CS 214: Introduction to computing systems
Winter Term, 2026

Lab 6

Using threads for parallelism

In this lab, you'll look at using pthreads to create multiple threads to speed up a computationally-intensive task. We'll be doing this on `euclid`.

Once you're logged on, begin the lab by creating a subdirectory for this lab and copying the given code from the `lab6` subdirectory of the course directory:

```
cp -r /home/courses/cs214/lab6 .
```

The `-r` stands for "recursive" since this copies the entire subdirectory. Then you can move your working directory into this new subdirectory:

```
cd lab6
```

(This is the only line you'll have to rerun to get back to the code after logging out; your working directory always starts in your home directory when you log in.)

Some of the files that you copied (`mandelbrot.c` and `bmp.h`) are a program to draw an image. Compile them with

```
gcc -Wall -std=gnu99 -o mandelbrot mandelbrot.c
```

and run the resulting executable:

```
./mandelbrot out.bmp
```

The `out.bmp` part is a command line argument, which tells the program which file you'd like to create as output. You can change the name as long as it ends in `.bmp`. This extension stands for "bitmap", which is a graphics format.

To view the output file, you'll need to bring it over to your computer. (This is one unfortunate aspect of working on a remote system.) I suggest using the program `sftp`, which stands for secure file transfer protocol. Open a connection with `sftp dbunde@euclid` (using your own username rather than mine...). Once you're authenticated, switch into the `lab6` directory with `cd lab6`. Then get the output file with `get out.bmp`. After this, you should be able to open the file by clicking on it; the file will be placed in whatever directory you're running `sftp` from.

The image that appears is a bitmap created by the program. It is a complicated black and white figure representing the *Mandelbrot set*. You can read more about this set on Wikipedia (or other online sources), but a short explanation is that this figure is based on repeating a simple mathematical operation and seeing if the result becomes unboundedly large. The specific operation depends on the pair of coordinates being considered. A point that is in the set (appearing black in the diagram) is one whose operation yields a bounded value (ideally forever, though we only look for 1,000 iterations of the operation). Points appearing white in the diagram are not in the set, meaning that the operation applied to their coordinates grew past a threshold within the first 1,000 iterations.

That's all well and good, but we need to look at the actual code. Open `mandelbrot.c` and examine the `main` function (you'll need to scroll down). It begins by writing the header of a bitmap file. Then it fills in a two-dimensional array called `pixels` that stores the desired color of each pixel. It concludes by writing the contents of this array into the file. For a single pixel, the `mandelbrot` function determines whether

that pixel should be drawn as in the set or not. The bulk of the program's running time is a series of calls to this function from `main`, right after the comment "`set pixels`". Find this comment and examine that paragraph of code. You're welcome to examine the `mandelbrot` function, but a detailed understanding of it is not necessary for the lab. Its most important feature is that it returns either 0 or 255 depending on whether the pixel at the given coordinates should be drawn as black or white. The pixels themselves are represented with three bytes each; each of these bytes should be a number 0–255 that gives the desired intensity of one color at a specific location; the colors represented are red, green, and blue. Other colors are displayed by showing combinations of these colors. To facilitate saving these triples of bytes to a file, the color values of each pixel are stored in a `struct` called `RGBTRIPLE` that is defined in `bmp.h`. The lines

```
pixels[i][j].rgbtBlue = color;
pixels[i][j].rgbtGreen = color;
pixels[i][j].rgbtRed = color;
```

are accessing one of these structures out of the 2D array and setting its fields.

As you may have noticed, this program takes a fair amount of time to run. Use the `time` program to measure how long it takes:

```
time ./mandelbrot out.bmp
```

(Note that when doing timing experiments like this, it matters what else is running on the system. The time may vary based on how many people try to run this program at once.) You'll get something like

```
real 0m1.397s
user 0m1.393s
sys 0m0.005s
```

The first entry is the running time (1.397 seconds in this case) as "wall clock" time, the amount of time that would be recorded by a clock on the wall. The user and sys times are times in "user mode" and "system mode". The system mode is time spent in the operating system (while the OS is executing system calls on the program's behalf). Everything else is user mode. The other difference is that user time and system time are recorded per thread: if you have 2 threads running in user mode, that time goes up twice as fast. (Alternately, the user mode amount is the sum of the amounts of time each thread spent in user mode.)

