15-418 Final Project: Limit Order Book

By Irene Liu and Lillian Yu

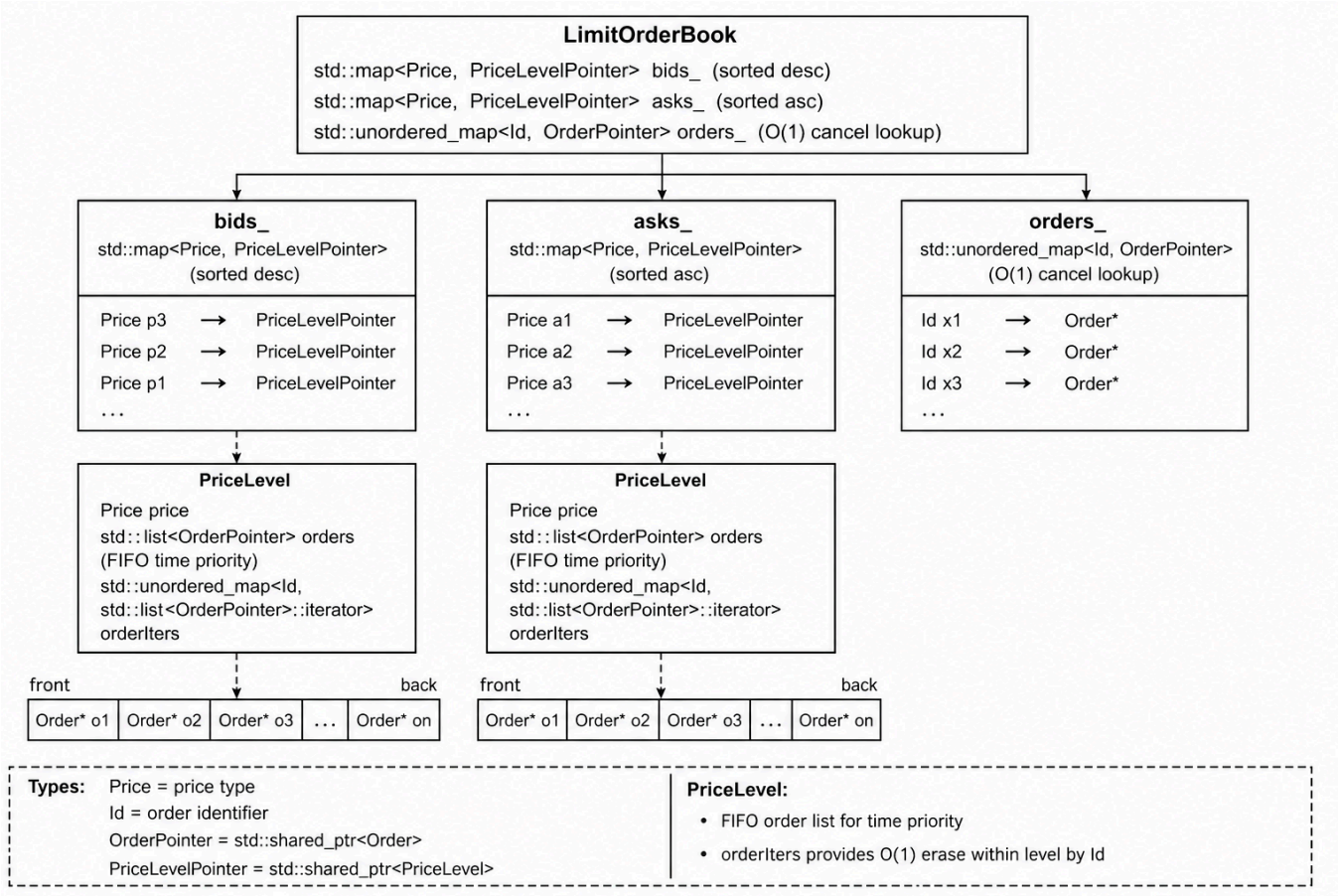
Summary

We parallelized a Limit Order Book with different synchronization strategies (coarse-grained, fine-grained, and batch locking). Our best implementation, coarse-grained locking, achieved 5.24x speedup on an 8-core machine while maintaining time-priority matching correctness.

Background

Limit Order Book. A limit order book (LOB) is the core data structure used by financial exchanges to match buy and sell orders. Orders are processed according to price-time priority, meaning that better-priced orders are matched first, and among equal prices, earlier orders take precedence.

Data Structures. To maintain this priority, the system typically maintains two ordered structures: a Bid book (buyers, sorted descending by price) and an Ask book (sellers, sorted ascending by price). Each of these structures is often implemented as a sparse list of “price levels” where each price level contains a FIFO (to account for time precedence) linked list of individual orders.



Key Operations. The LOB will also have three main operations: inserting new orders, cancelling existing orders, and an internal operation to fill orders if there are any bids and asks with matching prices.

