[]) hsvPixel)[2] *
(numLevels - 1)))
				.mapToObj(val ->
Integer.valueOf(val)).collect(Collectors.groupingBy(x -> x, Collectors.counting()));
		times.now();
		/*
		 * Obtain the cumulative probability array
		 *
		 * First construct an array with a (possibly 0) value for each element
		 * Then compute the parallel prefix sum of the array
		 */
		double[] cumulativeProb = IntStream.range(0, numLevels).parallel()
				.mapToDouble(
						value -> frequencyMap.get(value) != null ?
(double) frequencyMap.get(value) / numPixels : 0.0)
				.toArray();
		times.now();
		// compute the parallel prefix sum in place
		Arrays.parallelPrefix(cumulativeProb, Double::sum);
		times.now();
		// now create a new pixel array by replacing the brightness with the
		// value in the cumulativeProbability array
```

```java
		// and converting to RGB
		int[] equalizedPixels =
Arrays.stream(hsbPixels).parallel().mapToInt(hsbPixel -> {
			float hue = ((float[]) hsbPixel)[0];
			float saturation = ((float[]) hsbPixel)[1];
			float brightness = (float) cumulativeProb[(int) (((float[])
hsbPixel)[2] * (numLevels - 1))];
			return Color.HSBtoRGB(hue, saturation, brightness);
		}).toArray();
		times.now();
		// create a new image and set its pixels
		newImage.setRGB(0, 0, w, h, equalizedPixels, 0, w);
		times.now();
		return times;
	}
}
```