The focus of this lab will be in trying to reduce this delay. To do so, you'll be using the Pthreads library to create multiple threads. A sample program which uses pthreads is in `helloThread.c`. Take a look at this program. In `main`, two threads are created with `pthread_create`. This function takes a pointer to the data structure representing a thread (`pthread_t`) and the name of the function that the thread should run (`hello` in our case). `pthread_create` creates the thread and starts it on the given function. We use the `failUnless0` function to report any errors that occur.

Compile this sample program with

```
gcc -Wall -std=gnu99 -o helloThread helloThread.c -lpthread
```

(The `-lpthread` part tells the compiler (actually the linker) that you're going to use the pthread library.)

You can run the program with the following:

```
./helloThread
```

When you do, it almost certainly doesn't print what you expect; most of the time, the program finishes before both created threads have completed the `hello` function. This is because the main thread (the one running when the program starts which creates the others) doesn't explicitly wait for the threads it creates. To do this, you need to add the lines

```
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

between the calls to `pthread_create` and `printf`. Each of these calls causes the main thread to wait until the appropriate created thread finishes (which happens at the end of the `hello` function).

Before we can use pthreads for the Mandelbrot problem, we also need to learn how to pass arguments to the function each thread is running. Looking at the top of `hello`, we see that its argument has type `void*`. This means that the argument is a pointer, but without information about the type of data being pointed to. (That's done so that `pthread_create` can be used to call functions that want different types of data without changing its signature.) We simply pass a pointer to the desired argument as the last argument to `pthread_create`. For example, to pass 1 as the argument to the first thread, we call it with the following:

```
int arg1 = 1;
failUnless0(pthread_create(&t1, NULL, hello, &arg1));
```

Then, to use this value, we modify the body of `hello` to cast the argument to an integer pointer and use this value:

```
int* iArg = (int*) arg;
printf("Hello from thread %d\n", *iArg);
```

Make this change (and the corresponding change to the second thread creation) and verify that the threads now identify themselves when printing their greetings.

Now you're ready to apply what you've learned to the Mandelbrot problem. What you want to do is move the doubly nested loop after the `set pixels` comment into a function that will be called as a thread body. Set the code up to create two threads. The first thread will tackle the first half of the columns (values of `i` from 0 to `numCols/2`) and the second thread will handle the rest. Create a struct for the argument so you can pass the beginning and ending values of `i` to this function. (Note that `numCols`, `numRows`, and `pixels` are all global variables so you don't need to pass them.) Be sure to use `pthread_join` to wait for your threads to complete. (Create both and then wait for both; if you create and then immediately wait for one before creating the second, you're not letting the threads run in parallel.)

Once you've written the code, verify that you produce the same image. Then use `time` to compare the running time of the threaded program to the original serial implementation. The first two numbers from `time` should be nearly identical because the programs do almost exactly the same things and therefore need the same amount of CPU time. The wall clock times (next number) should differ, though. The *speedup* of our parallel implementation is the serial wall clock time divided by the parallel wall clock time. Since we are creating two threads and the computers have enough cores to run each on a separate thread, an ideal speedup would be 2. Yours is likely less than this, but should be greater than 1.

Next, try a different approach to parallelizing this program. Swap the order of the two loops (moving the `j` for loop to the outside) and then parallelize that version so that each thread runs a range of values of `j`. Again, verify that the resulting image is the same and compute the speedup. If you have time, see what happens when you parallelize the inner loop instead of the outer one (for each loop ordering). We'll talk about the results in class.