CS 758: Programming Multicore Processors (Fall 2016)

Homework 4

C++ Threads: Parallel B-Tree

You should do this assignment alone. No late assignments.

Filelist for the assignment:

- Template Files
- C++ and C documentation
- C++ thread support
- C++ atomics

You will perform this assignment on dual socket quad-core two-way threaded (16 threads total) Intel Nehalem processors (`ale-02.cs.wisc.edu` through `ale-07.cs.wisc.edu`) and godel.

As always, if you wish to learn more about ale, examine the `/proc/cpuinfo` file.

Note that for multithreaded x86 cores it is important to insert the `PAUSE` instruction in spin-wait loops. This can significantly improve performance of spin-wait loops because it avoids a memory-order violation on loop exit, and allows the processor to throttle the execution of the spinning thread, leaving more resources for the other thread executing on the core. (The PAUSE instruction is a no-op on non-multithreaded implementations.) There is a macro provided with your assignment code that will compile correctly on all architectures you will be using.

Purpose

The purpose of this assignment is to give you some experience writing a non-trivial shared-memory concurrent data structure. Additionally, you will implement a rudimentary version of high-level concurrency control, and will explore high-level and low-level synchronization of concurrent data structures.

Programming Environment: C++ Threads

We will use the C++11 thread support in this assignment primarily because of its easy portability. The mechanics of thread creation and the orchestration of thread movement has been done for you.

Programming Task: Concurrent Binary Tree

Trees of various flavors are common -- perhaps the simplest is the unbalanced binary tree, which will be the subject of this assignment. In general, you will discover that binary trees are not the most scalable of data structures, but they are conceptually simple.

You are given an implementation of a concurrent binary tree, which has been (somewhat) optimized for a small number of threads. Your task is to improve the tree's scalability and throughput on highly-concurrent machines.

The tree in question is used to implement a map abstraction. Maps are used to associate a unique key with a value, which for simplicity is also unique. Maps can be implemented with a variety of data structures (hash tables are common), though your implementation must use a binary tree. The tree supports the following primitive operations in *parallel*:

```
    int  Lookup( int key );
    void Remove( int key );
    void Set( int key, int value );
```

The above functions may be called in parallel by any number of threads, up to the number of thread specified in the constructor of the tree. The tree's constructor and destructor, as well as the `print()` function, need not be thread-safe.

Serializability Rules

Providing primitive atomic operations is often necessary but is not always a sufficient interface for writing parallel codes. In particular, it is desireable for *semantically-related sequences of operations* to appear atomic, even in the presence of concurrency. We would like these sequences (hereafter transactions on the data structure) to be **serializeable**, that is, to appear as though the transactions executed in a single global order. (Consult your reading list for more on serializability)

Hence, in addition to providing the primitives above, the tree sports a *transactional interface*:

```
    void InitiateTransaction();
    void CommitTransaction();
    void TransactionAborted();

    bool TransactionalLookup( int &data, int key );
    bool TransactionalRemove( int key );
    bool TransactionalSet( int key, int value );
```

In particular, a thread begins a transaction by calling `InitiateTransaction()`. Subsequent calls to the transactional access functions (`TransactionalLookup`, `TransactionalRemove`, and `TransactionalSet`) return a boolean value, indicating whether the current transaction must *abort* -- undo its changes and restart -- because of a violation of serializability. The transaction is ended by a call to `CommitTransaction()`. If the transaction is aborted, `TransactionAborted()` is called by the thread once the thread has undone all of its changes to the tree. The precise rules for implementing these functions are discussed below.

The transactional and non-transactional interfaces are **not** intended to be used at the same time -- the tree need not respect transactional semantics if transactional and non-transactional interfaces are used concurrently. However, all non-transactional accesses *must remain atomic in all cases*.

Serializability v. Atomic Access Examples

Consider these two transactions that wish to operate on our example binary tree (abort code not shown for simplicity):

Thread 1	**Thread 2**

`....tree.InitiateTransaction();`
`....TransactionalLookup( x, 7 );`
`....if( x > 5 ) {`
`........TransactionalRemove(7);`
`....} \\ ....tree.CommitTransaction();`
	`....tree.InitiateTransaction();`
	`....TransactionalLookup( y, 8 );`
	`....if( y > 5 ) {`
	`........TransactionalSet(8,5);`
	`........TransactionalSet(9,y-5);`
	`....}`
	`....tree.CommitTransaction();`

Notice that these two transactions seem to be composable -- their read and write sets don't overlap at all (Thread 1 looks at 7 and might remove it, Thread 2 looks at 8 and might update 8 and create 9). But when the transactions are applied to the tree below, we see that there actually *is a dire need* to synchronize these accesses (assume the implementation of `Remove()` seeks to avoid rotations by promoting leaf children over non-leaf children, promotes the right child in the event of a tie, and doesn't enforce balance).

```
          <5,5>
        /      \
    <3,3>       <7,7>
    /         /      \
<2,2>      <6,6>    <8,8>
```

