COP 6616 Fall 2017

Data Structure Project

Note 1:

Please, submit your work via Webcourses.

Submissions by e-mail will not be accepted.

Midterm Report (Part 1) Due Date: Monday, October 30th by 11:59 PM

Final Report Due Date: Wednesday, November 29th by 11:59 PM

Late submissions are not accepted.

Note 2:

This assignment requires the use of C++ and POSIX threads in order to be compatible with the RSTM library.

Part 1: Concurrent Data Structure Implementation

In this assignment, you will work with your project team. Refer to the document "Algorithm Assignments" to find out the specific algorithm assigned to your team. Carefully and thoroughly study the provided materials related to your algorithm.

Implement the concurrent algorithm described in the paper assigned to you. Make sure to meet the described *progress* and *correctness* conditions of the concurrent data structure.

Refer to "Report Writing Guide" and compose a report describing your algorithms, implementation, design decisions, and performance evaluation. See the "Performance Evaluation" section of this document for the format of your results. For this part of the project, disregard aspects of the performance evaluation dealing with transactions.

Additionally, in your report address the following questions:

- What is the *progress guarantee* that your data structure provides? Include an informal proof of why the data structure meets the specified progress guarantee.
- What is the *correctness condition* that your data structure provides? Include an informal proof of why the data structure meets the specified correctness condition.
- What are the key *synchronization techniques* that allow this design to meet the described correctness and progress guarantees?

- What did you have to change from the design described in the research paper in *your implementation* of the algorithm?
- What are the advantages and disadvantages of this data structure compared to its alternatives? Are there any specific use cases where this design would be more beneficial than the-state-of-the-art alternative container?
- Can you think of ways to improve the design of the data structure? If so, please attempt them and compare your design and the original re-implementation of the algorithm.
- What are the biggest obstacles you encountered in your implementation?

Part 2: Transactional Data Structure Implementation Using STM

A *transaction* is a sequence of instructions executed by a single thread that is expected to appear to execute atomically. One way to ensure that concurrent transactions are thread-safe is to use critical sections of code guarded by a lock. However, locks only allow one thread to enter the critical section, hindering parallelism for transactions that would not have conflicted with each other. Software transactional memory (STM) provides a system for executing atomic sections of code, as an alternative to using locks for critical sections. STM monitors the memory locations that each thread accesses, maintaining the locations that the thread reads in a *read set*, and the locations that the thread writes to in a *write set*. STM detects memory conflicts when concurrent transactions have overlapping read/write sets. If a memory conflict is detected, only one transaction is allowed to commit its changes, and the other transactions must abort and restart.

In this part of the project, you will transform your concurrent data structure into a *transactional data structure*. Your completed transactional data structure should be able to execute transactions (consisting of multiple linearizable operations) to appear to execute atomically. To perform your transformation, use the open-source RSTM library, with code and documentation found here: https://code.google.com/archive/p/rstm/ . The original paper for the RSTM algorithm can be found here: https://www.cs.rochester.edu/u/scott/papers/2006_TR893_RSTM.pdf

A file named "main.cpp" is provided with this assignment. This file can be used as a starting point for working with RSTM. Instructions for running "main.cpp":

1. Follow the instructions for building RSTM in the README file.
2. Add "main.cpp" to the "bench" folder.
3. Edit the file "bench/CMakeLists.txt". Add the word "main" to the list of base benchmarks starting on line 22.
4. Compile RSTM by running "make". This will generate an executable file called "rstm_build/bench/mainSSB64".

5. Navigate to the “rstm_build/bench” folder and run “./mainSSB64”.

For the initial version of your transactional data structure, replace all instructions on shared memory with TM_READ, TM_WRITE, TM_ALLOC, and TM_FREE instructions. Compare the performance of your transactional data structure to your original concurrent data structure.

