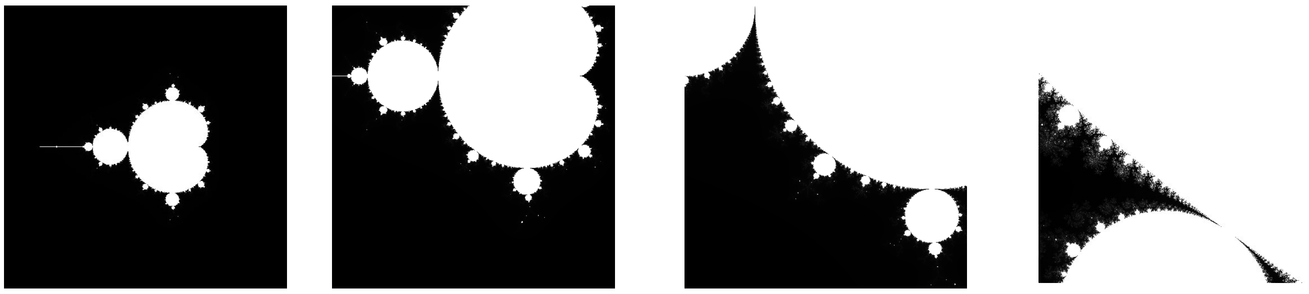
# Programming Assignment 3: Threads
# CSE 3320.001
# Due: March 26, 2018 5:30PM

## Description

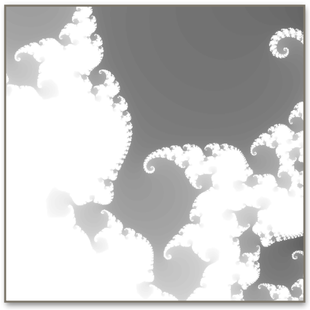
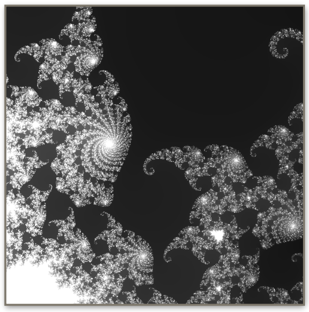
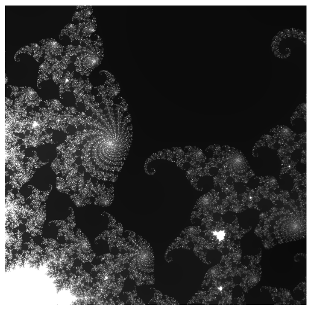
In order to study parallelism, we must have a problem that will take a significant amount of computation. For fun, we will generate images in the Mandelbrot set, which is a well known fractal structure. The set is interesting both mathematically and aesthetically because it has an infinitely recursive structure. You can zoom into any part and find swirls, spirals, snowflakes, and other fun structures, as long as you are willing to do enough computation.



```
./mandel -x -0.5 -y 0 -s 2 -o mandel1.bmp
./mandel -x -0.5 -y -0.5 -s 1 -o mandel2.bmp
./mandel -x -0.5 -y -0.5 -s 0.005 -o mandel3.bmp
./mandel -x -0.5 -y -0.5 -s 0.05 -o mandel3.bmp
./mandel -x -0.5 -y -0.5 -s 0.05 -o mandel4.bmp
```

You will be provided a program that generates images of the Mandelbrot set and saves them as BMP files. Run `make` to build the code. If you run the program with no arguments, it generates a default image and writes it to mandel.bmp. You can see all of the command line options with mandel -h, and use them to override the defaults. This program uses the `escape time algorithm`. For each pixel in the image, it starts with the x and y position, and then computes a recurrence relation until it exceeds a fixed value or runs for max iterations.

Then, the pixel is assigned a color according to the number of iterations completed. An easy color scheme is to assign a gray value proportional to the number of iterations. The max value controls the amount of work done by the algorithm. If we increase max, then we can see much more detail in the set, but it may take much longer to compute. Generally speaking, you need to turn the max value higher as you zoom in. For example, here is the same area in the set computed with four different values of max:

```
./mandel -x 0.286932 -y 0.014287 -s .0005 -m 50 -o mandel1.bmp
./mandel -x 0.286932 -y 0.014287 -s .0005 -m 100 -o mandel2.bmp
./mandel -x 0.286932 -y 0.014287 -s .0005 -m 500 -o mandel3.bmp
./mandel -x 0.286932 -y 0.014287 -s .0005 -m 1000 -o mandel4.bmp
./mandel -x 0.286932 -y 0.014287 -s .0005 -m 2000 -o mandel5.bmp
```

## Parallel Programming

What does this all have to do with operating systems? It can take a long time to compute a Mandelbrot image. The larger the image, the closer it is to the origin, and the higher the max value, the longer it will take. Suppose that you want to create a movie of high resolution Mandelbrot images, and it is going to take a long time. Your job is to speed up the process by using multiple CPUs. You will do this in two different ways: using multiple processes and using multiple threads.

