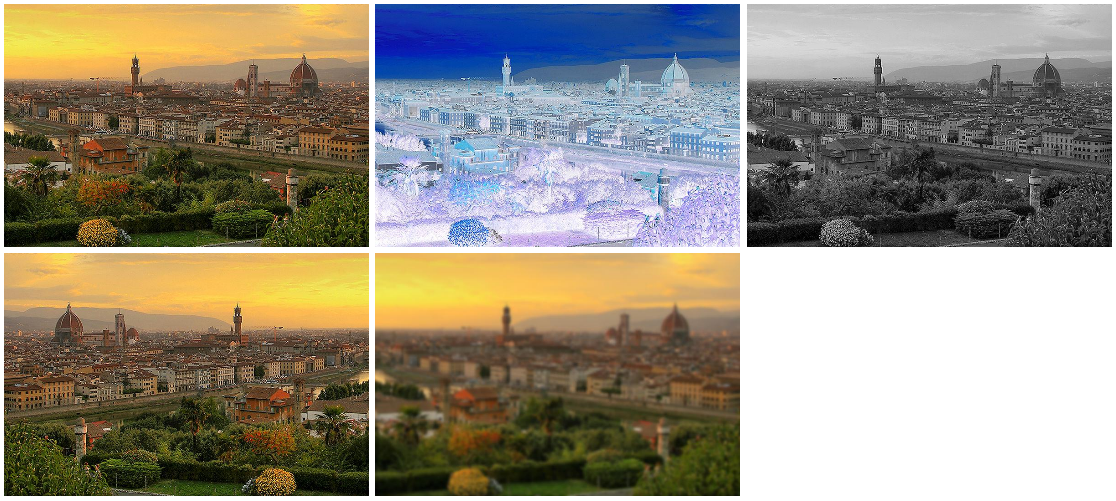
Homework 5: Parallel Image Processing



Original image of Sunset over Florence (the top-left image above) by http://www.flickr.com/people/sherseydc/ (http://flickr.com/photos/sherseydc/2954982676/) [CC-BY-SA-1.0 (http://creativecommons.org/licenses/by-sa/1.0) or CC-BY-2.0 (http://creativecommons.org/licenses/by/2.0)], via Wikimedia Commons

Due Wednesday, May 31 at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw5.java` text file that I have provided.

Make sure the file can be successfully compiled with `javac`. There should be *no compilation errors or warnings*; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must list whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

Processing large images can quickly become computation-intensive. Fortunately, many image transformations are naturally parallelizable, since each pixel (or groups of a small number of pixels) can often be processed independently. In this homework you will use Java streams and the Java fork-join framework for this purpose. You need to use Java version 8, which is the latest version. It is installed on the SEAS machines in `/usr/local/cs/bin`, and you can download it for your platform here.

As usual, good style matters. **Here are the rules:**

- **Do not modify the names of any types, classes, instance variables, and methods that our code defines in `hw5.java`. We are relying on them for testing purposes; your code will get no credit in cases where the test script can't find what it's looking for.**
- *Never* use type-unsafe features of Java, like casts and the `instanceof` expression. Similarly, never build your own version of `instanceof`, such as a method that returns `true` if an object has a particular class. If you ever need to figure out the class of some object, then your design is not as object-oriented as it should be.
- You may use any number of helper classes and methods that you require.

A few other tips:

- You will find the Java API Documentation useful. There you can find information on the various classes and interfaces in Java's stream and fork-join libraries. You may also find the `Math` class to provide useful functionality.
- Write comments where useful to tell the reader what's going on.
- Test your functions carefully: **eye-balling the results on actual images is not sufficient**. I encourage you to write your own test cases: craft some small image files (in the form of instances of the class `PPMImage` -- see below), and check that each transformation produces the expected output `PPMImage` objects.

We will be manipulating images in `ppm` format, which is a simple representation. Some image viewing applications can directly display `ppm` files, including the Aquamacs text editor on Mac OS X. However, many image-viewing applications require that the image first be converted to a JPEG. This can be done with the `pnmtojpeg` program at the command line as follows: `pnmtojpeg input-file.ppm > output-file.jpg`. This program comes with many Linux distributions as well as with Cygwin for Windows; it is also installed on the SEAS Unix machines. If you don't already have it, you can download `pnmtojpeg` as part of the NetPBM library. (The library also includes the program `jpegtopnm` for converting JPEGs into the `ppm` format; you can use this program to convert your favorite images so they can be manipulated by your code.)

Now on to the assignment! The file `hw5.java` includes three classes that are relevant for all problems (and one class that will be discussed later):

- `ImplementMe` is an exception that is used to mark places in the code that are currently unimplemented and need something from you.

- `RGB` objects each represent an RGB (red, green, blue) triple, which constitutes the color values for a single pixel. We're using such objects essentially just as tuples (which Java lacks), so the fields are public for easy access.

- `PPMImage` represents a PPM image file, whose format is described here. I've provided the functionality for parsing and unparsing PPM image files (one of the constructors performs parsing, and `toFile` performs unparsing), along with a sample image file in PPM format (`florence.ppm`) . Each `PPMImage` object includes integers that specify the image's width and height (in terms of the number of pixels) as well as the maximum value of any color (typically 256, but not always). Finally, the pixels themselves are represented by an array of `RGB` objects, which contains all `width * height` pixels contiguously, going row-by-row through the image from the top down, and from left to right in each row.

