EE155/CS122: 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. *Compute_thread*() is in a separate file, server_breaker_thread.cxx, which is the only .cxx file that you should turn in.

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. Make sure that your data-checking code actually works (e.g., by purposely introducing a bug and being sure that the bug gets caught) – otherwise you could easily have a bug in your code and not know it. (Note that if *n_stores* is zero, then there will be nothing to check! In this case, you should use the same code as usual to do the loads and checking, but simply do not print out an error message).
4. Do the entire sequence #1 to #3 above *n_loops* times. Each time, just start from 1 again. Note that g_mem[] is an array of unsigned characters; you should thus only store and load the lower 8 bits of the numbers you use.
5. Note that the code picks *n_loops* so that each thread always does the same number (specifically, 400M) of total loads and stores to our cache lines. For example, when we are doing 4 writes and 1 reads, and we are using 1000 cache lines, each thread will be told to do 400M / ((4+1)*1000) = 80K loops. However, when we use 10,000 cache lines and do 2 loads and no stores, we only loop over them 400M / (2 * 10K) = 20K times. This keeps the total number of accesses per thread constant across different configurations, and is meant to help you compare results more easily.

Collect the results and include them in your report.

Background

> The L1 cache holds 32KB of data, which is 512 lines. The L2 holds 256 KB of data, which is 4K cache lines. The L3 holds 8MB, or 128K lines. Thus, 100 lines will fit in the L1. 1000 lines will not, but will fit in the L2. 10000 and 100000 lines will fit in the L3, but not the L1 or L2. Each of the 4 cores has its own L1 and L2; there is a single L3 shared by all cores.

***Reference results to be used for all questions below*:**

0 stores + 2 loads

	1 thread	*2 threads*	*4 threads*	*8 threads*	*16 threads*
100 lines	400ms	410ms	430ms	860ms	1700ms
1K lines	460ms	470ms	490ms	910ms	1800ms
10K lines	650ms	660ms	700ms	1400ms	2700ms
100K lines	670ms	690ms	750ms	1400ms	2800ms

4 stores + 1 load

	1 thread	*2 threads*	*4 threads*	*8 threads*	*16 threads*
100 lines	310ms	2200ms	7500ms	7200ms	14000ms
1K lines	410ms	1400ms	2200ms	3600ms	7000ms
10K lines	2400ms	2400ms	2600ms	4400ms	8900ms
100K lines	2400ms	2900ms	3500ms	5200ms	11000ms

Submitting a correct program is worth 55 points. You should also answer the questions below *using the reference results just above*:

Question #1 (*10 points*). Look at the chart above for the load-only case. All 6 cases of 100 and 1K lines and 1T, 2T and 4T took roughly the same amount of time (about 440ms). Can you explain why adding more cores does not change execution time? Assume that each thread is assigned to a separate core until we run out of cores. Remember also that each thread gets assigned the same full workload, so that as we add more threads the total *number* of loads scales up accordingly (which may not guarantee that the total execution time scales).

Question #2 (10 points total)**.** Moving from 4T to 8T and 16T resulted in roughly doubling the execution time each time we doubled the thread count no matter how many cache lines we're using. Let's try to explain why, at least for the case of 100 or 1K lines. Focus on the cases that took roughly 440ms. A bit of math shows that the inner loop is then taking about 4 cycles; a bit of looking at the algorithm shows that the inner loop performs two loads. It is thus averaging .5 loads/cycle.

- From the slide 5 of archreview/quantitative_performance, are we limited by the ability to issue loads? Why or why not? (2 points)
- From slide 20 of archreview/quantitative_performance, are we limited by our caches? Why or why not? (3 points)
- Assuming that we are limited by issue slots to the ALUs (as the only option left), can you explain why moving from 4T to 8T and 16T resulted in roughly doubling the execution time? Remember that we have four actual cores, but each one is hyperthreaded and supports two threads issuing with SMT. (4 points)

Question 3 (5 points)**.** While going from 100 to 1K lines made little difference (as per Question 1), going to 10K and 100K lines slowed runtime by roughly 50%. Can you explain this? You

don't have enough data to know the answer with certainty, but as usual try to be as specific as possible.

Question 4 (*10 points*). Now consider the times for the 4-stores-and-1-load case. For any number of cache lines, adding more threads consistently caused the total time to get a *lot* slower. Any explanation of why?

Question 5 (*10 points*). Consider again the times for the 4-stores-and-1-load case. For any number of threads greater than 1, going from 100 to 1K, 10K or 100K lines did not consistently slow things down, even though we moved from the L1 to L2 to L3 cache. Explain why this is (hint – look at slide #16 from the archreview/coherence slides).

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 on the day specified on the class calendar. You must work on this assignment on your own. Submit your **server_breaker_thread.cxx** and a copy of your report as a PDF, both via **provide**.
- Remember to compile with the line **c++ -pthread -std=c++11 -O2 server_breaker.cxx ee155_utils.cxx server_breaker_thread.cxx**. You may add debugging or optimization flags as needed.

Resources:

- Use any Linux system in 120 for your development and benchmarking. If you are going to collect benchmark data, then reboot the machine to be sure that nobody else is using it from a previous session.
- It is possible to log into the systems remotely. They are named lab120a – lab120v. However, these machines are not exposed to the general internet. Instead, you must first **ssh** into a publically-visible machine (e.g., linux.eecs.tufts.edu), and then **ssh** into a lab machine. To use X-windows tools on the lab machine, the second **ssh** should be "**ssh -X"**. Also, note that you will not be able to reset the machine remotely. Finally, you should use **top** or something similar to check if another person is actively using the machine – obviously that could greatly affect your runtimes.
- You can try to debug on your own laptop if you install a C++ compiler. However, note that ee155_utils.cxx uses Linux-specific functionality in the function assign_to_core(), so you code may not compile under Windows. This lab does uses that functionality to make the results more repeatable; you can just comment it out, but remember to restore it for your final run.