False sharing

In this lab, you'll play with a program that demonstrates false sharing and how to remove it by adding padding. The lab builds on our earlier work generating the Mandelbrot set. The program we'll be using is `ppm-balance.c`, which reads in a .ppm file generated by our Mandelbrot program and checks the number of black pixels generated by each thread in the Mandelbrot program. This is interesting because the black pixels take longer to generate[1] so the difference in their numbers between threads is a measure of load balance.

Open the file and search for the pragma (there is only one), which is attached to the following block of code:

```
#pragma omp parallel num_threads(numThreads)
{
     int threadNum = omp_get_thread_num();
     int actNumThreads = omp_get_num_threads();

     int low = numRows*threadNum/actNumThreads;
     int high = numRows*(threadNum+1)/actNumThreads;

     for (int j = low; j < high; j++) {
          for (int i = 0; i < numCols; i++) {
               if(pixels[i][j] == 0)
                    numBlack[threadNum*dist]++;
          }
     }
}
```

This code uses the `parallel` construct of OpenMP to create `numThreads` threads, each running this block simultaneously. Each of these threads will count the black pixels in a contiguous group of rows whose pixel values were determined by a single thread in the Mandelbrot program (assuming that program and this one are run with the same number of threads).

The first two lines of the block use OpenMP library calls to set `threadNum` and `actNumThreads` to the thread number (a value from 0 to 1 less than the number of threads, with each thread having a distinct value) and the number of threads respectively. The next two lines use these values to determine the range of row indices for which the current thread will be responsible; the expressions are created so that the ranges are disjoint, cover 0 thru `numRows`-1,

[1] The pixel color is determined by a loop that iterates a process on a value until either the value exceeds a threshold or a given number of iterations have been performed. Black pixels are those for which all the iterations are performed. White pixels are those that cause the value to exceed the threshold before this point, which causes the loop to exit before performing all the iterations.

and make the size of each range the same up to rounding. All of this preamble, plus using `low` and `high` as the bounds of the next for loop, make that loop equivalent to a parallel for loop with default scheduling in OpenMP. The code is written as it is rather than using a parallel for loop explicitly so that each thread knows its ID number without multiple calls to `omp_get_thread_num`; equivalent code using a parallel for loop would need to call this method in each iteration rather than once per thread.

Next, the nested for loop iterates through all the pixels in the rows assigned to this thread to find those with value 0 (i.e. the black ones). Whenever one of these is found, the value of `numBlack[threadNum*dist]` is incremented. `numBlack` is the array storing the counts of black pixels. The variable `dist` controls how close together the counts of black pixels are stored. When it has value 1, they are stored in consecutive array cells, one cell per thread. When `dist` has value 2, the counts are stored in the even array cells, with an unused cell between each pair of pixel counts. In general, there are `dist`-1 unused array cells between each pair of pixel counts.

The purpose of `dist` is to enable and eliminate false sharing. When `dist` is 1 or other small values, then multiple pixel counts are stored in a single cache line. Since each pixel count is used by a different task, this causes false sharing. As the value of `dist` increases, the distance between pixel counts increases, eventually reaching the point where only one count occurs in each cache line, completely eliminating the potential for false sharing.

To get the timing measurements, we use the call `gettimeofday`, which gives the current clock time to microsecond precision. This function is called twice, once before the block quoted above and once after. The difference between them is then the time used by the block. This technique is used instead of running the entire program with `time` since that measures the time of the entire program execution rather than just the block of interest. Since the rest of the program takes significant time (especially reading the input), that part obscures the running time difference caused by false sharing, which is on the order of milliseconds.

Compile and run the program with the following lines:

```
gcc -Wall -std=gnu99 -o ppm_balance ppm_balance.c -fopenmp
./ppm_balance 1600x1600.ppm 16 1
```

The first argument is the name of the .ppm file to use. The second argument (value 16) is the number of threads to use (`numThreads` in the code). The third argument (value 1) is the distance between used cells in `numBlack` (`dist` in the code). Thus, this run puts the pixel counts into adjacent array cells, allowing lots of false sharing. This is reflected in the running time, which is the first value printed by the program (in microseconds). The program also prints the number of black pixels in each region; you can see that the first and last threads of the Mandelbrot program had to identify substantially fewer black pixels.

Next, run the program with a larger value of `dist`:

```
./ppm_balance 1600x1600.ppm 16 16
```

This run only uses every 16th cell of the array, which separates the counts substantially. Depending on your system, this should reduce or eliminate false sharing. Thus, the reported time should be significantly lower.

The program also illustrates one of the possible solutions to false sharing. The key to eliminating it is to prevent different threads from accessing nearby memory addresses. Often, the memory addresses are cells of an array of fields of a struct since those are guaranteed to be nearby in memory, but other variables can also be used in false sharing situations. Our strategy of lengthening the array and only using some of the cells is basically to add padding to the variables to separate them. The struct version of this is to add unused fields. The computation can also be restructured to avoid the sharing; for example, the false sharing in this program program would be eliminated if each thread kept its count in a private variable and then put only the final total into the array.