CS 575, Project 3: False Sharing

Peter Ferrero, Ph.D. Student, Oregon State University

May 6, 2018

Project 3 was ran on a Windows Desktop with an AMD FX(tm)-8320 Eight-Core 3.50 GHz Processor with 16.0 GB of installed RAM. The goal of this project is to examine how false-sharing within cache lines affects performance when using multiple threads to repeatedly add a value to the same entry in an array of structures. In theory, any two threads altering data values that exist on the same cache line will cause the cache to mistakenly invalidate all of the data entries residing on the same line. Thus, each time an invalidation occurs, the cache line will need to be reloaded from the main memory. This operation is very expensive in terms of performance. Thus, two easy fixes exist to remedy the situation. The first fix involves padding each structure within the array with empty data values. The idea is that if the padding is sufficient, each structure will exist on its own cache line. Then any thread changing a value within the structure will not invalidate the data in the structures belonging to the other threads. Thus, no expensive cache line reloads will be required. The second fix is to declare a temporary summation variable within each thread's private stack. The thread will use the variable for summing within the loop and then write the final value of the sum to the structure at the end of the loop. Thus, the cache line is only invalidated once per thread, drastically reducing the number of cache line reloads. The performance for this project was calculated as follows:

$$\text{Performance} = \frac{\text{Number of Sums} \times 8 \text{ Structs}}{\text{Execution Time} \times 1,000,000}$$

where the units of performance are MegaSums/sec. Since the computer these experiments were ran on contains eight cores, eight structures were defined instead of four. The simulation was ran 20 times to ensure a valid average performance was calculated in addition to a peak performance. For the padding experiments, the number of pads was varied from zero to fifteen. Since each cache line contains space for sixteen words, fifteen is the number of pads that will ensure each of the eight structures resides on its own cache line. The performance results of these experiments are summarized in Table 1. The padding experiments values are from the execution using fifteen pads since this allows for comparison between the padding and temporary variable results. The average and peak performance values are reported. The average and peak values agree well with one another signifying good consistency amongst the experiments.

Table 1: The peak and average performance for fine-grain and coarse-grain parallelism with static and dynamic thread workload scheduling. Performance values are given in MegaBodies calculated per second.

Threads		1	2	4	8
Padding	Peak	232.714	465.130	865.033	1652.18
	Average	232.088	461.414	857.954	1523.57
Temporary Variable	Peak	232.633	465.736	867.547	1684.07
	Average	231.657	459.976	864.367	1552.82

From these results, it is seen that for each fix, the effects of cache line reloading is virtually eliminated. The performance scales roughly with the number of threads used during execution.

Figure 1 explores the effect of padding on program performance. It is seen that padding does not affect the performance of a single thread however, it become very important as the number of threads increases. For two threads, three pads are required to place four structures on two separate cache lines. The threads are then free to work on their assigned structures without triggering false-sharing within the cache. Similarly, using four threads required seven pads to achieve optimal performance and eight threads required all fifteen pads. As noted earlier, fifteen pads corresponds to the case where each of the eight structures resides on its own cache line, thus each of the threads is allowed to perform its work without interfering with the other threads. However, this approach is extremely wasteful in terms of the memory usage. To effectively use eight threads, the amount of memory required is sixteen times that which is needed for a single thread. Thus, a better approach is to define temporary variables to store the summation values within the parallel loop. The horizontal lines in Figure 1 denote the performance of using temporary variables. These results show that this approach achieves identical performance while requiring a fraction of the memory usage.

The results of this study show that false-sharing can easily be solved using padding or temporary variables. Both of these techniques allowed for optimal parallel thread performance to be recouped. However, since memory usage is usually a concern, it is recommended to store any variables that will potentially cause cache line conflicts as temporary variables on each thread's private stack. This approach allows for best of both worlds in terms of memory usage and performance.

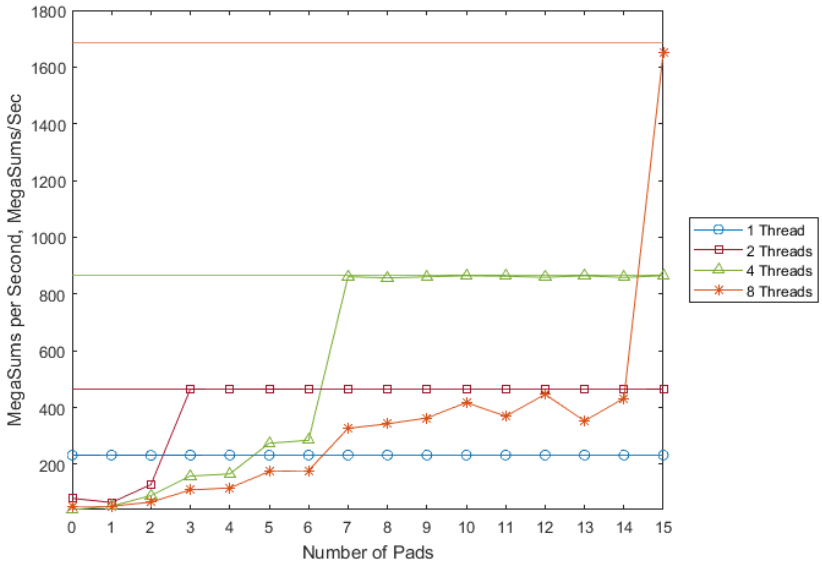

Figure 1: The effect padding and temporary variables on peak performance when using multiple threads.