CS 758: Programming Multicore Processors (Fall 2016)

Homework 5

Lock-free Queues

You should do this assignment alone. No late assignments.

Special thanks to Mike Marty at Google for providing the original code for this assignment.

Filelist for the assignment:

- Template Files
- C++ memory_order reference
- Herb Sutter's atomic weapons
- Scott Synthesis12
- Introduction to Lock-Free Programming

The purpose of the assignment is to give you experience optimizing parallel code using C++ synchronization mechanisms (including "C++11 atomics"). This exercise is intended to allow you to play with some classic locking techniques as well as newer lock-free methods in a simple yet challenging task. Hopefully, this will give you an idea about the subtleties and complexities involved in writing larger and more complex programs using these techniques.

You will test this assignment first on Ale-01 through Ale-07 - each containing two Xeon x5550 processors which have 4 cores each for a total of 8 cores.

Then, you will again use godel to see if your application is scalable to 64 cores (plus however many threads). You will use the same run.sh interface to godel.

Lock-free Programming

Lock-free techniques are supported in C++11 by the introduction of atomic<T>, which use platform-specific operations (think Load-Linked/Store Conditional or atomic Fetch-and-Add) to guarantee atomic reads and writes for atomic types. Additionally, an atomic compare/exchange method is introduced to allow publishing of updates.

```
template< class T > bool atomic_compare_exchange_*( std::atomic<T>* obj,
                                                    T* expected, T desired );
```

Programming Task: Fixed Size Queue

In this assignment, we expect you to implement a fixed sized queue with the following interface:

```
template<typename T> class FixedSizeQueueInterface {
 public:
  virtual ~FixedSizeQueueInterface() { }
  virtual bool Read(T* data) = 0;
  virtual bool Write(const T& data) = 0;
};
```

You should be familiar with queues from your undergraduate data structures courses.

To start you off, we provide you with a sample mutex-based queue implementation (MX) of the above queue interface (in queue.h), and a benchmarking program to evaluate the performance of queues with the above interface. Your job is to optimize the implementation using the following techniques -

1. Develop a queue implementation with fine-grained locks (FG).
2. Develop a lock-free circular queue implementation which works with a single producer and a single consumer (LF1).
3. Develop a lock-free circular queue implementation which works with a single producer and a multiple consumers (LF2).
4. Develop a lock-free circular queue implementation which works with a multiple producers and a single consumer (LF3).

Template for your convenience for HW5

A template is provided for your convenience with the basic setup of the program. This template provides you the initialized queue interface to work with, and the skeleton for the other implementations. You are free to change the benchmarking program for your own purposes, but your code will be checked by programs which assume you adhere to the interfaces provided. A makefile has also been added to the template that will help you in compiling the code. You might need to change the flags in the makefile during debug phase. Do revert them back to allow compiler optimizations.

Download the template from here.

- Queue.h contains the mutex-based queue implementation, as well as skeletons of other implementations for you to fill out.
- test.cc is the benchmarking program, which needs to be modified slightly as you add newer implementations, and is invoked thus-

./test <num_iterations> <producer_threads> <consumer_threads>

The number of producer threads is only used for the mutex and the LF3, while the number of consumer threads is used for the mutex and LF2. The benchmarking program runs the tests for a given number of iterations, and averages the throughput. Some useful macros and test parameters are provided at the beginning of test.cc

Problem 1: Utilize fine-grained Locks

Write a version of the fixed-sized queue with fine-grained locks using std::mutex. To achieve this, your implementation needs to have one lock per element of the queue, instead of using one BIG lock for the whole queue. Feel free to move the declaration of kMaxQueueSize=1000 to the file queues.h from test.cc. You will need this to declare a fixed size array of mutexes. For simplicity, you may assume that the number of producer or consumer threads are always in powers of two, and that the total number of operations is always divisible exactly by the number of producers/consumers.