## Find an image

Explore the Mandelbrot space a little bit, and find an interesting area. The more you zoom in, the more interesting it gets, so try to get -s down to 0.0001 or smaller. Play around with -m to get the right amount of detail. Find a configuration that takes about 5 seconds to generate on omega. If you find an image that you like, but it only takes a second or two to create, then increase the size of the image using -W and -H, which will definitely make it run longer.

## Part 1: Multiple Processes

Now, write a new program `mandelseries` that runs mandel 50 times, using what you learned in your shell program. Keep the -x and -y values the same as in your chosen image above, but allow -s to vary from an initial value of 2 all the way down to your target value. The end result should be 50 images named mandel1.bmp, mandel2.bmp, etc.

Generating all those frames will take some time. We can speed up the process significantly by using multiple processes simultaneously. To do this, make `mandelseries` accept a single argument: the number of processes to run simultaneously. So, `mandelseries 3` should start three mandel processes at once, then wait for one to complete. As soon as one completes, start the next, and keep going until all the work is complete. `mandelseries` should work correctly for any arbitrary number of processes given on the command line.

## Part 2: Multiple Threads

Instead of running multiple programs at once, we can take a different approach of making each individual process faster by using multiple threads.
Modify `mandel.c` to use an arbitrary number of threads to compute the image. Each thread should compute a completely separate band of the image. For example, if you specify three threads and the image is 500 pixels high, then thread 0 should work on lines 0-165, thread 1 should work on lines 166-331, and thread 2 should work on lines 332-499. Add a new command line argument -n to allow the user to specify the number of threads. If -n is not given, assume a default of one thread. Your modified version of `mandel` should work correctly for any arbitrary number of threads and image configuration. Verify that your modified `mandelseries` produces the same output as the original.

## Part 3: Evaluation Report

Write a short lab report that evaluates your two parallel versions:

- In your own words, briefly explain the purpose of the experiments and the experimental setup. Be sure to clearly state what your command line arguments were, so that your results can be reproduced to review your report.
- Measure and graph the execution time of `mandelseries` for each of 1, 2, 3, 4, 5, and 10 processes running simultaneously. Because each of these will be fairly long running, it will be sufficient to measure each attempt only once.
- Explain the shape of the curve. What is the optimal number of processes? Why? Is it possible to have too many processes? Why?
- For the following two configurations, measure and graph the execution time of multithreaded mandel using 1, 2, 3, 4, 5, 10, and 50 threads. The execution time of these experiments may be upset by other things going on in the machine. So, repeat each measurement five times, and use the fastest time achieved.
    - A: mandel -x -.5 -y .5 -s 1 -m 2000
    - B: mandel -x 0.2869325 -y 0.0142905 -s .000001 -W 1024 -H 1024 -m 1000
- Explain the shape of the two curves. What is the optimal number of threads? Why do curves A and B have a different shape?

## Code

The code for this assignment may be found on the course GitHub page at: https://github.com/CSE3320/Fractal-Assignment

## Grading

This assignment must be coded in C or C++. Any other language will result in 0 points. You programs will be compiled and graded on `omega.uta.edu`. Please make sure they compile and run on `omega` before submitting them. Code that does not compile on omega will result in a 0.

Good coding style, including clear formatting, sensible variable names, and useful comments. (10%)
A correct implementation of multi-process Mandelbrot. (30%)
A correct implementation of multi-threaded Mandelbrot. (30%)

A lab report which is clearly written using correct English, contains an appropriate description of your experiments, contains correct results that are clearly presented, and draws appropriate conclusions. (30%)

Your programs are to be turned in via blackboard. Submission time is determined by the blackboard system time. You may submit your programs as often as you wish. Only your last submission will be graded.

## Academic Integrity

This assignment must be 100% your own work. No code may be copied from friends, previous students, books, web pages, etc. All code submitted is automatically checked against a database of previous semester's graded assignments, current student's code and common web sources. By submitting your code on blackboard you are attesting that you have neither given nor received unauthorized assistance on this work. **Code that is copied from an external source, excluding the course github or blackboard, will result in a 0 for the assignment and referral to the Office of Student Conduct.**