In your report, address the following questions:

- The size of a transaction is defined as the number of operations (e.g. push/pop) in the transaction. What performance differences do you observe between the concurrent data structure and the transactional data structure with transaction size of 1? Provide an explanation for the performance differences that you observe.

- Perform tests with higher transaction sizes. What occurs when you increase the transaction size?

Modify your transactional data structure to try to improve its performance. Compare the performance to your initial version of the transactional data structure.

In your report, address the following questions:

- What modifications did you implement? Did they improve the performance of the transactional data structure? Include your interpretation of the results.

- Provide an informal proof of the correctness of your modified implementation.

Part 3: Transactional Data Structure Implementation Using Transactional Boosting (EXTRA CREDIT)

Note: this part is optional and can increase your grade on the project to a maximum of 150%.

Transactional Boosting (TB) is a methodology for creating transactional data structures by using a system of abstract locks. Before performing an operation in a transaction, TB first obtains a lock or set of locks associated with that method call. The locks are arranged in such a way that two operations that commute are allowed to proceed concurrently, but two operations that do not commute will need to obtain the same lock. Two operations commute if executing them in any order will yield the same end result. The locks are usually based on keys, not physical nodes in the data structure. Once the lock(s) associated with the method call are acquired, then TB treats the underlying concurrent data structure as a black box to execute the operation. When a transaction completes its operations, it releases all of its locks. If a transaction fails to acquire a lock, it aborts. Then it must execute the inverses of the operations that have already been performed, so that the data structure returns to its original state. The original paper for TB can be found here: http://dl.acm.org/citation.cfm?id=1345237

Use TB to transform your concurrent data structure into a transactional data structure. You will need to arrange the locks in such a way that allows commutative operations to execute in parallel, but prevents non-commutative operations from running concurrently. Note that for some data structures, an operation will need to obtain more locks than just for the key in question. Also, for each operation, you will need to provide an inverse operation for TB to execute when a transaction aborts.

Evaluate the performance of your TB implementation to your STM implementations.

In your report, address the following questions:

- How did you arrange the locks? For each operation, which lock(s) are obtained before the operation is performed? Which operations are commutative, and which are non-commutative? Provide an informal proof of the correctness and efficiency of your design.
- Provide your interpretation of the performance differences that you observe between the TB implementation and the STM implementation.

Performance Evaluation

Write a test module using 1, 2, 4 and 8 threads that share the data structure. Have each thread execute a fixed number (e.g. 500,000) of operations on the shared data structure. Vary the distribution of the operations applied, e.g. in a Test Scenario 1 you can apply 50% push and 50% pop on a stack, and in a Test Scenario 2, you can apply 25% push and 75% pop on a stack, etc. Perform tests on 3 different distributions of operations. Also vary the transaction sizes, e.g. in a Test Scenario 1 you can perform transactions with 1 operation each, and in a Test Scenario 2, you can perform transactions with 2 operations each. Perform tests on 3 different transaction sizes.

Plot your results on 9 graphs: 3 for each distribution of operations and 3 for each transaction size. The 3 graphs with transactions of size 1 should display the results of the concurrent data structure. All of the graphs should display the results of your initial and modified transactional data structures. The x-axis should represent the number of threads used, and the y-axis should represent the total execution time in seconds needed to complete all operations. In your report provide a summary of your experimental evaluation. Make sure to specify the platform you used in your experiments and the details about your experimental setup. Additionally, include your interpretation of the observed results (very important).

Include a README file that provides thorough instructions on how to run your test program.

Grading policy:

Implementation of concurrent data structure: 35%

Documentation of concurrent data structure, including statements and proof of correctness, efficiency, and experimental evaluation: 25%

Implementation of transactional data structure using STM: 20%

Documentation of STM transactional data structure: 20%

(EXTRA CREDIT) Implementation of transactional data structure using Transactional Boosting: 30%

(EXTRA CREDIT) Documentation of TB transactional data structure: 20%

Additional Instructions:

Cheating in any form will not be tolerated.