Depending on the ordering of the transactions, both of the following trees are valid, and either could result from a serializeable execution:

```
          <5,5>
        /      \
    <3,3>       <8,5>
    /         /      \
<2,2>      <6,6>    <9,3>
```

```
          <5,5>
        /      \
    <3,3>       <6,6>
    /                \
<2,2>               <8,5>
                        \
                        <9,3>
```

This example illustrates the high-level notion of transaction serializability (both trees represent the same map, but are structurally different). The next example illustrates that *individual atomic operations alone are not sufficient to guarantee transactional serializability*. Consider:

Thread 1
```
....int sum = 0;
....tree.InitiateTransaction();
....for(int i=0;i<N;i++) {
........TransactionalLookup( x, i );
........sum += x;
....}
....tree.CommitTransaction();
```

Thread 2
```
....tree.InitiateTransaction();
....TransactionalLookup( y, 8 );
....if( y > 5 ) {
........TransactionalSet(8,5);
........TransactionalSet(9,y-5);
....}
....tree.CommitTransaction();
```

Thread 1 is trying to find the sum of all the data values in the tree, while Thread 2 is attempting the same transaction as the earlier example. Suppose initially the tree looks as it did for the previous example:

```
          <5,5>
        /      \
    <3,3>       <7,7>
    /         /      \
<2,2>      <6,6>    <8,8>
```

We assume that each `TransactionalX()` call is atomic. Hence, the dynamic sequences of TransactionalX() calls from each thread are (assume N is 10):

Thread 1	Thread 2
`Lookup( x, 0 );`	
`Lookup( x, 1 );`	
`Lookup( x, 2 );`	
`Lookup( x, 3 );`	
`Lookup( x, 4 );`	`Lookup( y, 8 );`
`Lookup( x, 5 );`	`Set( 8, 5 );`
`Lookup( x, 6 );`	`Set( 9, y-5 );`
`Lookup( x, 7 );`	
`Lookup( x, 8 );`	
`Lookup( x, 9 );`	

The course of dynamic execution could interleave these accesses in any way that orders their single-threaded executions. Suppose we interleave the operations such that Thread 2's Set operations occur near the end of Thread 1's scan:

Thread 1	***Order at Tree***	**Thread 2**
`Lookup( x, 0 );`	`Lookup( x, 0 );`	
`Lookup( x, 1 );`	`Lookup( x, 1 );`	
`Lookup( x, 2 );`	`Lookup( x, 2 );`	
`Lookup( x, 3 );`	`Lookup( x, 3 );`	
`Lookup( x, 4 );`	`Lookup( x, 4 );`	
`Lookup( x, 5 );`	`Lookup( x, 5 );`	
`Lookup( x, 6 );`	`Lookup( x, 6 );`	
`Lookup( x, 7 );`	`Lookup( x, 7 );`	
	`Lookup( y, 8 );`	`Lookup( y, 8 );`
	`Set( 8, 5 );`	`Set( 8, 5 );`
`Lookup( x, 8 );`	`Lookup( x, 8 );`	
`Lookup( x, 9 );`	`Lookup( x, 9 );`	
	`Set( 9, y-5 );`	`Set( 9, y-5 );`

A problem arises because Thread 1 observes Thread 2's change to key 8 but not Thread 2's addition of key 9. Hence, the sum that Thread 1 derives is not the sum of any valid tree.

Provided Code

The code provided is an impelementation of the concurrent tree in C++, along with a host of test programs, that compiles cleanly on all three architectures on which you will perform this assignment. The purpose of each file is briefly summarized below:

File	**Contents**
`Barrier.*`	Implements an object-oriented barrier using POSIX interfaces
`CTree.*`	Implements a concurrent binary tree -- you will heavily modify these files in this assignment.
`fatals.*`	Bails out of the program, displaying an error message.
`main.C`	Spawns threads according to the number of available processors, runs the tests in Tests.C according to preprocessor options.
`ProcMap.*`	Determines the physical processor identifiers for the system, and gives a generic interface to processor affinity.
`Stats.*`	Tracks some statistics about the execution.
`Tests.*`	Provides a set of tests for the concurrent tree, including a single-thread test, a parallel non-transactional torture test, a transactional torture test, and the throughput test. You may modify the single-threaded test and the torture tests as you wish for your own testing purposes. You may not modify the throughput test in any way.
`Transactions.*`	Implements some transactions for testing and throughput analysis. The transactions may not be modified (except for the purpose of debugging your code).

Your final solution must pass *all* provided torture tests. To allow for expediency, you need only report your statistics for the throughput test. Preprocessor flags in main.C can be toggled to run only a subset of the available tests at a time. These tests include:

- Serial tests: These tests exercise the tree using only a single thread.

