Assignment 1: Multithreading and the Problem of False Sharing

Tai Duc Nguyen
ECEC 622: Parallel Computer Architecture

April 20, 2020

Abstract

In modern computing systems, CPUs often have multiple cores that can manage multiple threads. Mutithreading can accelerate the computing capabilities of many applications. One such application is called SAXPY, or Single-Precision A·X Plus Y: a scalar multiplies a vector, which is added to another vector. Variants of this extremely simple benchmark can test a system's computing and memory bandwidth. In this assignment, the problem of False Sharing is explored in multithreaded applications using SAXPY as the benchmark.

1 Experimental Setup

In order to measure the effect of False Sharing, a baseline execution time is established by running SAXPY in a serial fashion (no False Sharing): looping through all N elements and do N operations $\bar{Y} = a \cdot \bar{X} + \bar{Y}$.

Then, a multithreaded application is built such that N operations are chunked into T threads: thread i will operate on all elements from i*chunk_size to i*chunk_size*2. This **chunking method** can reduce the effect of False Sharing but may not do so completely because there is a chance that the cache line of each processor/thread is independent from one another; or, only overlap slightly.

Finally, another multithreaded application is created using the **striding method**: thread i will execute the operation above for element i, i + num_threads, i + num_threads*2, i + num_threads*3... and so on. This will guarantee the effect of False Sharing because cache coherency is maintained on a cache-line basis, and not for individual elements. Hence, simultaneous updates of individual elements in the same cache line coming from different processors/threads invalidates entire cache lines every time the vector $\bar{Y}$ is updated by any processors/threads.

Note: running `make all` will generate the report in `rpt.txt`

2 Experimental Result

Number of elements	Number of Threads	Execution Time (seconds) Serial	Chunking	Striding	Comparison Chunking vs Serial	Striding vs Serial
10000	4	0.000002	0.000250	0.000192	125.00	96.00
10000	8	0.000002	0.000269	0.000244	134.50	122.00
10000	16	0.000002	0.000536	0.000397	268.00	198.50
1000000	4	0.000814	0.000576	0.000782	0.71	0.96
1000000	8	0.000519	0.000513	0.001306	0.99	2.52
1000000	16	0.000547	0.000586	0.001815	1.07	3.32
100000000	4	0.051123	0.035417	0.077242	0.69	1.51
100000000	8	0.051444	0.037226	0.073843	0.72	1.44
100000000	16	0.052356	0.038447	0.486753	0.73	9.30

Figure 1: Results of execution times for all 3 scenarios: Serial, Chunking, and Striding

3 Discussion

From Figure 1, it is apparent that multithreading only provides computational benefits at $N > 10^6$ elements. Below 10^6, the overhead of threads, and their associated data structures made the total execution time much higher than doing things serially. At $N = 10^6$ elements, chunking provides some small benefits but not much due to some False Sharing is still present. Striding, however, does not provide any benefits due to false sharing. At $N = 10^8$ elements, same behaviors applies to both chunking and striding, excluding the case where $T = 16$ on striding. In this case, each thread has very little work, hence, False Sharing is magnified: every operation on any thread will cause the cache line to be invalidated.