**C++ Matching Engine**

**Design principles**

The project consists of two main components – an order generator class which constructs simulated orders based on either an input file or randomized order details to forward to a matching engine class.

Each component is run in its own dedicated thread and orders are communicated via an intermediary queue protected by locks/condition variables for consistency.

**Order Generator**

The order generator takes predetermined orders from a CSV format input file. Alternatively it will randomly create orders with different price/quantity, side, GFD, IOC and ISIN values.

**Matching Engine**

Once the Order Generator has signalled that there are orders to retrieve from the buffer the Matching Engine will obtain the lock and pull out the new orders.

These are then sent to a function Add\_Order\_to\_Book which models adding the order to either the BUY or SELL limit order book or the BUY or SELL marketable orders holding vector (since market orders cannot be added directly to a limit book).

The limits orderbook is a vector that is ordered so that, for bids, the first element is the lowest bid and the last element is the best (highest) bid. There is a counter currentBidsDepth that points to the best bid for quick access. Since in a real market most operations are inserts near the best bid this minimises the performance penalty of order inserts. Of course a pathological case occurs with bids inserted far from the best bid but this is unlikely in most real world cases. The sells vector operates in a similar manner. As a consequence matching algos match from right to left of the vector as they sweep down the book.

Market orders are held in a simple vector with earliest arrivals at the front and newer arrvials pushed to the back – thus they are always in order. Once a market order at the front is matched against the element must be erased which is a linear operation in vector size but can be optimised.

Since market orders are maintained in a vector, when a new limit order is detected a decision must be made whether to insert into the limit order book and possibly match – or to match with an existing market order. This is a key design choice since it is possible to deprioritise certain types of orders depending on the percentage of market to limit orders inbound at any time.

The decision was made to make this matching choice “atomically” as the inbound order hits, similar to adding the limit order to the limit book (thus interacting first with limit orders) and then model a prioritisation algo for interacting with market orders. In terms of performance and customisability perhaps more work could be done to understand the tradeoffs here under different order profiles.

As such the Add\_Order\_to\_Book() function sends the order to the checkForMatch() function firstly (unless a flag “checked” stating this has already been checked is set).

Arrival time is the differentiator for two limit orders of the same price and side. However, should a limit order in the book take priority over a newer or older market order – there are arguments for and against this. Some modelling of fairness is required but probably too complex to achieve in time set.

The logic as to where to match first is complex. Limit orders can match with limit or market orders, but market orders can only match with limit orders. It is simpler where we can match an incoming order with existing liquidity at the top of either market/limits books but where we potentially need to match against multiple orders of different sizes from different books the logic becomes very complex and has not been implemented.

A guaranteed percentage of market orders to be executed within a certain time window was considered but time was not available to implement.

There is still much to complete to make this a fully fledged matching engine but I hope there is enough to see the framework.

**Concurrency**

A large amount of time was dedicated to making this program run using modern C++20 lockfree atomics. A solution was nearly reached but there were odd problems with atomic waits and notifies that were most likely caused by program reordering effects from the compiler which would need memory fences to address. This became too complex to fix in the given time.

As such regular mutexes, locks and condition variables were used instead in a standard fashion.

**Matching and Allocation**

Where a large market or limit order may potentially match with multiple smaller orders already in the book functionality to “sweep and match” is provided. The functionality is similar for both order types but uses a different temporary allocation buffer to hold partial execs within the sweep.

Limits – Allocation\_Vec\_Buys and Allocation\_Vec\_Sells

Markets -- Market\_Orders\_Allocation\_Vec\_Buys and Market\_Orders\_Allocation\_Vec\_Sells

Given more time perhaps these vectors but be merged to save on space. Or perhaps keeping separate would be better with additional concurrency happening. More testing is required. The ability to customise size to system wide settings is available.

The function Sweep\_And\_Match\_Market\_Orders is used for limit orders to match potentially many market orders. If the last order touched is only a partial exec then the residual amount left is updated in the market order. If the limit order exhausts all existing market orders then the residual amount is added to the limit book as a new limit order.

The equivalent functionality for sweeping of limit orders already in the limit book is contained within the checkForMatch function in a similar way.

**IOC**

IOC orders are supported and if no match is found immediately they will be deleted within the Add\_Order\_to\_Book() function without being added to either Limit or Market book.

**GFD**

A function has been provided Purge\_GFD\_Orders() that will run at midnight and delete all orders with a GFD setting of true using the STL remove\_if algorithm with a predicate.

**Top of Book and Depth display**

The function Publish\_Top\_Of\_Book() will publish these items – there is an atomic bool that must be obtained first to prevent multiple functions accessing the Orderbooks at the same time. Accessing top of book and depth are both O(1) constant time accesses.

**Logging**

Setup to use Boost Log v2 which has support for async logging so as not to impact app performance.

**Testing**

In addition to the scenarios in the provided input file, around 10 unit tests have been setup to model testing of various order patterns that were observed to exercise different logic/components in the system.

Some of these represented key bugs that needed to be addressed to guarantee correct functioning of the sweeping and matching algorithms.

There is logging to the console for all test results along with a pre/post view of the Bids/Asks order book to clearly demonstrate the effect of the scenarios.

Order cancels and flushing of the order book have been tested and verified to work correctly also.

**Environment**

Built with:

Microsoft Visual Studio Community 2019

Version 16.9.3

ISO C++20 Standard (/std:c++20)

Using BOOST Log from recent BOOST v1.75.0 – this is supposed to be a header only file but actually may require linking a of a boost library.

If you don’t have this just change

BOOST\_LOG\_TRIVIAL(info) to std::cout