CS3211 Assignment 1 Report

Data Structures

Our order book is structured as a map of various instruments to their corresponding list of orders, i.e., TSMap<Instrument, TSLinkedList> to achieve instrument-level concurrency.

To facilitate cancel order processing, we have another map of order IDs to some relevant information struct, i.e., TSMap<OrderId, OrderInfo>. Information includes client_id to enforce order cancellations are only done by the original client, and instrument to index directly into the order book.

The two concurrent data structures, i.e., TSMap and TSLinkedList, are elaborated below.

TSMap

TSMap is organized into a fixed arbitrary number of Buckets, where each Bucket contains a list of key-value pairs. While we can allocate a bucket to a key in various ways, e.g., first letter of the key, our implementation uses a hash function provided by the C++ Standard Library specialized to the key type in hope of an even distribution.

All operations provided by TSMap are bucket-concurrent, i.e., operations on different buckets can happen concurrently, and is achieved by simply locking the Bucket mutex at the start of each operation.

TSLinkedList

TSLinkedList is a fine-grained singly-linked linked list, with just a reference to the front Node. Each Node contains an Order, a mutex, and a pointer to the next Node. An empty TSLinkedList is defined to have a front dummy Node which points to a back dummy Node.

All operations provided by TSLinkedList are done via traversal by hand-over-hand locking, i.e., acquiring the current Node mutex while holding on to the previous Node mutex. Since the traversal is unidirectional from the front to the back, there is no possibility of deadlock.

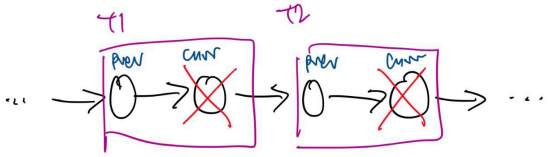
Concurrency

We aim to support phase-level concurrency.

Firstly, with our order book separated into different buckets for each instrument, orders from different instruments can execute concurrently in separate parts of the TS Map. Secondly, with traversal-based operations, all orders can be executed concurrently regardless of type. The second point is elaborated further in the next few paragraphs.

As we have two kinds of order, i.e., buy/sell order and cancel order, we have two corresponding TSLinkedList operations on the order book, namely **matching** and **deletion** respectively. Note that the matching operation constitutes *deletion* upon successful match and *insertion* otherwise.

The correctness of deletion can be justified as deletion is done only on the current Node. Recall that at each step during the traversal, we hold the locks of the previous and current Node. Hence, when assigning the next node of current Node to the next node of previous Node, we know that the next node of current Node will not be deleted by some thread in front of the current thread in the traversal.



The requirement of no matchable resting orders right before inserting an active order is enforced since we hold the last Node lock at the end of the traversal until the active order is inserted to the list, and no other threads can access. Matching/canceling newly added orders is also enabled due to the traversal nature of the matching and deletion operations.

Lastly, our implementation interleaves partial matches as every match operation traverses through the list once and returns. Such an implementation promotes concurrency indirectly.

Others

As mentioned previously, matching, i.e., a TSLinkedList operation, is done via traversal to determine the best price. The best price is only updated when the current price is strictly greater than the best price seen thus far, if any. Also, orders are only added at the back of the linked list. Hence, the matching process is compliant to the specified price-time priority rule.

To prevent time-of-check vs time-of-use, the timestamp of every “atomic” operation is recorded before releasing the lock when returning to be printed. The timestamps are encapsulated with the result of the operations. For example, MatchInfo returned by match() contains the relevant match information along with the timestamp-of-check.

Testing

For a sanity check, we ran our code with various sanitizers, e.g., ThreadSanitizer, AddressSanitizer, etc. and fixed the reported errors, restructuring code as needed to avoid data races.

To test that our engine can receive and execute orders from different clients concurrently, we wrote separate test files with varying numbers of threads submitting buy and sell orders interleaved with each other. To efficiently run all test cases in one go, we wrote a script, i.e., run_tests.sh to run all the test files in order with output.

We also stress test the engine’s performance with many clients, writing a script that generates a test file with a sequence of commands from a specified number of clients (we used 100 as an arbitrary number of threads to test), and 100 different instruments.