EE-193: Parallel Computing
Lab 2: Breaking your server

This assignment is to implement a program like the one we discussed in the lecture. The goal is to use false sharing as much as possible, so as to build a program that brings the memory system to its knees.

We've supplied you with the file server_breaker.cxx. It is already written – with the exception of the function *compute_thread*(), which you must write yourself. Its arguments are documented in the file.

The surrounding code, in *main*(), picks various configurations; it calls *run*() to run them. Mostly, *run*() just picks which memory addresses to use, starts/ends a timer, and prints out statistics; it relies on *compute_thread*() to do the hard work.

Your job is, first, to write *compute_thread*(). It should work as follows:

1. Sweep through the lines and store into them. Store 1 into our location on the first line, 2 into our location on the second line, etc.
2. Do #1 (i.e., the stores just above) *n_stores* times. Actually store the same data every time. E.g., each time you would store 1 into the first line.
3. Similarly, sweep through the lines, read back the data and check it. Do this *n_loads* times. (Note that if *n_stores* is zero, then there will be nothing to check! In this case, you should do the loads but not check them)
4. Do the entire loop #1 to #3 above *n_loops* times. Each time, keep incrementing the numbers we store. So if there were 4 lines, then on the first loop we would store and load the data values 0,1,2,3; on the second loop we would store and load the data values 4,5,6,7; etc.
5. Note that the code pick *n_loops* so that we always access the same number of cache lines. For example, you will note that when we are doing both writes and reads, and we are using 1000 cache lines, each thread will loop over them 204800 times. However, when we use 10000 cache lines, we only loop over them 20480 times. This keeps the total number of accesses per thread constant across different configurations, and is meant to help you compare results more easily.

Next, collect the results. You may note that the first run is often slower due to a cold cache; you can ignore this and only report the subsequent runs.

Finally, analyze the data. Usually when we add more cores to a problem, we expect the amount of time taken to shrink. That does not happen here; sometimes adding more cores makes no difference, and sometimes it hurts. Why?

Similarly, sometimes adding more lines hurts badly, and sometimes not. Why? If you are doing only reads, and all of the lines fit into a core's L1 or L2, would you expect that adding more cores would increase the amount of time that the program takes?

Talk about a few things:

- how cache coherence treats writes differently from reads
- size of the L1, L2 and L3 caches
- time spent accessing home agents over the ring

You get extra credit if you can show that you occasionally get the wrong answer, and that it's the computer's fault and not yours ☺. (This is highly unlikely, but possible).

Logistics:

- The lab assignment is due ????? at midnight. You must work on this assignment on your own. Submit your file via **provide**, as well as a copy of your report as a PDF.
- Remember to compile with the line **c++ -pthread -std=c++11 -O2 server_breaker.cxx ee193_utils.cxx**. You may add debugging or optimization flags as needed.

Resources:

- From now on, we will stop using Dell24. Instead, use any of the lab machines in rooms 116 or 118 for both development and benchmarking. To ensure consistent benchmarking, please reboot the machine before collecting your final benchmark timing. Note that the machines in lab 120 are older, less powerful machines; please do not use them.