Multiprocessor caching and false sharing

Abstract

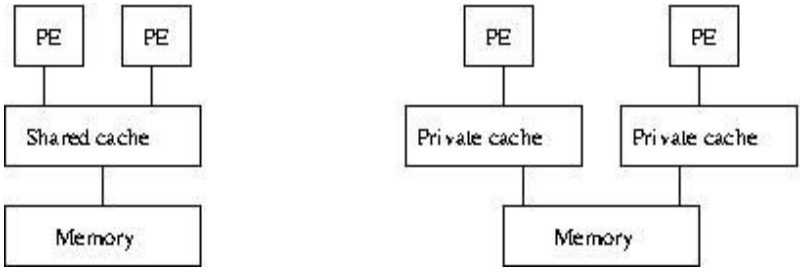
This lesson presents the cache coherence problem for private caches and the potential for false sharing. It includes a short program that demonstrates the occurrence of false sharing in a program designed to check the load balance of the parallel Mandelbrot program discussed in the "Race conditions" lesson of unit 4 on OpenMP.

Using this lesson

It is envisioned that the material in the next section be given to students in a lecture or possibly as a reading. The skeleton of presentation slides are included to support this. Code for use in demonstrating false sharing is provided. This could be used in class, by explaining it and showing the results (following the rest of this document), or as a lab or HW using the provided exercise instructions for students.

Multiprocessor caching

In the previous discussion of caching, the cache was presented as living between the processor and the memory system. This picture is more complicated for parallel systems because processing is done by multiple entities (potentially multiple processors but definitely multiple cores within each processor). The following images depict two ways this could be done, using PE (processing element) to stand for a processor or core:



The image on the left shows multiple PEs using a single cache while the image on the right shows each PE with a private cache. Real systems typically have several levels of cache, often arranged as a combination of these ideas; the level(s) of cache closest to the PEs are private while the lower level(s) of cache are shared.

Having private caches creates an added complication when code on the different PEs access the same memory address. In order to preserve the illusion of having a single memory,

changes made by one of the PEs must be made visible to the others rather than just being made to the private cache. This need to share changes is called the *cache coherence* problem since it is concerned that the caches maintain a single coherent view of the contents of memory.

There are different ways to provide cache coherence, but the upshot is that private caches for the different PEs will sometimes need to invalidate a cache entry because that entry is needed by a different private cache. This is an unavoidable cost of using the same data on different PEs. Sometimes there is also an avoidable component of the cost, however, and that is the focus of this module.

Recall that each cache entry stores data from more than one address; this is how caches exploit spatial locality. In a multiprocessor setting, however, this can lead to a phenomenon called *false sharing*, which is when different PEs access different memory addresses but those addresses are close enough to be stored in the same cache entry. No data is actually shared between the PEs, but the memory system still invalidates the cache entries as if it were.

Demonstrating false sharing

In order to demonstrate false sharing, we'll use the program `ppm-balance.c`. This program takes a .ppm file, which stores an image using a single number of each pixel. The image is conceptually divided into strips, each consisting of a contiguous collection of rows. Each strip is examined in parallel and the number of black pixels (i.e. those whose value is 0) is counted. The program then prints the amount of time the counting took in microseconds and the black pixel counts. Its ostensible purpose is to measure the load balance of the program `mandelbrot.c` used in the "Race conditions" module in unit 4. The strips correspond to the pixels generated by each thread and black pixels require more processing than white ones.[1]

The key part of `ppm-balance.c` is the following loop, which actually counts the black pixels:

```
#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++) {
```

[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.

```
            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. 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, 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.

Returning to the code itself, the nested for loop iterates through all the pixels in these rows to find those with value 0. Whenever one of these is found, the value of `numBlack[threadNum*dist]` is incremented. `numBlack` is the array storing the counts of black pixels. When `dist` has value 1, the loop stores the pixel counts 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.

To run the program, it must be supplied with appropriate command line arguments. The first argument is the name of the ppm file to read. The second and third arguments are optional,

but they give the number of threads to request (`numThreads` in the code) and the distance between used cells in `numBlack` (`dist` in the code), respectively. The system I ran on[2] supports 16 hardware threads so I used that value as `numThreads`. I found the best illustration using the following parameters:

```
./ppm_balance 1600x1600.ppm 16 1
./ppm_balance 1600x1600.ppm 16 16
```

The first version uses every cell of `numBlack` and thus suffers from false sharing. The time varied somewhat, but the fastest times were in the range 16-17ms. The second version uses only every 16th cell of the array and runs complete in 7-8ms. Thus, false sharing causes the time of this loop to roughly double.

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 our example program would be eliminated if each thread kept its count in a private variable and then put only the final total into the array.

Common pitfalls

Students sometimes have trouble understanding the role of the cache, basically that the system should behave as it would if there weren't a cache. This is an issue when talking about multiprocessor caching since the need for cache coherence is only clear if they're thinking in terms of an uncached system.

A particular piece of code that I anticipate being challenging for students is the calculation of the range of indices for each thread in the sample code:

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

Students can actually treat this as a black box, but in order to understand it, they need to see that `high` is the same as `low` with `threadNum` one higher, that `low` is 0 when `threadNum` is 0 and `high` is `numRows` when `threadNum` is `actNumThreads`-1.

[2] A 2.1GHz AMD Opteron with 16 cores running Fedora Linux version 31. The code was compiled with gcc 9.2.1. These same parameters worked on a login node of Blue Waters.