Inputs and Outputs. The order book's inputs include messages requesting to place limit orders, market orders, or modify/cancel existing orders. Limit orders have a price and a quantity request, meaning they can either match immediately, if a price match is present in the LOB, or rest on the LOB. On the other hand, market orders only contain a quantity request and are instantly filled with whatever the best prices are on the other side of the LOB. The output of the algorithm, therefore, is a continuous response to the slew of messages that match orders on the LOB. Specifically, we have the following:

- Inputs: Stream of 100k to 5M messages (order type, price, quantity ticker)
- Outputs: Trade records (buyer ID, seller ID, quantity, price) for matched orders

We also consider different realistic workloads and their distributions:

Workload	**Limit %**	**Market %**	**Cancel %**	**Scenario**
Balanced	60%	20%	20%	General
Crossing	30%	60%	10%	High-activity scenario
Resting	70%	10%	20%	Liquidity-provision scenario
Skewed	60%	20%	20%	One single hot ticker

Parallelism Benefit. Order books are used for exchanges that process millions of orders per second. The high volume of data, as well as its time sensitivity, makes parallelizing LOBs to optimize performance a very useful task. Moreover, there are many parts of this system that have opportunities for parallelism. For example, operations on different tickers will not affect one another, since they are on different order books, making them very data-parallel. Additionally, parsing the messages associated with each order (placement or cancellation) could be pipelined to hide latency. This leads to the following heuristics usually in place as our computational limiting/expensive factors in fulfilling LOB operations:

1. Snapshot check: read the bid and ask maps to see if the order is crossing.
2. Traversal: walk matching prices from best to worst.
3. For each level: acquire lock, extract quantity, update level state, release lock.
4. Update global order index.

Dependencies. Within each ticker, the matching engine introduces strong dependencies that limit parallelism. The main dependency we must navigate is that matching bids and asks depends on a strict price-time priority of orders that must be maintained across a globally shared state, a rule legally required by exchanges in practice. As such, it is pertinent that order

inserts and cancellations lead to safe and synchronized access of shared memory structures. This leads to inherent sequential components, making parallelism more difficult. For example, a single, but large, market order can lead to many matches across multiple price levels, leading to a long read-modify-write chain that alters the global state. Note that due to such dependencies, the order matching workload is not amenable to SIMD execution since there is no independent iteration that can be partitioned.

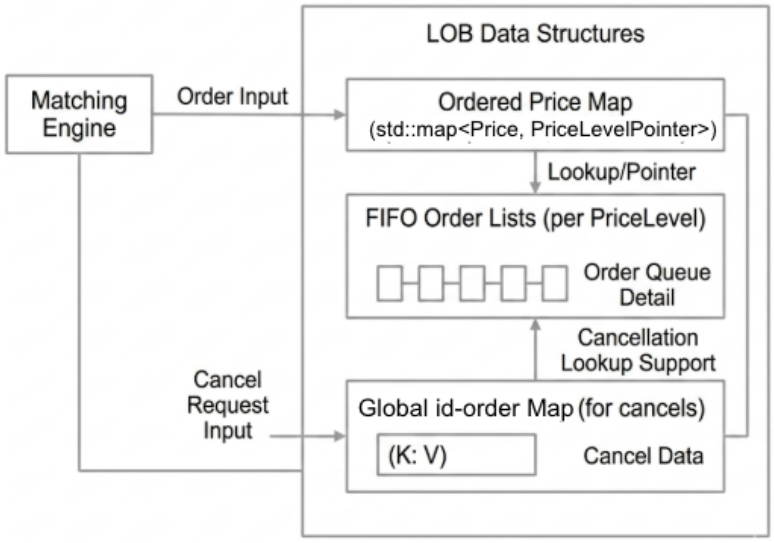
Approach

We used C++17 as our language of choice, leveraging the pthread API to write parallel code. Our program was then run on the 3.0 GHz Intel Core i7-9700 processors with 8 cores, via the GHC lab machines.

Serial

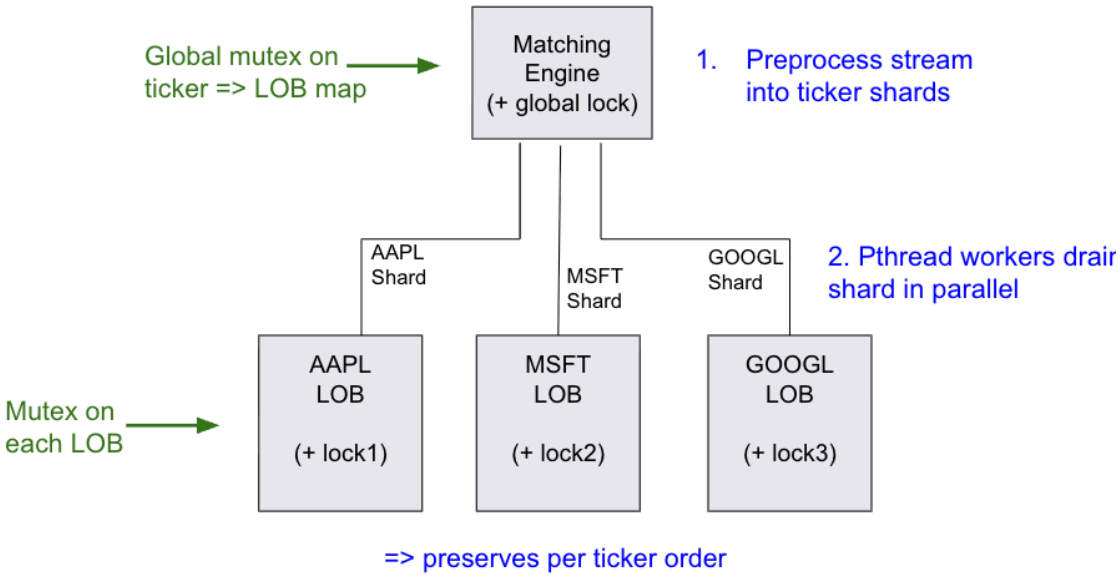
Our serial LOB was primarily for two purposes: correctness validation, since all parallel implementations must produce identical trades for the same message stream, and a latency baseline. The serial engine processes messages sequentially, maintaining the order book state with:

1. Bid/Ask maps: `*map<Price, PriceLevelPointer>*` for O(log P) price level lookup
2. Price levels: `*deque<OrderPointer>*` to maintain FIFO time-precedence within each price
3. Order index: `*map<OrderID, OrderPointer>*` for O(1) cancel operations.



Coarse-Grained

Overview. The main insight enabling our initial parallelism was recognizing that the LOB's data structure can naturally decompose along ticker boundaries and each ticker can maintain an independent bid and ask book with no shared state.



Shard Drainage. Our strategy leveraged data parallelism by partitioning the input message stream by ticker in a single preprocessing pass, creating independent shards for each active ticker. This shard contains all messages for that ticker in their original order, preserving the per-ticker message sequence required for correctness.

Parallel Mapping. During parallel execution, we spawn pthread worker threads up to the number of cores available (n = 1, 2, 4, 8). Each worker thread uses a shared `*std::atomic<std::size_t>*` index (fetch_add) to dynamically claim the next available ticker shard, operating on its own CoarseGrainedLimitOrderBook instance to drain the shard. So overall, one thread processes all messages for that ticker sequentially within the critical section, but multiple threads process different tickers' books in true parallel.

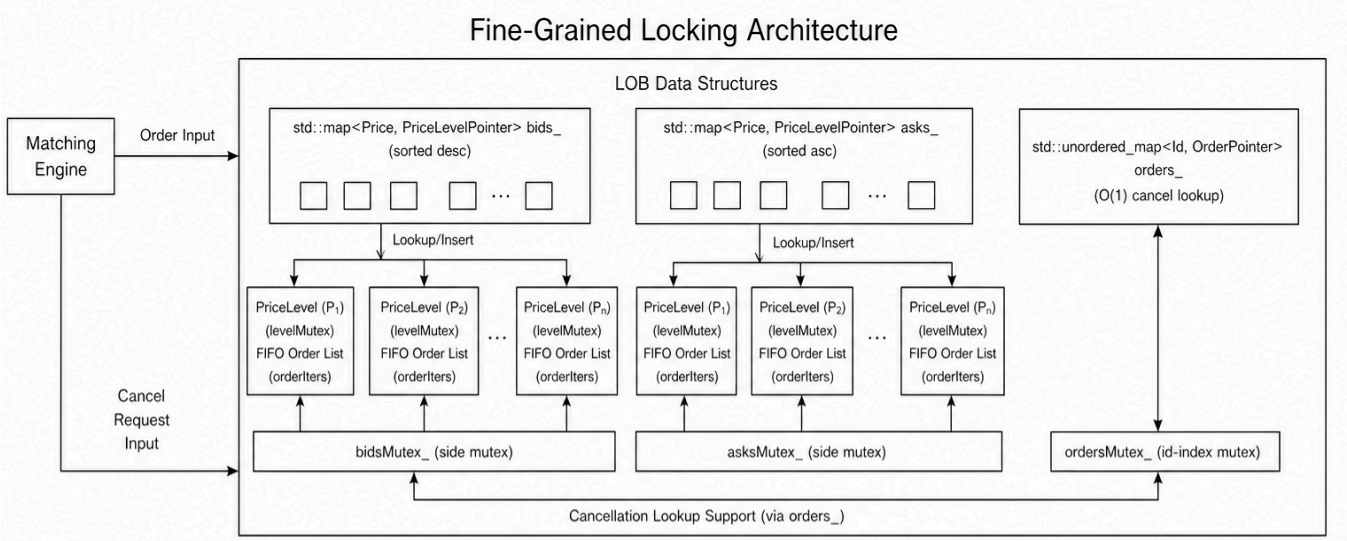
To maintain a synchronized state across threads, we utilize mutexes. A single mutex wraps each serial LOB book and is grabbed per operation within the LOB by the thread managing that LOB. Additionally, the matching engine contains a global `*booksMapMutex_*` mutex that protects the matching engine's map from ticker to that ticker's LOB. This mutex is grabbed once per shard, when a thread is taking over that ticker's shard (in drainShard).