A)

Provide a couple of sentences in the final write-up describing how you convinced yourself your algorithm was implemented correctly.

Problem 2: Use Lock-free programming

For this problem, you will use lock-free techniques to implement a circular fixed-sized queue. As lock-free programming can be rather tricky, you will do this in stages.

The first part is (almost) straightforward. You can start by creating a regular circular queue, and then identifying the variables shared between the producer and the consumer threads. These need to be atomically updated in a way which guarantees consistent updates of the queue. Finally, develop tests to check that there are no data races even in the absence of locks. Verifying that an implementation works is usually more difficult than creating the implementation in the first place. You could try out extreme conditions - making the queue size as 1 for instance would help you locate tricky conditions more easily.

1. Develop a lock-free circular queue implementation which works with a single producer and a single consumer (LF1).

2. Develop a lock-free circular queue implementation which works with a single producer and multiple consumers (LF2).
3. Develop a lock-free circular queue implementation which works with multiple producers and a single consumer (LF3).

As you are working, try to understand how the constraints on the queue - fixed size and circular, help your case for the lock-free implementations. Also, if you do not manage to get these working correctly, an analysis of what error is occurring (for instance the ABA problem) will get you partial credit.

For simplicity, you may assume that the number of producer or consumer threads are always in powers of two, and that the total number of operations is always divisible exactly by the number of producers/consumers.

A)

Describe your implementations for each of the above queues.

B)

Describe at least 2 notable examples of tricky scenarios you encountered while writing the lock-free implementations.

C)

Provide a couple of sentences in the final writeup describing how you convinced yourself your algorithm was implemented correctly. Describe how the queue constraints simplify your program.

Problem 3: Analysis of Fixed Size Queues

For this analysis, set the number of operations as 10000000, queue size as 1000, and run each experiment on ale nodes for 5 iterations. For each graph, make sure the axes are named correctly, have title, captions and legends as needed, and in general, are self-sufficient. Use only std::memory_order_seq_cst for this portion.

1. Compare the throughput of all 5 queue implementations (MX - provided, FG, LF1, LF2, LF3) for a single producer, single consumer scenario.
2. Compare the throughput of MX, FG, LF2 for a single producer, multiple consumer case with number of consumer threads = [1, 2, 4, 8, 16]
3. Compare the throughput of MX, FG, LF3 for a multiple producer, single consumer case with number of producer threads = [1, 2, 4, 8, 16]

Repeat this on godel (upto 256 threads) and cavium (upto 48 cores).

Problem 4: Analysis with different memory models

For this analysis, set the number of operations as 10000000, queue size as 1000, and run each experiment on ale, cavium and godel for 5 iterations. For each graph, make sure the axes are named correctly, have title, captions and legends as needed, and in general, are self-sufficient.

1. Compare the throughput of all 3 lockfree queue implementations (LF1, LF2, LF3) for a single producer, single consumer scenario using both std::memory_order_seq_cst and std::memory_order_acquire/release.

Tips and Tricks

You will be using the Intel compiler from Intel ComposerXE 2015. It is installed in /s/intelcompilers-2015. Before attempting to compile anything with Cilk, be sure to source the following files like below:

- `source /s/intelcompilers-2015/bin/iccvars.sh intel64`
- **Start early. Perhaps not as early as HW4.**
- Run multiple experiments to reduce the chances that your performance numbers are wrong due to contention.
- Run `top` on ale-01/02 and see if anyone else in your class is currently running experiments. If so, wait a few minutes. This isn't needed for godel or cavium ;).

What to Hand In

Please turn this homework in on **paper** at the beginning of lecture. You must include:

- Answers to questions in Problems 1 and 2.
- The plots as described in Problem 3, 4 including a brief explanation of the observed trends.
- Mail in the source code for all the 4 fixed queue implementations developed by you.

Important: Include your name on EVERY page.

Page last modified on October 16, 2016, at 05:18 PM, visited times