- Parallel tests: These tests stress the tree using multiple threads, but using only atomic operations (no transactions).
- Transactional tests: This test runs many intensive concurrent transactions on the tree, and looks for serializability violations.
- Throughput test: This test runs realistic transactions on the tree. You will report your transaction throughput from this test.

Transaction Types

This section is informative -- it details the types of transactions that are implemented in `Transactions.*`.

Name	**Behavior**
Update	Reads then writes a small number of random elements.
Conditional Add	If a given key does not exist in the map, the key is inserted with value 0.
Conditional Remove	If a given key exists in the map and has value 0, the key is removed.
Lookup	Small read-only query of several key/value pairs.
Scan	Large read-only query of many key/value pairs.
TortureX	"Torture" version of transaction X. Regular transactions include delays to emulate work within the transaction. The torture versions have this delay removed to place additional stress on the tree.

(Additional) Rules for Implementation

- You may add any public or private data members, or even whole classes, to `CTree.*`, but you may not modify existing interfaces.
- You may perform any additional tests on your code if you wish. Feel free to modify any files you require to add additional testing, but you must be able to run the original throughput test on your concurrent tree implementation.
- If you elect to abort a transaction that calls `TransactionalLookup()`, `TransactionalSet()`, or `TransactionalRemove()` (by returning true from any of those functions), the tree should be left unmodified. The code in `Transaction.C` will not undo the action that caused the abort. If you wish to change this semantic of the interface, you must re-code `Transaction.C` appropriately.
- The Remove family of operations must actually remove a node from the tree.
- `class ConcurrentTree` must be implemented as a binary tree.
- Degenerate cases of `Lookup()`, `Set()`, `Remove()`, and their transactional counterparts are not error cases. `Lookup()`s of non-existent keys should return `NOT_IN_TREE`. `Set()`s on non-existent keys should create a new node and insert it into the tree. `Remove()`s of non-existent keys should not change the tree, nor should they raise an error of any kind.

Problem 1: Improve Concurrency of Atomic Tree Operations

The code provided uses a single lock to protect the entire binary tree while performing `Lookup()`, `Set()`, and `Remove()` operations. Improve this locking scheme in any way you prefer, subject to the rules above. Your code should compile cleanly on all target architectures.

Remember: This level of synchronization is intended to protect the ***physical validity of the tree***.

Problem 1 is purposely open-ended. In general, work here will pay off when you work on Problem 2. The availability of scalable atomic operations will be of great value.

Problem 2: Improve Transactional Throughput

Next, you will leverage your concurrent operations from Problem 1 to create serializeable transactions. The provided code uses another global lock to provide transactional access -- a sufficient solution for 2-way hyperthreaded machines, but grossly inadequate for the agressively-threaded CMPs.

As with Problem 1, you are free to improve the design in anyway you prefer, subject to the above set of rules. Remember: This level of synchronization is intended to protect the semantic value of the map, ***not necessarily*** the physical structure of the tree.

Bear in mind that, while you must provide a correct solution to the transactional torture tests, you seek to optimize the throughput test. Again, this problem is open-ended -- challenge yourself to improve your throughput as much as you can.

NOTE: To reduce your burden on this assignment, your tree will not be tested with any transactions which require a remove. If you want to test your implementation with remove operations, modify Transactions.h line 22.

Problem 3: Description of Synchronization Strategies

Describe the approaches you used on Problems 1 and 2. Provide neatly-drawn pictures where appropriate to illustrate tree locking.

Problem 4: Evaluation

Once your implementation passes all provided tests, report the transaction throughput on an ale node. Report the throughput for 1, 2, 4, 8, and 16 threads.

Also report the transaction throughput on godel for 1, 2, 4, ... 256 threads.

Problem 5: Questions

1. With how many threads does your implementation perform the best? Why do you suppose this is the case?
2. Compared to your other assignments on Ale, how does your implementation scale? Why do you suppose this is the case?
3. What, if any, additional testing did you use to debug your implementation?
4. Which was more difficult: Problem 1 (ensuring atomicity of primitive operations) or Problem 2 (ensuring serializability of transactions)?
5. Of the transactions implemented in `Transaction.C`, which was the most troublesome in Problem 2?
6. How C++ threads compare to the other programming models (OpenMP and Cilk) you've used in this class? Given the option, what would you have used on this assignment?

Tips and Tricks

- *Start early.* You never know when the TA will disappear the night before the assignment is due.
- *Make a plan* before doing Problems 1 or 2. An ounce of planning is worth a pound of coding. Draw trees.
- There's a handy `GetThreadID()` function packaged with the default implementation.
- *Start early.*
- Make use of the tests provided, and make your own if they're not sufficient for your needs.
- Did we mention *start early*?

What to Hand In

- Please turn this homework in *on paper* at the beginning of lecture.

- The source code for all files that differ from the provided source code.
- Your description from Problem 3.
- The table from Problem 4.
- Answers to questions in Problem 5.

Important: Include your name on EVERY page.

Page last modified on September 30, 2016, at 11:55 AM, visited times