CS 226: Operating systems and networking
Fall Term, 2012

Lab 4

Using threads for parallelism

In this lab, you'll look at using multiple threads (both manually programmed and managed via OpenMP) to speed up a computationally-intensive task. Begin by copying the code from the `lab4` subdirectory of the course directory:

```
cp  /home/courses/cs226/lab4/* destination
```

where *destination* is where you'd like to copy them to. There are two files, `mandelbrot.c` and `bmp.h`. Compile them with

```
  gcc -Wall -std=c99 -o mandelbrot mandelbrot.c
```

and run the resulting executable; it takes a single command line argument, which should be a filename ending in `.bmp`. This extension stands for "bitmap", which is a graphics format. The program will create the file whose name you give. To view this file, you'll need to mount your euclid directory onto your lab machine. (This is one unfortunate aspect of working on a remote system.) You can find directions to do this on `http://cs.knox.edu/mounting.html`.

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 a pair of coordinates. 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 they started growing exponentially within the first 1,000 iterations.

The program 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, 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 blah.bmp
```

where `blah.bmp` is replaced with the name of the file you'd like to create. (Note that when doing timing experiments like this, it matters what else is running on the system. The class should try to spread out over the different systems (`euclid`, `huygens`, `barrow`, `descartes`, `newton`, and all the lab machines) and potentially coordinate when different runs are made to minimize interference.)

The focus of this lab will be in trying to reduce this delay. Begin by having the program create two new threads, one to compute the left half of the image and the other to compute the right half. To do this, you'll move the doubly-nested loop that fills in the pixels into a thread routine. The first thread will tackle the first half of the columns, leaving the others for the second thread. Use `pthread_create` to create two threads, following the the approach taken in lecture (the code we wrote is on the course website (`http://courses.knox.edu/cs226`, under lectures). On the Linux systems, you may need to add `-lpthread` to your compile line; this tells the linker to link with the PThreads library.

As in class, you'll need to use `pthread_join` so that the parent thread waits for all of the created threads to complete before generating the output file. The first argument to this function is the data structure associated with a thread ("returned" as the first argument of `pthread_create`). The second argument can be `NULL`. The `pthread_join` command causes execution of the calling thread to pause until after the argument thread completes. Thus, you'll want to call `pthread_join` once on each child thread, but only after both have been created.

Verify that this change causes the image to be correct. Next, time the program using `time` and compare the results to those of the original program. The first two numbers should be nearly identical between the two implementations because they do almost exactly the same things and therefore need the same amount of CPU time. The wall clock times 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. In practice, parallel programs cannot normally achieve *linear speedup*, as this would be called, since some parts of the program run serially and the system calls to create and manage threads incur some overhead.

In my runs, I get a speedup of only about 1.3 using 2 threads, hardly impressive for a program that splits so nicely into separate pieces. Try some other approaches to parallelizing this program (parallelize the inner loop instead, swap the loops, etc). To facilitate your exploration of the different parallelization options, use OpenMP (following the example from Wednesday's class). Start by replicating the results of your hand-coded threads, but then branch out and try other approaches.