CS3211 Assignment 1 Report

Tan De Shao (A0218351L)
Alvin Tay Ming Hwee (A0218390E)

Description of data structures used:

The program uses various classes and data structures to ensure efficient and safe concurrent processing. For example, an order class is created to capture the order information(order_id, price, etc.). To allow multiple threads to read and write from a data structure concurrently, an atomic_map class is implemented using fine-grained locking with a **shared::mutex** (allows **concurrent reads** and **exclusive writes**). The underlying data structure of the atomic_map class is implemented using **std::vector<std::list<std::pair<K, V>>>**. Each bucket in the vector has its own mutex, which needs to be acquired before accessing the bucket. To calculate the bucket, each K is hashed and % to the size of the vector. Currently, the size of the vector is 1000 and can be increased if needed. i.e. this means that there are 1000 buckets, each having its own mutex. This significantly reduces the chances of **collision** when there are multiple read/writes to the vector, which allows for a finer-locking **i**mplementation and ultimately, greater concurrency. The orderbook object is shared by multiple threads and it supports public operations such as match_order and delete_order. These operations satisfy the requirements specified in the project document. Specifically, the match_order operation is designed to add an active order to the orderbook if there are no matchable resting orders, and this is achieved by calling the add_order method, which is a private method only used within the match_order method. The orderbook is implemented using two main data structures, called sell_book and buy_book, which store resting orders. The data structures are implemented using std::set, which is an ordered set allowing the programmer to specify a comparator. To facilitate efficient matching, the resting orders in sell_book and buy_book are ordered based on their best price, with the first element being the best-priced order. To enable fine-grain locking, the resting orders are split up based on their instruments, resulting in the orderbook having two attributes:
atomic_map<std::string,std::set<Order,OrderComparator>>sell_book and
atomic_map<std::string, std::set<Order, OrderComparator>> buy_book.
note*: there is another DS called atomic_map_mutex, but the implementation encompasses the same key ideas considered in the atomic_map.

Explanation of testing methodology:

The multithreaded program is tested and debugged using various methods. To access the scripts used for testing/debugging, please cd into the scripts folder first. A writer program is created to write tests to the scripts folder, enabling efficient testing and scalability of the tests. Tests were created to accommodate up to 40 threads and about 30 test cases are generated from this writer program (~/scripts/writer.cpp). To facilitate regression testing, a bash script named ~/scripts/scripts.sh was created to run all the tests in the scripts folder and provide a concise summary of passed and failed tests and their corresponding errors. Each test file in the scripts folder, such as 1.in, 2.in, etc., has a comment on the first line describing the scenario being tested, which helps us track tested scenarios and identify new ones. The program is compiled with thread and address sanitizers to detect memory errors, such as memory leaks and use-after-free, and to identify multi-threaded issues such as deadlocks and data races. For debugging, the **SyncCerr** object and **getCurrentTimestamp** method are useful tools that help us identify bugs in our program efficiently. However, having too many output statements on the console can be chaotic, hence, during the process of debugging, we would save the output onto ~/scripts/test.out and use the grep command in the shell to filter out useful lines that would provide us with meaningful information.

Synchronization primitives used:

```
    atomic_map_mutex buy_book_mutex;
    atomic_map_mutex sell_book_mutex;
```

```
    atomic_map_mutex buy_book_phase_mutex;
    atomic_map_mutex sell_book_phase_mutex;
```

The key is the instrument and the value is the mutex. Each of these mutex locks its respective (book + instrument), i.e. it can specifically lock the buy book of a GOOG instrument.

buy_book_phase_mutex and *sell_book_phase_mutex* are used at the top of match_order to lock match_order so that for each book + instrument, only one thread can run, i.e., for each instrument, one buy and one sell order can run concurrently.

buy_book_mutex and *sell_book_mutex* are used within match_order and delete_order to prevent data races when the book of a particular instrument is being mutated.

lock_guard, unique_lock and scoped_lock are used with these mutexes. The usage of the synchronization primitives will be elaborated below.

Explanation of level of concurrency (Phase-level concurrency second kind):
Our implementation allows orders for different instruments of opposing sides to execute concurrently, i.e. one buy and one sell.

match_order

```
void OrderBook::match_order(...) {
...
std::lock_guard<std::mutex> phase_lock((is_sell ? buy_book_phase_mutex.get(instrument) :
sell_book_phase_mutex.get(instrument)));
```

Close to the top of the match_order function, the thread will acquire the **(book + instrument) lock**, which ensures that for each (book + instrument), only one buy and one sell can run concurrently from this point on. This is the only place where buy_book_phase_mutex and sell_book_phase_mutex is used.

```
while (new_order.getCount() != 0) {
  std::unique_lock<std::mutex> opp_lock((is_sell ? buy_book_mutex.get(instrument) :
sell_book_mutex.get(instrument)));
  if (orders.empty()){
    opp_lock.unlock();
    std::scoped_lock scp_lock {sell_book_mutex.get(instrument), buy_book_mutex.get(instrument)};
    //Similar to CAS pattern
    if (!orders.empty()) {
      continue;
    }
    add_order(...);
    return;
  }
```

opp_lock is acquired before the (book + instrument) is read. E.g. for buy order it will lock the (sell book + instrument). If the book is empty, it will release opp_lock and acquire both (sell_book + instrument) and (buy_book + instrument) lock. After acquiring **scp_lock**, similar to the CAS pattern, if the book is not empty, it will match again. Else, the order is added to the book.

```
  auto it = orders.begin();
  Order other_order = *it;

  if (
     (new_order.getIsSell() && (other_order.getPrice() < new_order.getPrice())) ||
     (!new_order.getIsSell() && (new_order.getPrice() < other_order.getPrice()))) {
      opp_lock.unlock();
      std::scoped_lock scp_lock{sell_book_mutex.get(instrument),buy_book_mutex.get(instrument)};

      //Similar to CAS pattern
      if (orders.empty() || other_order.getOrderId() != orders.begin()->getOrderId() ||
other_order.getCount() != orders.begin()->getCount()) {
        continue;
      }
      add_order(...);
      return;
  }
```

Note that **opp_lock** is still being acquired here. The mutexes logic for this portion is similar to above, following the CAS pattern and taking into consideration the ABA problem in the second if condition.

```
    ...
    orders.erase(it);
    if (other_order.getCount() != 0) {
      orders.insert(other_order);
    }
    opp_lock.unlock();
    ...
```

After we are done with mutating the book, we release **opp_lock** to allow fine-grain locking.

delete_order

```
void OrderBook::delete_order(...) {
    ...
    std::lock_guard<std::mutex> lock((is_sell ? sell_book_mutex.get(instrument) :
buy_book_mutex.get(instrument)));
    auto is_erased = is_sell ? sell_book.get(instrument).erase(o) : buy_book.get(instrument).erase(o);
    ...
```

For delete_order, the thread acquires the **(book + instrument) lock** to mutate the book that contains the order.