Finally, to unify each shard's LOBs into a single `*std::vector<Trade>*` containing all executions across all tickers, we require a third lock `*mergeMutex*`. Each worker thread holds a pointer to this mutex, and attempts to grab it on completion, in order to merge its results into the global trades vector.

Optimizations. After our initial transition from serial to parallel mapping, we aimed to further optimize by reducing deep copies in memory Initially, to partition the workload for the workers, our algorithm copied full OrderMessage objects into per-ticker shards. However, we found that the overhead of allocating and copying heap-allocated ticker strings (`*std::string*`) for every single message severely dominated the wall time. We optimized this by changing the serial algorithm's partitioning phase to only store 8-byte `*std::size_t*` indices referencing the original message vector. Additionally, we further prevented deep copies by utilizing `*std::move*` and `*std::make_move_iterator*` to prevent moving full Order or Trade objects on and off of threads. All of this reduced our memory bottleneck, and improved speedup on the parallel machines.

Fine-Grained

Overview. Fine-grained locking attempts to maximize parallelism by reducing lock granularity from the entire order book (coarse-grained) to individual price levels. We hypothesized that concurrent operations on different price levels should not block each other, enabling multiple threads to operate on the same book simultaneously.

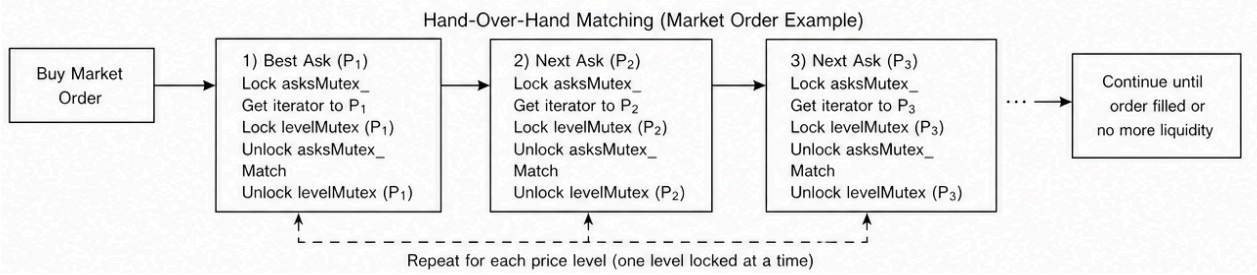


Per-level Locking. Our architecture maintains (1) a side-level mutex (one for bids, one for asks) to protect the map structure containing price levels, (2) a per-level mutex to protect each price level's order deque, and (3) a global orders mutex to protect the order index used for fast cancellations. The lock acquisition hierarchy proceeds from the side mutex to acquire or look up a price level, and then to the level mutex to insert or remove orders at that specific price, and then finally to the orders mutex to update the global order index.

For a non-crossing limit order insertion, the process acquires the side mutex to get or create the level pointer, releases it, then acquires the level mutex to insert the order into the deque, releases that, and finally acquires the orders mutex to update the global index. This results in exactly six lock operations per non-crossing order (acquire and release for each of the three locks), compared to just two lock operations in the coarse-grained approach. The theoretical

benefit of fine-grained locking is that while thread A inserts at price level 100, thread B can simultaneously insert at price level 101 without contention, each acquiring different level locks in parallel. However, in practice, the bid-ask spread (usually the top one to three price levels) processes 80% or more of all volume, meaning that most threads compete for locks on the same hot levels.

Hand-Over-Hand Matching. Hand-over-hand locking is a technique for traversing multiple protected resources sequentially while maintaining correctness and avoiding deadlock. In the context of market order matching, a market order must match against multiple price levels in priority order, from best price to worst. Rather than acquiring and releasing the side mutex multiple times during the traversal, hand-over-hand locking acquires the side mutex, retrieves the best price level pointer, releases the side mutex, then acquires that level's mutex, matches orders at that level, releases the level mutex, and repeats for the next level until the order is fully filled.



For each price level traversed during matching, our algorithm performs two lock operations: one side lock acquire-release pair and one level lock acquire-release pair. For example, for a market order that matches against 10 price levels, this results in 20 lock operations, plus 1 additional orders mutex operation to update the global index, for a total of 21 lock operations. In contrast, the coarse-grained approach acquires a single mutex at the start of the market order processing and releases it after all levels have been matched, requiring only 2 lock operations regardless of how many levels are involved.

False Sharing Optimization. False sharing occurs when two threads modify different variables that share the same cache line, which on Intel i7-9700 processors is 64 bytes. When one thread modifies data near the beginning of a cache line, and another thread modifies data near the end of the same line, the entire cache line must be invalidated and reloaded by the coherence protocol, even though the threads are accessing logically independent variables. In our fine-grained limit order book, we hypothesized that concurrent inserts to a price level's order deque could trigger false sharing, as multiple order pointers would be packed into the same cache line, and simultaneous insertions by different threads would cause the cache line to be repeatedly invalidated and reloaded.

