Race conditions in OpenMP

(using the Mandelbrot set as an example)

Abstract

This lesson uses the Mandelbrot set to demonstrate race conditions in an OpenMP program using parallel for loops.

Using this lesson

This lesson can be used in a variety of ways. This document presents the lesson in a form similar to a lecture narration. The provided slides follow this presentation. The lesson can also be used as a lab, in which case the student instructions (which overlap significantly with this document) can be provided to students.

Race conditions

A race condition is when different tasks in the program have operations that can interfere with each other and the correctness of the program depends on the order in which those operations are performed. Typically, a race condition occurs when multiple tasks use the same variable and the order in which they do it affects the result. This lesson looks at two types of race conditions and their solutions. We'll start by illustrating each kind with non-programming examples.

For the first example, consider the kitchen in a restaurant as an example of a parallel system. The dishes to be prepared are the tasks that need to be run and the cooks are the processors or cores available to run those jobs. Representing the shared variable is a single cutting board which is needed to prepare several of the dishes. This might be ok if only one cook at a time ends up using it, but problems could occur if 2 cooks attempt to use it at the same time, resulting in cross contamination between the dishes they are making or a shouting match between them. The situation is easily resolved by getting separate cutting boards for each cook, corresponding to using separate variables in each task.

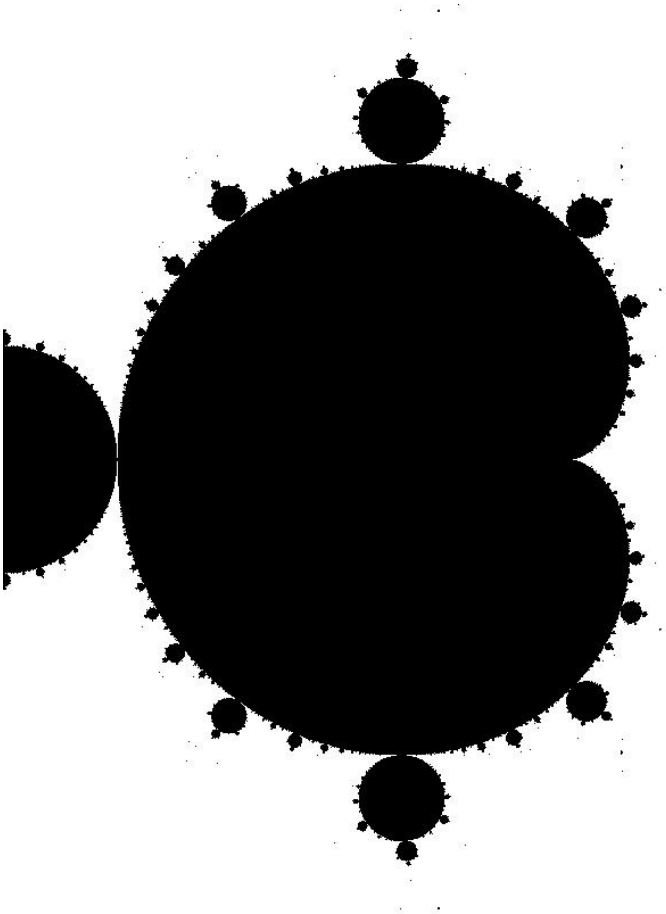
As an example of the second kind of race condition, consider a group of chefs making an opera cake (shown to the right; image by Arnold Gatilao, 2007, from https://en.wikipedia.org/wiki/Opera_cake and shared with a CC 2.0 license). The defining feature of this cake is its large number of layers. In principle, these layers could be made by different chefs and then added together to form the final product. However, the tray on which the cake is built (and the partially-built cake on it) cannot be

used simultaneously by multiple chefs since the layers must be deposited one at a time. Because the layers made by each chef must be combined, this situation cannot be entirely resolved by providing more resources and is thus different from the first type of race condition. The need to combine different parts means that resolving this sort of situation requires what is called a reduction rather than simply providing additional resources.

Mandelbrot set

To illustrate race conditions with code, we'll use the Mandelbrot set. This is a famous fractal which is defined by a mathematical operation. The details are not crucial to understanding the example, but we include them here for completeness. The general idea is based on points in the complex plane, where complex numbers represent points in 2D. The imaginary part of a number gives its y coordinate and the real part of the number gives its x coordinate. For example, the number 1+2i represents the point at x=1, y=2.

The Mandelbrot set is defined using a recursive process on complex numbers. When this process is run on a complex number z, the result of the first round is z itself and the result of round i is z plus the square of the result from round i-1. The Mandelbrot set is the set of all points for which this process stays bounded. In practice, representations of the Mandelbrot set are created by running the process for a set number of rounds on each point and including those whose value remains below a threshold for that number of rounds. Here is the Mandelbrot set depicted in this way, using 1,000 rounds and a threshold of 4:



The black points are in the Mandelbrot set and the white points are not. The image is centered around the value 0, which represents the point (0,0).

