CS 378 – Project #4

Due Monday, Dec 1, Midnight

You may, but don’t have to, work in groups up to 2. Turn in a single project for all members of the group with all names listed.

Scaling of Simple Operations

Measure the scalability of simple memory operations when applied by multiple threads. Strong scaling varies the number of thread while fixing the problem size (each thread does less work as the number of threads is increased). Weak scaling increases the problem size as a function of number of threads (each thread does the same amount of work as the number of threads is increased and more total work is done).

On Stampede, measure the strong (10M operations) and week scaling (number of threads times 10M operations) for these operations:

• Reading a global variable. All threads read from the same global variable.
• Write a global variable. All threads write a value to a global variable without reading the global.
• Read Modify Write (non-atomic): All threads read, increment, and write a global variable. Yes, there are race conditions, ignore them.
• Atomic Read Modify Write: All threads do atomic increments to a global variable. You may use the gcc __sync builtins or C++11 atomics.
• Locking and unlocking a Mutex (Required for 2 person groups only, optional for 1 person)

You may use OpenMP or pthreads or c++11 to create the threads. Compare and plot scaling (1 thread runtime divided by N thread runtime) and runtime as a function of number of threads for all the operations. i.e. compare the performance of non-atomic RMW to atomic RMW to the others.

Perform experiments up to as many threads as there are hardware contexts on stampede. Submit jobs or use idev so your code runs without interference.

Notes

Run a couple loops to warm up the system before running the loops you collect numbers for. This should ensure the OS schedules the threads on different cores and thus ensure they are running simultaneously.

Don’t just launch threads and have them start the measurement loop. Launching threads is fairly expensive. You need to launch all your threads before allowing any of them to do the measured loop. A barrier or condition variable can accomplish the needed synchronization (there are other ways too).

Turning in

Please write up your experiment and results. Show the plots for all the results. Turn in by email an archive (.zip or .tgz) to the TA.
Sizable Bonus

Perform the same experiment as before, but instead of having threads access a single global variable, have them access an item in a global array whose index depends on which thread number they use. Vary the distance between the locations accessed.

For example, run with thread 0 accessing 0, thread 1 accessing 0 (TID * 0), then with thread 0 accessing 0, thread 1 accessing 1 (TID * 1), then with thread 0 accessing 0 and thread 1 accessing 2 (TID * 2). Continue this pattern for multiples up to 64 (and with varying the number of threads as before. Does the performance of the system change drastically at some multiple?)