Extra Credit Assignment

This assignment requires a machine with at least 4 processing cores. Use the CADE lab1 or lab3 machines.

This assignment is extra credit. It will replace your lowest score from assignments 1-12 (not the final project).

The purpose of this assignment is to practice your knowledge of multithreaded programming. The assignment may be done either alone, or in pairs if you prefer.

Before starting, review the lecture 23 and 24 slides on multithreading.

The Problem

We have been brought on to an on-going project to design a fractal visualizing tool. The project is already well-underway, and currently draws Mandelbrot fractals to the screen, albeit slowly. See the wikipedia article for information on fractals and the Mandelbrot set.

Start by downloading the provided files. Add the files to a package named `extracredit`.

We recognize the parallel nature of fractal drawing, and you are tasked with making the tool more interactive, and faster, using multithreading.

Familiarize yourself with each of the provided files until you feel like you have a rough idea of what each does. Run the `main` method in `MandelbrotTester.java`, which opens a window and renders the Mandelbrot set. Notice that the image does not display until it has been fully computed.

Play around with the controls:

- Clicking the image anywhere will redraw the image centered on that point

- Pressing the '+' and '-' keys will zoom the image in and out

- Pressing the 'd' key will redraw the current image

NAVIGATION

Home
- My home
  - Site pages
  - My profile
  - My courses
    - Computer Science
      - Previous Semester
        - CS 1410-1-S13
        - CS_2100_S_13
        - CS2420-S13
          - Participants
          - General
          - Getting started; Java review
          - Generic programming; Object Oriented Programming
          - Algorithm analysis; Data Structures
          - Basic Sorting Algorithms
          - Recursive Sorting Algorithms
          - Linked Lists
          - Stacks and Queues

Part 1

Modify the program so that the image is updated simultaneously while it is being computed. We don't want to stare at a blank screen while we wait for it to compute the pixels.

To do this we will create a separate painter thread that runs concurrently with the main thread that computes the image. This painter thread will continuously redraw the pixels as they are computed. We don't want this thread to take up too much computing power, so it should only update at a reasonable frame rate (30 times per second).

- Fill in the `run` method in `Painter.java` - this class implements `Runnable`, which allows it to run in its own concurrent `Thread` of execution. See the comments in `Painter.java` for instructions.
- Add code to the `runViewer` method in `MandelbrotViewer.java` that creates and starts a `Thread` with a `Painter` object. This `Thread` should run forever (until closing the window automatically stops it). We should now be able to see the pixels fill in as they are computed. See the lecture 23 and 24 slides for information on the `Runnable` interface, and creating and running a `Thread`.

Part 2 - Parallel Rendering

Before continuing, make sure you are working on a machine with at least 4 processor cores.

As we can tell, drawing a Mandelbrot takes a fair amount of CPU power (notice that the total time taken to render a frame is displayed after it is rendered). Since modern computers have multiple processing cores, we want to utilize them to speed up the process and draw fractals faster. Luckily, each pixel is independant, and can be computed without any knowledge of any other pixel. This leads to obvious opportunities for parallel computation.

- Fill in the `run` method of the `MandelbrotWorker` class, which has been started for you. Follow the instructions in the comments (**only fill in the code described for part 2).** Notice that `MandelbrotWorker` implements `Runnable`, meaning its `run` method can execute concurrently in its own `Thread` (just like `Painter`). `MandelbrotWorker`s will have a simple task: render a portion of the screen. Since they can run in their own thread, we can create several of them and render multiple portions of the image simultaneously.

- Fill in the code for `drawStaticParallel` in `MandelbrotViewer.java` based on

the instructions provided in comments. This method should create 4 threads that run a `MandelbrotWorker` that each render a pre-determined portion of the image simultaneously (use the `MandelbrotWorker` constructor that takes a `WorkAssignment`).

    - The method `badDistribution` in `MandelbrotViewer.java` returns 4 `WorkAssignment`s. A `WorkAssignment` is a simple class that represents a rectangular portion of the image that needs to be rendered.

- In `MandelbrotTester.java`, comment out the line that runs the `basicViewer` and uncomment the line that runs `staticParallelViewer`.

Part 3 - Dynamic Work Distribution

The code we wrote in part 2 is fixed to use exactly 4 rendering threads. We want to be able to use fewer threads if we don't want to take over the whole CPU, and more threads if the power is available.

- Fill in the code for `drawDynamicParallel` in `MandelbrotViewer`, following the instructions given in comments. Notice that by default, it uses `badDistribution`, to create 4 work assignments. This time, however, the assignments are placed in to a `WorkQueue`. Create `numThreads` threads, each with its own `MandelbrotWorker`, but this time, create the workers with the constructor that takes a `WorkQueue`, passing them each the same queue.

- Modify your code in `MandelbrotWorker.run` so that it can use a `WorkQueue` (follow the instructions in comments listed for part 3). A `MandelbrotWorker` should retrieve an assignment from the queue and render it, then repeat, until the queue is empty.

