Homework #6: Freakin' Fast Fractal Factory

Due Wednesday, May 5th at 11:59 p.m.

In this assignment, you will parallelize a fully-featured Mandelbrot set viewer that we provide to you. Your goals are:

- Practice parallel and concurrent programming in Java.
- Demonstrate software development best practices, including (but not limited to) use of a version control system, build automation, continuous integration, documentation, static analysis, and testing.
- Practice benchmarking and performance tuning skills.
- Have fun exploring the limitless complexity and beauty of the Mandelbrot set, while marveling at the simplicity of the underlying mathematics.

This assignment is due Wednesday, May 5th at 11:59 p.m..

The Mandelbrot set

The Mandelbrot set is the set of complex numbers for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, $f_c(f_c(f_c(0)))\ldots$ is bounded in absolute value. Don't worry if that doesn't make much sense. You don't need to know much about the Mandelbrot set for this assignment; you don't even need to understand complex numbers. [1]

For some complex numbers c that are *not* in the Mandelbrot set, f_c diverges quickly; for others it diverges more slowly. A complex number's *escape count* characterizes how fast it diverges. More formally, the escape count for c is the number of iterations it takes for f_c to diverge beyond a fixed *bailout radius*, which can be any real number ≥ 2. Because you can't iterate forever, real programs presume a point c is in the Mandelbrot set if f_c hasn't exceeded the bailout radius after some maximum number of iterations.

One can make beautiful images from the Mandelbrot set by using the escape count of f_c to determine the color of each pixel, where c is the point in the complex plane corresponding to the pixel. (The pixel's x value is the real part of c, and the y value is the imaginary part.)

[1]If you're new to fractals, though, we encourage you to watch this video: `https://www.youtube.com/watch?v=FFftmWSzgmk`.

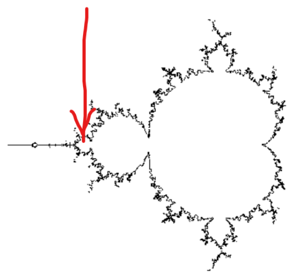
Rendering Mandelbrot images

At `https://bit.ly/32SPotw`, we have provided a Swing GUI application that allows you to render Mandelbrot images, computing the escape count and determining the color for each pixel using standard algorithms. Our GUI supports the following operations:

- Mouse click: Centers the image at the clicked position.
- Arrow keys: Scrolls the image up, down, left, or right.
- Plus/minus keys: Zooms in and out.
- '.' key: Restores the image to its initial state.
- 'c' key: Cycles through a set of color schemes.
- 't' key: Toggles the display of the time that it took to render the previous image.

First, assemble a project from the source files we've provided to you. Your project must use all the relevant tools we've introduced throughout the term, including SpotBugs. Your project should build on Travis-CI and run `MandelbrotMain` locally with `gradle run`.

Our code renders the Mandelbrot images sequentially, using the `SEQUENTIAL_RENDERER` in the `MandelbrotMain` class. Use the 't' key to see how long it takes to render each image. To see the program at its slowest, render a region entirely contained within the small circular bulb on the left in the initial image:



Write at least two parallel renderers using different Java concurrency tools. At least one renderer should use executors from `java.util.concurrent`, and at least one renderer should use parallel streams. You might write additional renderers that use the same underlying concurrency primitives but differ in other nontrivial ways, such as the granularity of the concurrency. Try to make the new renderers as fast as possible without changing the pixel rendering in any way. Running the viewer with your new renderers must produce exactly the same images as our sequential renderer. Please restrict your optimizations to parallelization; do not make algorithmic changes to the computation.

Benchmark your parallel renderers using the built-in timing facility (the 't' key). Can you explain any observed differences in their performance? Can you make them even faster? Should you consider a different approach? You should aim for performance that is superlinear in the number of physical processors on your machine. We observed a 20x speedup on a 12-core machine, and a 3.1x speedup on a dual core machine. Note that the rendering speed will vary greatly depending on the region you are rendering! Be sure to do apples-to-apples comparisons between the various renderers.

You should plan to complete your solution well before the assignment deadline so that you have time to explore how you might improve the performance of your parallel implementation, and to enjoy your spiffy, fast Mandelbrot set viewer. You should definitely switch back and forth between the different color schemes (which you do by pressing the 'c' key), as they provide very different images. We encourage (but do **not** require) you to add functionality to the program, such as adding new color schemes, storing the image to an output file (using `javax.imageio`), providing read and write access to the coordinates of the rendered region so the user can return to it in future runs of the program, etc.

Finally, describe your strategies for parallelization in a `discussion.md` file. Explicitly describe what constitutes a unit of work (to be done in parallel with other work) and describe how you achieve safety in the face of concurrency. You should also discuss your benchmarking experiments, how you used them to help improve the performance of your implementation, and any changes you made to your implementation and/or parallelization strategy as a result of those experiments.

Evaluating your work

When you are done, your solution should include a discussion and at least two parallel renderers, and it should be easy for us to recompile and run your program with your different renderers. Please remember that we are evaluating your software development process in this assignment; include all artifacts needed for us to understand your work, and frequently commit your work so we can see how your solution evolved over time.

This homework is worth 100 points. Your work will be evaluated approximately as follows:

- Implementation of an executor-based parallel renderer: 30 points
- Implementation of a streams-based parallel renderer: 30 points
- Demonstrating software development best practices: 30 points
- Performance benchmarking and discussion: 10 points