For more details, including color representations of the Mandelbrot set, the reader is referred to its Wikipedia page: https://en.wikipedia.org/wiki/Mandelbrot_set.

Parallel for loops in OpenMP

The image shown above was computed using the program mandelbrot.c available in the module directory.[1] This image uses a function mandelbrot that takes a pair of coordinates and returns the color to draw that point, which is either 0 (black; in the set) or 255 (white; outside the set). It also has code to create a file (named on the command line) with the Mandelbrot set representation in the ppm format. The only part of the program we need to focus on to demonstrate race conditions, however, is the following loop in `main`:

```
//set pixels
```

[1] Compile the code and run it with a command line argument specifying the file name for the .ppm file it will create as output. The commands for compilation and running it are in the comment at the top of the code file.

```
for (int j = 0; j < numRows; j++) {
     for (int i = 0; i < numCols; i++) {
          x = ((double)i / numCols -0.5) * 2;
          y = ((double)j / numRows -0.5) * 2;

          pixels[i][j] = mandelbrot(x,y);
     }
}
```

This loop sets the color for each pixel of the output image. The pixel coordinates are `i` and `j`. First, the loop converts each coordinate to a real value between -1 and 1. Then, it calls the `mandelbrot` function and stores the result in the 2D array `pixels`.

We will use OpenMP to parallelize the `j` loop here. The idea is that different iterations of the j loop are independent since the color of each pixel depends only on its coordinates and thus all the iterations are independent. (This idea is nearly correct, but our first attempt fails because it shares the variables `x` and `y` between tasks.) Converting the for loop into a parallel for loop partitions the loop iterations between several tasks, each of which can run in parallel.

The syntax to make a parallel for loop in OpenMP is to add a pragma (basically a suggestion to the compiler) right before it. For the loop above, this looks like

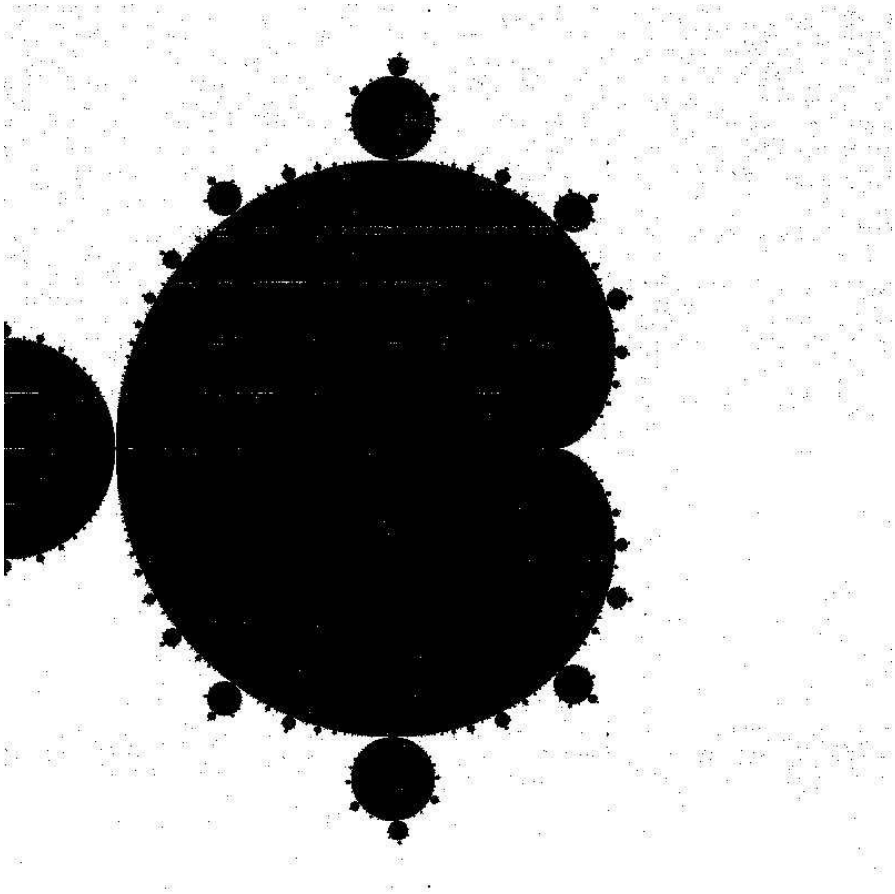
```
#pragma omp parallel for
for (int j = 0; j < numRows; j++) {
```

The `omp` part specifies that this pragma concerns OpenMP. The words `parallel for` mean that you want to create multiple tasks and split the iterations of the following line (which must be a `for` loop) between them. By default, this creates one task per core; other instructions can be added to create a different number of tasks or to customize them in other ways (as we'll see).