To address this, we implemented a 64-byte alignment strategy where order pointers were padded to cache line boundaries, ensuring that each order pointer occupied its own cache line and eliminating false sharing between concurrent inserts. The implementation involved

modifying the deque insertion logic to insert padding null pointers when necessary, so that each actual order pointer started at a 64-byte boundary.

From the above key data structures, we can summarize the following operation complexity:

Operation	**Coarse-Grained**	**Fine-Grained**	**Bottleneck**
Insert limit (non-crossing)	O(log P)	O(log P + level lock)	Map lookup + level lock
Cancel order	O(log P)	O(log P + level lock)	Map lookup + level lock
Market order match	O(log P + M * L)	O(M * L * 3) locks	Hand-over-hand traverse

Where P = price levels, M = matches, L = average traversal locks

Batching

One observation we made is that non-crossing limit orders (i.e. orders that don't immediately match) are passive, as they go directly to the rest() functions without triggering matching logic. Since in the fine-grained implementation, each non-crossing order required three lock acquisitions (side lock, level lock, orders lock), we could alternatively consider grouping consecutive non-crossing orders and inserting them as a batch to reduce acquisitions from N to 1 per batch, aiming to improve performance.

A new BatchingMatchingEngine overrides the drainShard() method, where instead of processing messages one-by-one, we detect runs of consecutive non-crossing limit orders and accumulate them into a batch vector. When a crossing, market, or cancel order is encountered, the accumulated batch is inserted via a new batchRest() method, then active orders are processed normally.

Results

Performance was evaluated on a variety of workloads, specifically at 3 different order counts (100k, 500k, 5M), 3 different ticker counts (3, 8, 16), and 4 different thread configurations (1, 2, 4, and 8 threads). We also observed the impact of different workload distributions: balanced, resting-heavy, crossing-heavy, and ticker-skewed to analyze the parallelism strategy impact on different real-world scenarios.

The baseline is the sequential, single-threaded CPU implementation i.e. one thread processes all 500k orders against all tickers one at a time, with no parallelism or multi-threading overhead. This represents our no parallelism case.

Experimental Setup

Input Parameters. The benchmarks test a range of order counts and ticker counts to explore scaling behavior.

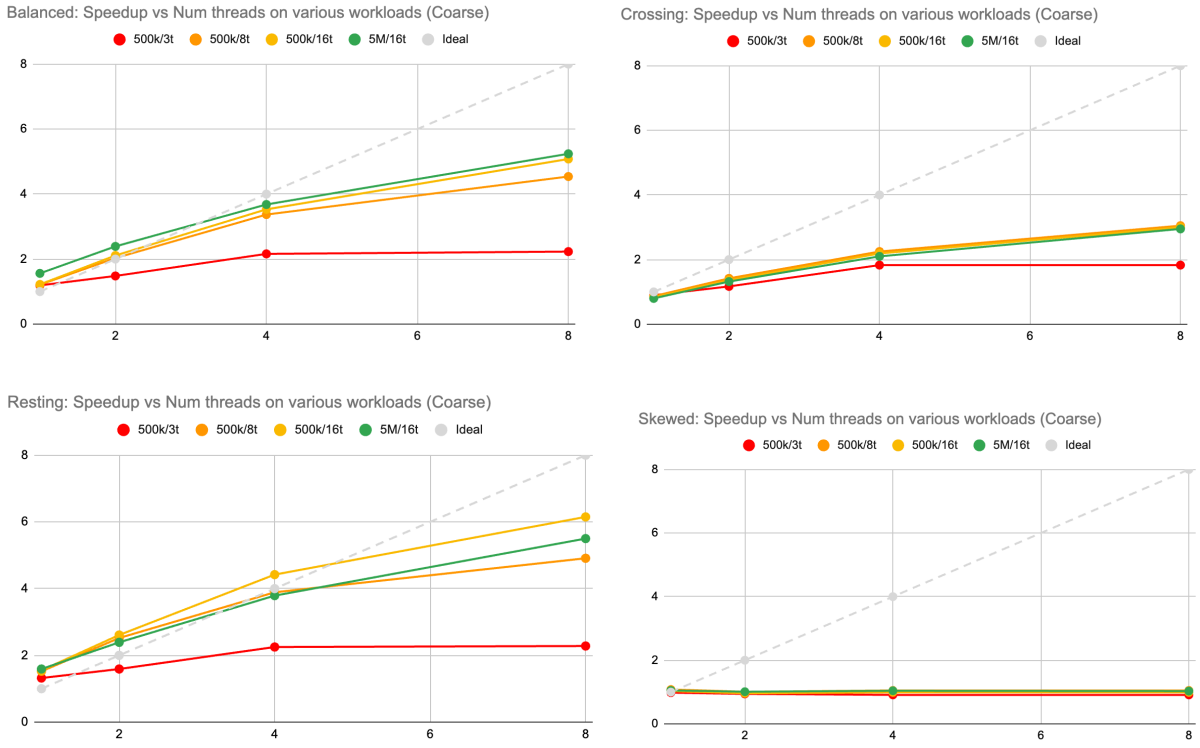
Configuration	Orders	Tickers	Context
500k/3t	500,000	3	Small ticker set, high contention per ticker
500k/8t	500,000	8	Medium ticker set, moderate contention
500k/16t	500,000	16	Larger ticker set, better parallelism potential
5M/16t	5,000,000	16	10x order volume, stress test parallelism

