Parallel Mandelbrot

This project must be implemented in C (and not C++ or anything else)

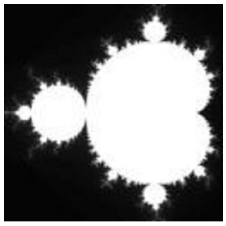
The goals of this project are:

- To learn how to exploit coarse-grained parallelism with processes.
- To learn how to exploit fine-grained parallelism with threads.
- To gain experience evaluating parallel performance with both processes and threads.
- To have a little fun with the Mandlebrot set.

The Mandelbrot Set

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. You can know in detail about this set in http://en.wikipedia.org/wiki/Mandelbrot_set. 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. For example, here are three images starting from the entire set and zooming in:

mandel -x -0.5 -y 0 -s 2



mandel -x -0.5 -y -0.5 -s 1

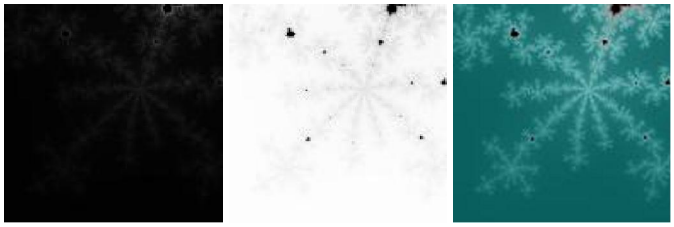
mandel -x -0.5 -y -0.5 -s 0.25

You are given source code for a simple program that generates images of the Mandelbrot set and saves them as BMP files. Just download all of the files and run `make` to build the code. If you run the program with no arguments, then it generates a default image and writes it to `mandel.bmp`. Use `display mandel.bmp` to see the output. You can see all of the command line options with `mandel -h`, and use them to override the defaults. (Each of the images on this page is labelled with the command that produces it.) 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.

```
int iterations_at_point( double x, double y, int max )
{
        double x0 = x;
        double y0 = y;
        int iter = 0;

        while( (x*x + y*y <= 4) && iter < max ) {

                double xt = x*x - y*y + x0;
                double yt = 2*x*y + y0;

                x = xt;
                y = yt;

                iter++;
        }

        return iter;
}
```

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, but others are possible. Here are a few color variations of the same configuration:

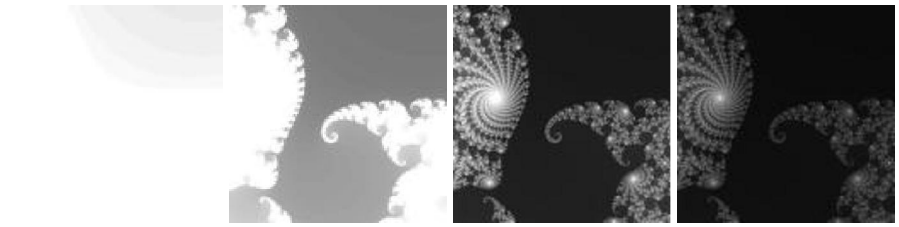
mandel -x -.38 -y -.665 -s .05 -m 1000



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 -m 100 -m 500 -m 1000



Parallel Programming

Now, what does this all have to do with operating systems? It's very simple: 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.

Step One: Find a Good 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. 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.

Step Two: 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. Double check that your modified `mandel` produces the same output as the original.

Step Three: Multiple Processes

Now, write a new program `mandelmovie` that runs `mandel` 50 times, using what you learned in the previous project. 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` and so forth. For fun, you can stitch all of the images into a movie with a command like this: `ffmpeg -i mandel%d.bmp mandel.mpg`, and then play with with `ffplay mandel.mpg`. You are given an example of a large movie with 1000 frames.

Obviously, generating all those frames will take some time. We can speed up the process significantly by using multiple

processes simultaneously. To do this, make `mandelmovie` accept an optional argument: the number of processes to run simultaneously. So, `mandelmovie -p 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. `mandelmovie` should work correctly for any arbitrary number of processes given on the command line.

Step Four: Execution Time

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

- Measure the execution time of `mandelmovie` for each of 1, 2, 3, 4, 6, and 12 processes running simultaneously. Because each of these will be fairly long running, it will be sufficient to measure each configuration only once. Record the execution times in a file.
- For the following two configurations, measure and graph the execution time of multithreaded `mandel` using 1, 2, 4, 6, 12, 18, and 24 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
-

Defensive programming

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinately, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

Though this project does not have much scope for defensive programming, you need to handle the situations that may arise.

For example, you need to show proper error messages in case of invalid no. of arguments (less or more than expected).

- `mandelmovie -p`
- `mandelmovie -p 2 3`
- `mandel -n 2 2`
- `mandel -n`

Hints

`mandelmovie` is a simple loop, once you think it through. Use an integer to keep track of the number of running processes. If the number is too low, fork a new process; if it is too high, wait for something to exit. Keep looping until done.

If your movie starts off zooming very slowly, and then accelerating to the end, try this: To get a nice smooth zoom from s=a to s=b, start with `s=a` and then multiply it each time by `exp(log(b/a)/51)`.

`mandel.c` uses the standard `getopt` routine to process command line arguments. To add the `-n` argument, you will need to add `n:` to the third argument of `getopt`, add an additional case statement underneath, and update the help text.

Where `main` previously made one call to `compute_image`, you will need to modify it to create N threads with `pthread_create`, assign each to create one slice of the image, and then wait for each to complete with `pthread_join`.

`pthread_create` requires that each thread begin executing a function that only takes one pointer argument. Unfortunately, `compute_image` takes a whole bunch of arguments. What you will need to do is modify `create_image` from this:

```
void compute_image( struct bitmap *b, double xmin, double xmax, double ymin, double ymax, double itermax );
```

to this:

```
void * compute_image( struct thread_args *args )
```

where `thread_args` contains everything that you want to pass to `compute_image`. Now, for each thread, allocate a `thread_args` structure and pass it as the fourth argument to `pthread_create`, which will turn around and pass it to `compute_image`.

Pthreads requires `compute_image` to return a `void *`, but since it doesn't actually need to return any data, just `return 0;` at the end of the function.

Do's & Don'ts

- DO NOT COPY from friends, previous semesters' students, internet or any other source. Your submitted codes will be passed through copy checker and you will face severe grading penalty if caught. You can discuss with others but the code must be done by you and your groupmates only.
- Keep your code clean, appropriately commented and modularized.