- Modify the code in `WorkQueue.java` so that it is free of race conditions, since we now have multiple threads potentially modifying the queue simultaneously.

- In `MandelbrotTester.java`, comment out the line that runs the `staticParallelViewer` and uncomment the line that runs `dynamicParallelViewer`. Notice it runs the viewer with 2 threads. Experiment running the viewer with 1-4 threads (using more than 4 will have no effect since our work queue only contains 4 assignments).

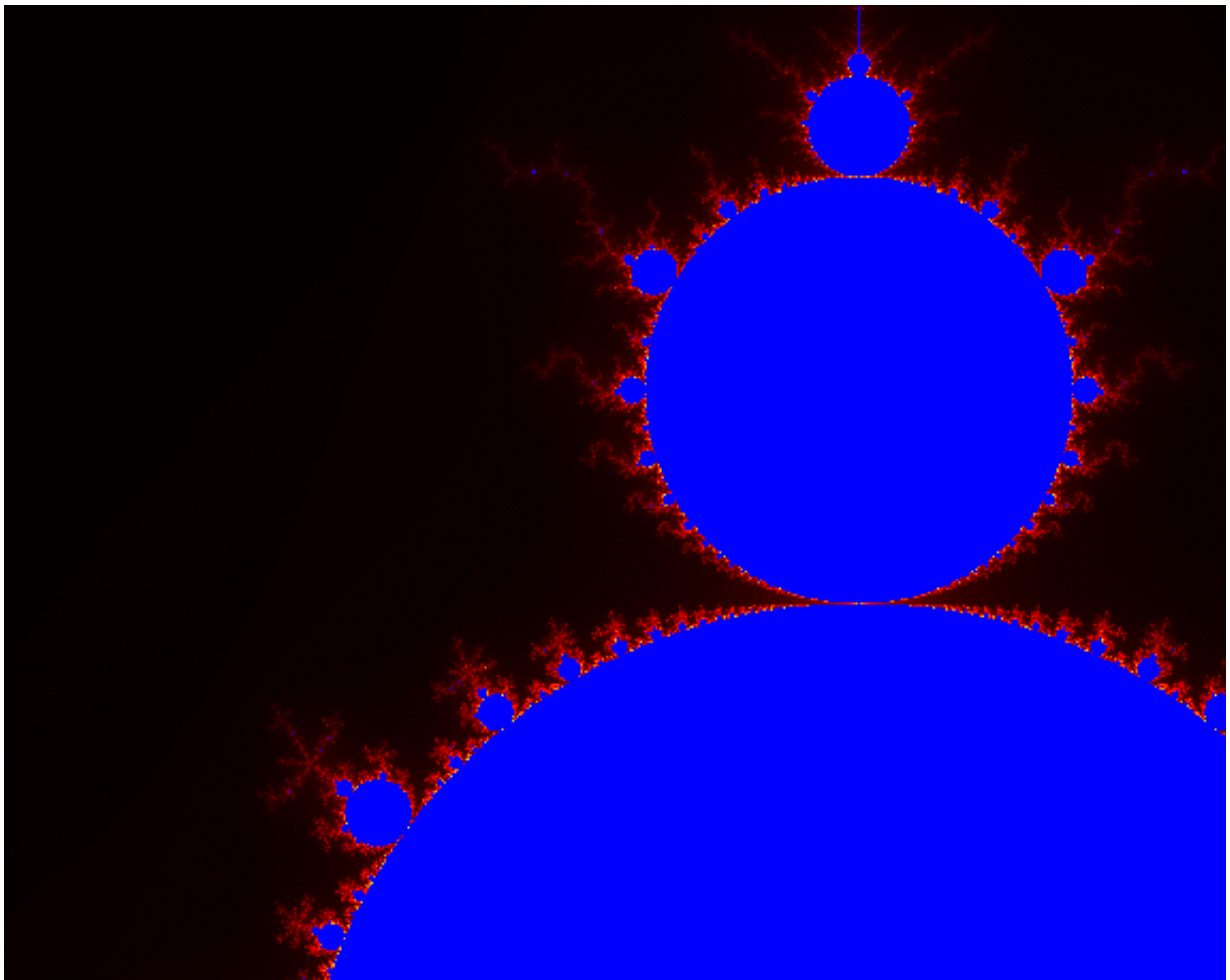
Part 4 - Workload Balancing

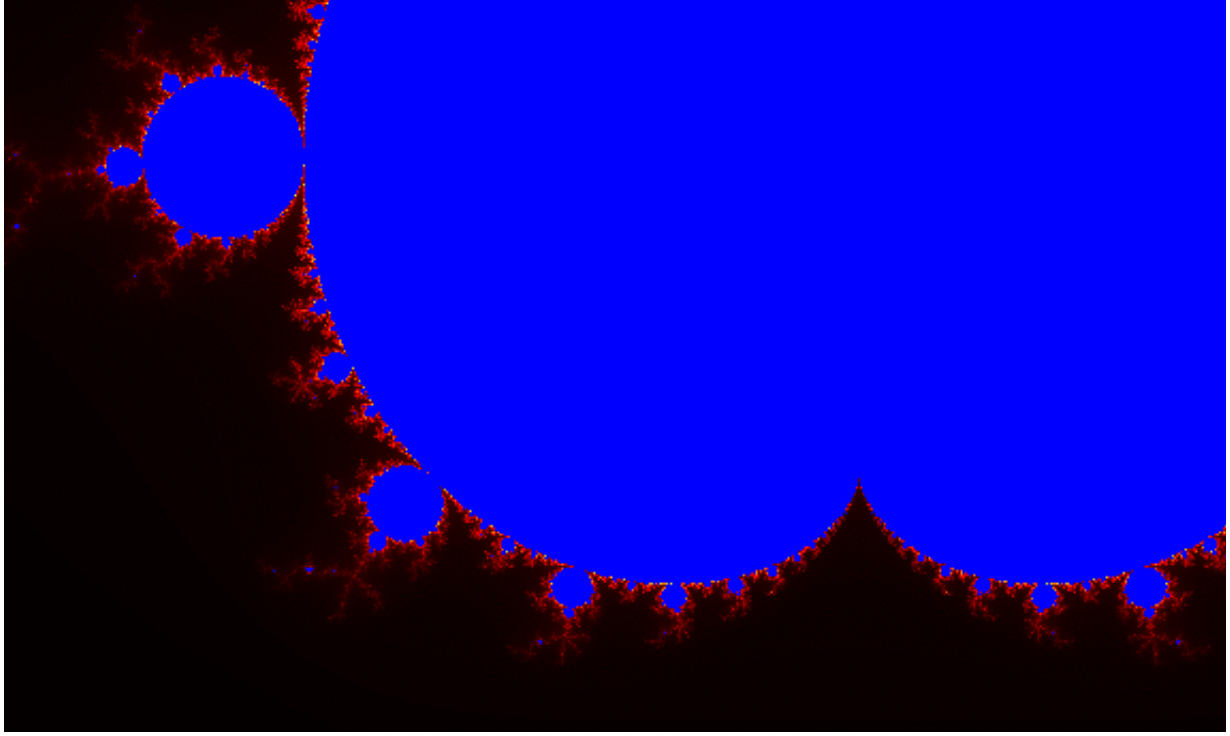
In parts 2 and 3, we used multiple cores to render the fractal instead of just one, so it is

reasonable to expect a 4x speedup if using 4 cores. What is the difference in time required to render the default image with 1 core, vs. with 4 cores? Achieving 4x speedup does not seem to be the case...

The assignments given to each thread do not require the same amount of work - some of them are a lot more expensive, causing one or two threads to run longer, and the other threads to finish early and terminate, leaving the CPU core underutilized. We must try to minimize the total difference in runtime required to process each assignment. The cause of this problem is that the Mandelbrot set requires a lot more work to determine the color of certain pixels than others.

The image below shows the most expensive regions to render in blue (blue pixels require the maximum possible amount of work). The easiest to render are black, and the red/yellow pixels are somewhere in the middle. Since the user can change the center point and zoom, we have no way of knowing ahead of time which pixels will take the longest to render.