Order Generation. All orders are generated deterministically using a seeded PRNG to ensure reproducibility across runs. The generation process is split as follows:

1. Order Type Distribution (varies by workload): Balanced, Crossing, Resting, and Skewed (detailed in the background section).
2. Limit Order Properties:
    a. Price: uniformly random in [100, 200] (fixed range across all tickers).
    b. Quantity: uniformly random in [1, 100] shares.
    c. Side: 50% BUY, 50% SELL (alternating for reproducibility).
    d. Placement: each LIMIT order assigned randomly to one of the N tickers.
3. Market Order Properties:
    a. Quantity: uniformly random in [1, 100] shares.
    b. Side: 50% BUY, 50% SELL.
    c. Execution: immediately matches against existing limit orders at best available price.
    d. Ticker: same random assignment as limit orders.
4. Passive vs. Active Order Split:
    a. A LIMIT order that doesn't cross the spread is "passive" (goes to rest(), no matches).
    b. A MARKET order or LIMIT order that crosses the spread is "active" (triggers matches).
    c. For resting workload: ~70% of 500k orders are passive LIMIT, therefore ~350k orders bypass matching engine, testing book insertion overhead.
    d. For crossing workload: ~60% of 500k are MARKET orders, therefore ~300k orders trigger hand-over-hand matching across price levels.

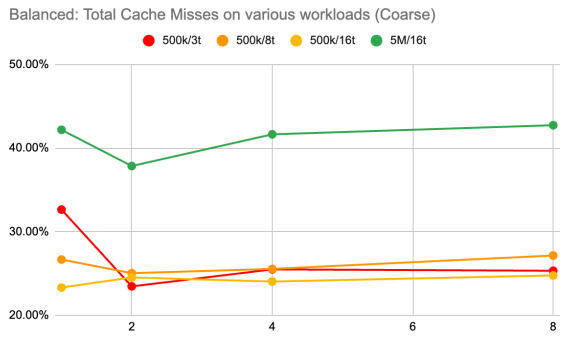
Coarse-Grained

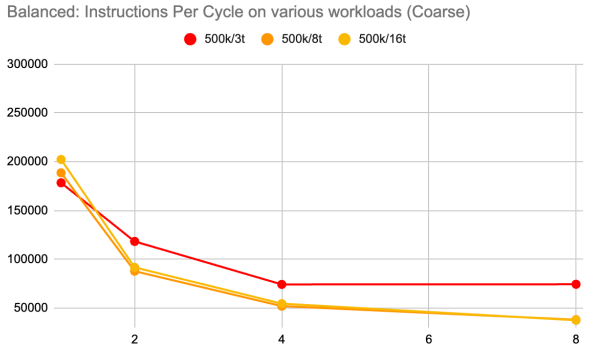
Speedup. We compared speedup for coarse-grained on 1, 2, 4, and 8 threads against our single-threaded serial implementation. Ultimately, the best speedup occurred at 6.15x on 8 threads with 500k messages, 16 tickers, and the resting workload.

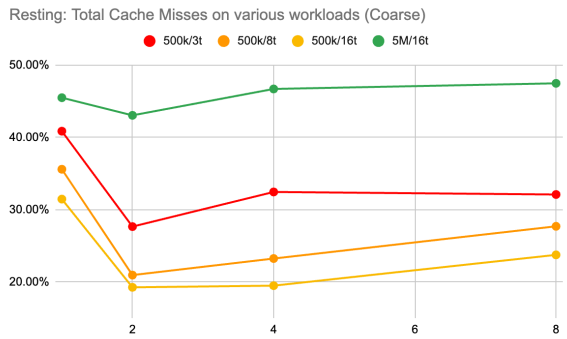


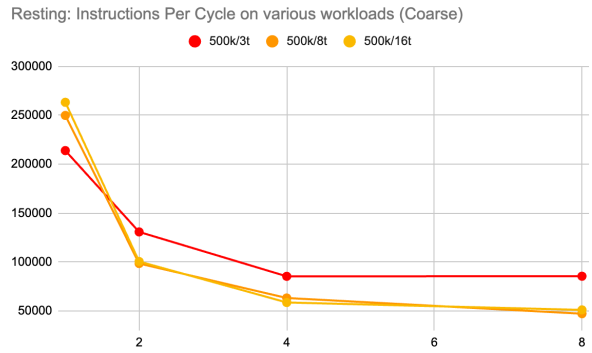
This coarse-grained locking implementation achieves strong scaling on passive-order-heavy workloads (resting: 77% of ideal at 8 threads) but fails on active-order-heavy (crossing: 27%) and load-imbalanced (skewed: 15%) workloads. The 1-thread overhead relative to sequential baseline reveals that coarse-grained locks add significant cost; parallelism is justified only when there are many independent tickers and low lock contention. The saturation of 5M/16t speedup suggests memory bandwidth, not lock granularity, becomes the limiter at greater scales.