If you add this line, recompile,[2] and then run the program, it runs noticeably faster. Prepending the command lines with `time` lets us measure the running time of each version. On my system,[3] the original (serial) version took 3.39 seconds and the parallel version took only 0.58 seconds, for a speedup of 3.39/0.58=5.84. Unfortunately, it also gets the wrong answer since it suffers from a race condition based on unnecessarily shared variables. Specifically, the program produces an image such as the following:

[2] With gcc, you need to add -fopenmp to the compile command.

[3] A 2.1GHz AMD Opteron with 16 cores running Fedora Linux version 31. The code was compiled with gcc 9.2.1.



This image is largely correct, but has a number of artifacts, namely the "haze" of dark pixels around the dark figure and the partially white streaks in the figure. (The differences are even more visible in the .ppm files actually created by the program; see correct.ppm and withRace.ppm in the module directory.) Both of these are caused by the variables `x` and `y`, which are shared between the different tasks. The resulting race condition means that sometimes the values of `x` and `y` are changed during the computation and the wrong color is selected for a given pixel.

Fixing the race condition

In order to fix the race condition, we need to end the interference between tasks by having each task use its own copy of the variables. There are two ways to do this with the given code. The first is simply to delete the existing declaration of these variable and to declare them inside the loop as follows:

```
#pragma omp parallel for
for (int j = 0; j < numRows; j++) {
     for (int i = 0; i < numCols; i++) {
```

```
            double x = ((double)i / numCols -0.5) * 2;
            double y = ((double)j / numRows -0.5) * 2;

            pixels[i][j] = mandelbrot(x,y);
        }
    }
```

Now `x` and `y` are local to the loop. The compiler knows to create separate variables for each task in this case.

The other solution is to explicitly say that the variables shouldn't be shared. This is done by adding the qualifier `private` to the pragma as follows:

```
#pragma omp parallel for private(x,y)
for (int j = 0; j < numRows; j++) {
     for (int i = 0; i < numCols; i++) {
          x = ((double)i / numCols -0.5) * 2;
          y = ((double)j / numRows -0.5) * 2;

          pixels[i][j] = mandelbrot(x,y);
     }
}
```

Now the compiler knows that these variables should be private to each task. (There is also a qualifier `shared` that tells it that variables should be shared between tasks.)

Making either one of these changes fixes the program so that it creates the same image as the original serial version, but approximately 6 times as fast.

Using a reduction

To illustrate a race condition that can be resolved using a reduction, we return to the Mandelbrot set computation, but this time attempt to count the number of black pixels in the image, which corresponds to the number of times the `mandelbrot` function returns 0. To count these pixels, we'll create a counter variable `numBlack` and add an if statement to check the return value of `mandelbrot`. The following is the new main loop, with modifications shown with underlines:

```
int numBlack = 0;  //number of black pixels in image

#pragma omp parallel for private(x,y)
```

```
for (int j = 0; j < numRows; j++) {
     for (int i = 0; i < numCols; i++) {
          x = ((double)i / numCols -0.5) * 2;
          y = ((double)j / numRows -0.5) * 2;

          int pixel = mandelbrot(x,y);
          pixels[i][j] = pixel;
          if(pixel == 0)
               numBlack++;
     }
}

printf("number of black pixels: %d\n", numBlack);
```

When creating an 800 by 800 image, the correct number of black pixels (computed by removing the pragma line) is 222,680. When the code is run as shown, the value is slightly undercounted. This occurs because of a race condition on the variable `numBlack` when it is incremented. Although `numBlack++` is a single statement of C code, it requires multiple operations: loading the current value into a register, incrementing that register, and then storing the value from that register into memory. An undercount occurs when one thread performs these operations and another thread performs its load operation before the first thread finishes.

This code pattern where the results of many computations (the color calculation for each pixel) is combined into a single value is called a reduction. Other common examples include adding up all the numbers in an array or finding the minimum (or maximum) value in an array.

Because reductions are fairly common, parallel for loops in OpenMP provide a special type of variable for them. To inform the compiler that a variable is participating in a reduction over iterations of a loop, use the reduction qualifier. It is also necessary to specify the variable and the kind of reduction being performed. Here is the pragma for our program using this qualifier to denote that numBlack is being used in a summing reduction:

```
#pragma omp parallel for private(x,y) reduction(+:numBlack)
```

Using this line makes the program function correctly, causing it to report 222,680 black pixels for an 800 by 800 image.