CPE-2600 FALL 2024

Systems Programming

Lab 12 Multithreading

Acknowledgement: Many aspects of this assignment are modeled after this: Operating Systems Principles (fiu.edu). See additional acknowledgements in the template code.

Topical Concepts

Multithreading

Like multiprocessing, multithreading can deliver a number of benefits, again, with the most obvious being some speedup by carrying out processing in parallel.

In the previous lab, we employed multiprocessing by running separate, independent processes. This week we will "parallelize" within each processes by using separate threads each calculating a different portion of the Mandelbrot image. Threads are ideal for this addition as opposed to processes since threads typically share heap memory without additional overhead.

While we have learned various techniques of coordinating shared resources in lecture, it turns out we will not really need them for this project. Mandelbrot calculations are considered "embarrassingly parallel" [see: Embarrassingly parallel - Wikipedia]. This is because each pixel in the image can be calculated independently, requiring no information from any of its neighbors.

A summary of the activity will be to add an option to the mandel program which will tell it how many threads to split into to calculate a single image. You will then benchmark the program with various combinations of multithreading and multiprocessing. No Lab 11 features should be removed.

The bulk of your changes will relate to how the loops in **`compute_image()`** operate.

```c
void compute_image( struct bitmap *bm, double xmin, double xmax, double ymin, double ymax, int max )
{
    int i,j;

    int width = bitmap_width(bm);
    int height = bitmap_height(bm);

    // For every pixel in the image...

    for(j=0;j<height;j++) {

        for(i=0;i<width;i++) {

            // Determine the point in x,y space for that pixel.
            double x = xmin + i*(xmax-xmin)/width;
            double y = ymin + j*(ymax-ymin)/height;

            // Compute the iterations at that point.
            int iters = iterations_at_point(x,y,max);

            // Set the pixel in the bitmap.
            bitmap_set(bm,i,j,iters);
        }
    }
}
```

Note that loops counters i and j simply iterate to each pixel of the image and call **`iterations_at_point()`** to get the "color" at that pixel. We can easily parallelize this by splitting the image up into 2 or more distinct regions and have a separate thread iterate its own region.

The Exercise

Specific Instructions

1. You will work in the same repository as the last assignment. You shall create a new branch "lab12dev" and do all new work on that branch. In the meantime, the main branch of your repository should remain at the completed, deployable Lab 11 lab project.

2. Add a command line argument to set the number of threads to be used to calculate an image. The exact place that you make your changes will depend a bit on how you completed Lab 11. If you left mandel.c mostly unchanged you can simply add a new command line argument to the mandel program to give it the number of threads it should split into. If you modified the mandel program to become mandel movie, you again would simply add a new command line argument. Do not remove any features from Lab 11 – we will ultimately be doing both multiprocessing and multithreading.

3. Using "pthreads" spin off the requested number of threads to calculate a single image. A couple of notes:

    a. As with last week, each region may take more or less amount of time to calculation. Unlike last week, you do not have to explicitly handle that and ensure there are always the specified number of threads running (although you can if you want…). Rather, if 4 threads are requested, split the image up into 4 regions and spin up 4 threads. The main thread will need to join those 4 threads before completing the rest of the process.
    b. You will need a way to pass each thread's region to it. You can pass a void* to the thread method. Keep in mind that that pointer could point to memory that contains that thread's assignment or, a pointer is 8 bytes on our system and it does not have to be interpreted as a memory address.

c. You should support a minimum of 1 thread up to a maximum of 20.

4. Finally, re-run your runtime data collection from last week by also varying the number of threads used in addition to number of processes. Create a table with # threads on one axis and # processes on the other. In each cell place the runtime to generate 50 images with that combination. **Add** to your exiting report in README.md:

    a. A brief overview of your implementation of multiple threads.
    b. The table described above.
    c. A brief discussion of your results. Answer the following questions:
        i. Which technique seemed to impact runtime more – multithreading or multiprocessing. Why do you think that is?
        ii. Was there a "sweet spot" where optimal (minimal) runtime was achieved?

Deliverable

- When you are ready to submit your assignment prepare your repository:
- Make sure your name, assignment name, and section number are all files in your submission - in comment block of source file(s) and/or at the top of your report file(s).
- Make sure you have completed all activities and added all new program source files to repository.
- Make sure your assignment code is commented thoroughly.
- Make sure all files are committed and pushed to the **main branch** of your repository.
- Tag your repo with the tag ```Lab12vFinal```

NOTE: Do not forget to 'add', 'commit', and 'push' all new files, branches, and changes to your repository before submitting.

To submit, copy the URL for your repository and submit the link to associated Canvas assignment.

The completed assignment is due as per Canvas date. You will be asked to demonstrate your program and/or play your movie.