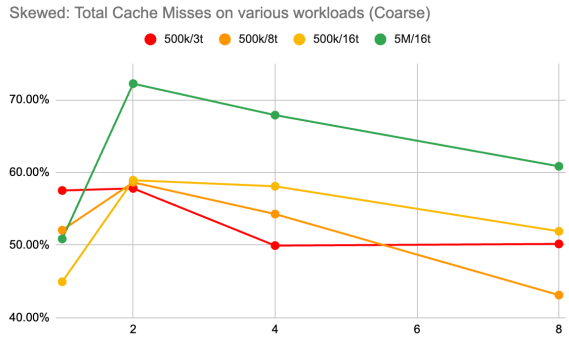
Cache Misses. The profiling results reveal that workload composition fundamentally determines parallelism scaling potential. The crossing workload exhibits the highest instructions-per-cycle throughput (1.5 - 1.85 IPC) while having the worst cache miss rates (44 - 72%), indicating that market order matching (despite requiring hand-over-hand traversal across price levels) is latency-tolerant due to high instruction-level parallelism. The tight matching loop executing arithmetic and comparison operations maintains CPU pipeline occupancy even while waiting for cache misses, enabling efficient data-parallelism across independent tickers.

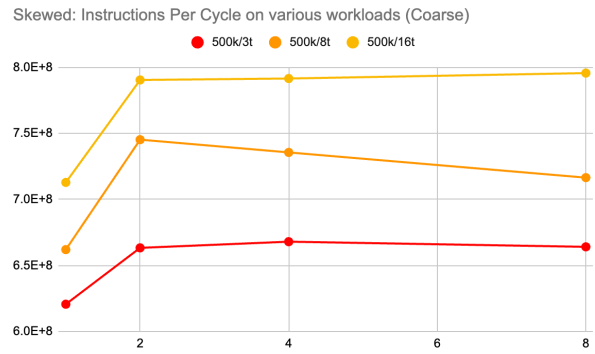












**Note: we omit certain traces if they don't make sense to be compared in a given setting.*

We observe 8-thread speedups reach 3.0x at 500k/16t despite frequent memory stalls. On the other hand, the resting workload's passive insertion operations exhibit lower IPC (0.6 - 1.2) and higher single-threaded wall times (263k µs at 500k/16t), reflecting the overhead of repeated lock acquisitions and data structure updates without the computational work that masks memory latency. This explains why crossing achieves faster absolute wall times (29k µs) than resting (51k µs) at 8 threads because crossing's work is more parallelizable and so the algorithmic cost per order is lower.

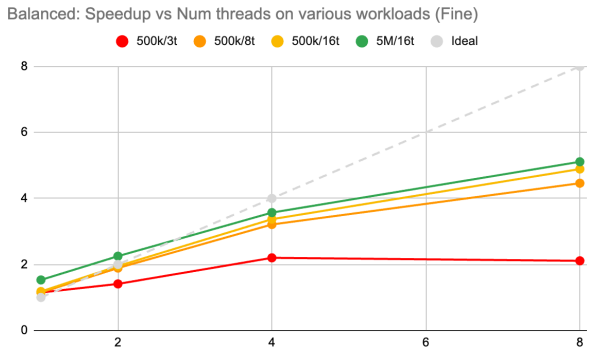
Execution Time. On 8 threads over a balanced 5M/8t workload, we found that 21.71% of the time was spent partitioning messages by ticker, 0.05% was spent on launching threads. And 78.24% was spent on actual computing. Furthermore, the time spent merging LOBs sequentially only amounted to 3% of the total unrolled thread time, suggesting that the memory optimizations

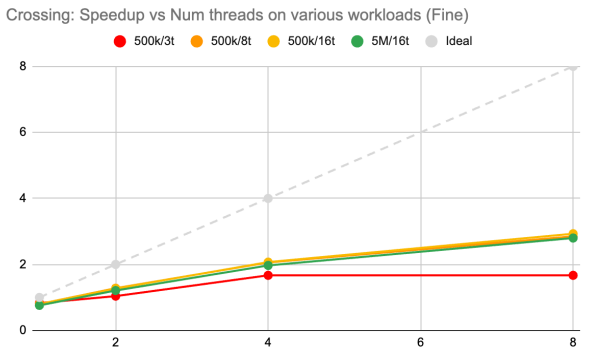
were highly effective. One potential point of improvement is reducing the time spent partitioning messages by ticker, since that is our sequential bottleneck right now. This could potentially be done by parallelizing this part with a lock-free queue for each ticker.