Your job is to implement the methods `negate`, `greyscale`, `mirrorImage`, `mirrorImage2`, and `gaussianBlur` in the `PPMImage` class. Here are the implementation requirements for your five methods:

1. All five methods should be *side-effect-free*, returning a new `PPMImage` object and leaving the old one unchanged.

2. Some methods will use Java streams, while others will use the Java fork-join framework, so **pay close attention to the requirements below**.

3. Implement the `negate` method, which produces the color negative of a given image (see the second Florence image above). To create the color negative of an image, simply replace each color value `v` within each pixel with `max-v`, where `max` is the image's maximum color value.

   The `negate` method should achieve parallelism using a parallel stream. This is quite natural because each pixel can be visited once in order to perform the negation. The method `Arrays.stream` converts a given argument array into a sequential `Stream`, and then you can invoke various stream operations to make the stream parallel, perform the computation you desire, and convert the result back to an array. The parallel execution is over the array of pixels, but your `negate` method should ultimately produce a new `PPMImage`.

4. Implement the method `greyscale`, which produces a greyscale version of a given image (see the third Florence image above). To do this, you intuitively want to average each pixel's R, G, and B values to produce a new value that all three should be replaced with. However, in practice the following formula tends to produce more pleasing pictures, so you should use it instead to produce the new color value: `.299 * R + .587 * G + .114 * B`
   You should round the resulting value to the nearest integer (see the `Math.round` method). As in the previous problem, **achieve parallelism using a parallel stream**.

5. Implement the `mirrorImage` method, which produces the mirror image of a given image (see the fourth Florence image above). This method is not as natural a candidate for streaming, because it needs to reorder pixels. Nonetheless, the transformation is naturally parallelizable since each pixel can be transformed in isolation. **Use Java's fork-join framework in order to exploit parallelism for this method, rather than using streams.** You should experiment with different partitioning schemes and sequential cutoff values. You may not see big speedups, though, due to the limited computation being done for each pixel.

6. It turns out that streams are expressive enough to handle `mirrorImage`. Illustrate this by implementing `mirrorImage2`, which should have identical behavior to `mirrorImage` but **use only streaming operations, with no explicit parallelism**.
   Hint: One way to do this is to use `IntStream.range` to generate a stream of integers, one per pixel, and then use other stream operations to process these integers in parallel.

7. Finally, implement the method `gaussianBlur`, which blurs an image using a Gaussian filter (see the fifth Florence image above). The blurring replaces each pixel's value with a kind of average of the pixel values surrounding it in the image. How

this is done is determined by the Gaussian filter, which is produced by the `Gaussian.gaussianFilter` method that I've provided.

A Gaussian filter is a square matrix, represented as a 2D array of doubles, of size `2*radius+1` by `2*radius+1` where `radius` is a parameter to the method. The other parameter, `sigma`, is a double that determines the values in the filter. You can experiment with different values for each parameter.

Given such a filter, you create the new value for a pixel's RGB triple by conceptually placing that pixel at the center of the filter. Then the new value for the pixel's R value, for example, is produced by summing the contribution from each R value covered by the filter, with each such value scaled by the filter. For example, consider this filter (resulting from the call `gaussianFilter(1, 2.0)`):

```
[[0.1018680644198163, 0.11543163961422663, 0.1018680644198163],
 [0.11543163961422663, 0.1308011838638283, 0.11543163961422663],
 [0.1018680644198163, 0.11543163961422663, 0.1018680644198163]]
```

Then the blurred R value at row `i` and column `j`, which we'll denote `R[i][j]` is computed as follows, rounded to the nearest integer:

```
0.1018680644198163 * R[i-1][j-1] + 0.11543163961422663 * R[i-1][j] + 0.1018680644198163 * R[i-1][j+1] +
0.11543163961422663 * R[i][j-1] + 0.1308011838638283 * R[i][j] + 0.11543163961422663 * R[i][j+1] +
0.1018680644198163 * R[i+1][j-1] + 0.11543163961422663 * R[i+1][j] + 0.1018680644198163 * R[i+1][j+1]
```

You should use a *clamping* semantics to handle situations when this calculation would take you out of bounds of the image. Specifically, whenever this process requires accessing a pixel whose row (column) is out of bounds, that row (column) should be substituted with the closest row (column) that is in bounds. For example, if the pixel at row -1 and column 1 is required, you'd instead use the pixel at row 0 and column 1, and if the pixel at row -1 and column -2 is required, you'd instead use the pixel at row 0 and column 0. Similarly, if the pixel at row 100 is required but the image has height 96, then you'd instead use the row value 95.

Though it can be implemented with Java streams, similar to what you did in the previous problem for the mirror image, Gaussian blur is not a natural candidate for that approach, since the values for one pixel depend on the values of its neighbors. Therefore, instead **implement this method using Java's fork-join framework**. You should experiment with different ways of partitioning the problem and different sequential thresholds. The Gaussian blur performs non-trivial computation per pixel (especially for large radius values, e.g. 60), so **you should be able to achieve significant speedups versus a sequential implementation when running on multicore hardware**.