- In the file `MandelbrotViewer.java`, fill in the code for `betterDistribution`. This method should return an `ArrayList` of `WorkAssignment`s that break the rendering task in to more even pieces than those returned by `badDistribution`. It must break the task in to at least 8 portions (return 8 different assignments).
- Fill in the code for `goodDistribution`. This method should return work assignments that represent portions of work as close to equal as possible, for all possible viewing positions of the fractal. A general strategy is to make the assignments smaller (and thus more of them), however, they should not be so small that the thread spends a large amount of time simply requesting tasks from the queue. Recall that removing an item from the queue is a critical section, and only one thread can do this at a time.

When preliminary coding is complete and your program compiles without error or warning, test the program thoroughly and systematically.

Your code should be well-commented (Javadoc comments are recommended) and formatted such that it is clear and easy to read. Be sure to put the names of both programming partners in the header comment of each file.

Zip your source code files (`.java` only) and **upload the zip file here by 11:59pm on Monday, April 29**. Please submit just one solution per pair (i.e., one partner should upload the zip file, the other should not upload anything).

1. Analysis Document (must be written and submitted by each programming partner) ***due Monday, April 29 at 11:59pm***

Submission status

Submission status	Submitted for grading
Grading status	Graded
Due date	Monday, 29 April 2013, 11:55 PM
Time remaining	Assignment is overdue by: 76 days 14 hours
Last modified	Monday, 29 April 2013, 9:39 PM
File submissions	Analysis.pdf

Feedback

Grade	95.00 / 100.00
Graded on	Friday, 3 May 2013, 6:24 PM
Graded by	Daniel Kopta
Feedback comments	Part 1 - 20/20

	Part 2 - 20/20 Part 3 - 15/20 Each of your threads are creating their own separate semaphore, which is not ...