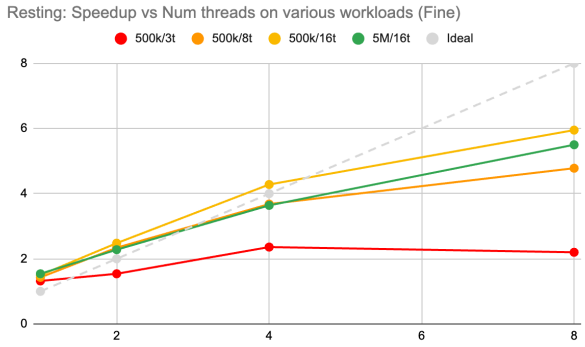
Speedup Limitations. From looking at the execution time breakdowns, the initial phase of partitioning incoming messages into ticker shards consumes roughly 21.7% of the total execution time for 8 threads, creating an upper bound on the max theoretical speedup, according to Amdahl's Law. This is the main bottleneck, causing our speedup to degrade as k scales. Additionally, from looking at the speedup of different workloads, we find that the second main reason is due to load imbalance. Since task size is quite coarse, if there are only a few tickers, as observed in our 3t configurations, or if most of the messages fall into a particular ticker, like in the skewed workload, there will be little space for parallelism, and we observe a poor speedup. Finally, at scale (5M configurations) the system likely becomes memory-bound, explaining the flattening of our speedup curve. With 50% of cache references being misses at 5M, there is likely a lot of communication and messages being passed on the data bus, in which case adding another thread would only lead to more stalling, as threads wait for their data to travel across the memory bus.

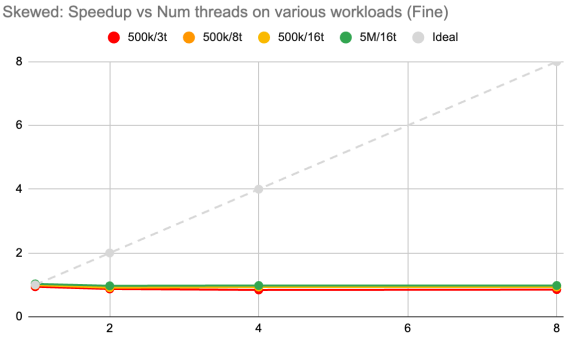
Fine-Grained

Speedup. The fine-grained locking approach, which uses per-price-level mutexes with hand-over-hand lock acquisition during order matching, underperforms the simpler coarse-grained approach across all workloads, achieving between 95-100% of coarse-grained speedup at 8 threads. This counterintuitive result stems from the synchronization overhead of hand-over-hand matching dominating any parallelism benefits. When a market order traverses multiple price levels to find liquidity, fine-grained locking requires 5.5x more lock acquisition/release pairs compared to coarse-grained's single lock pair, translating to many more memory fences and cache coherency messages. Each lock operation forces a cache coherency round-trip (approximately 3 to 4 more cycles on CPUs), accumulating to 30 - 40 total cycles of synchronization overhead per market order versus 6-8 cycles with coarse-grained locking. At the single-threaded level, fine-grained exhibits 3-5% worse performance than coarse-grained, revealing that the inherent complexity of hand-over-hand lock management and cache-line bouncing outweighs its potential benefits even without contention.



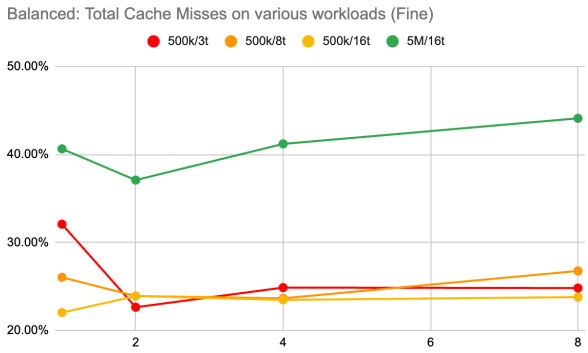


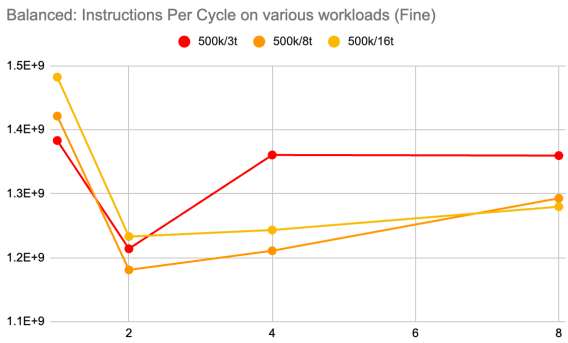


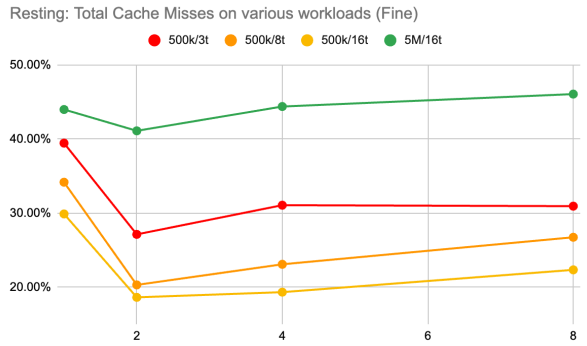


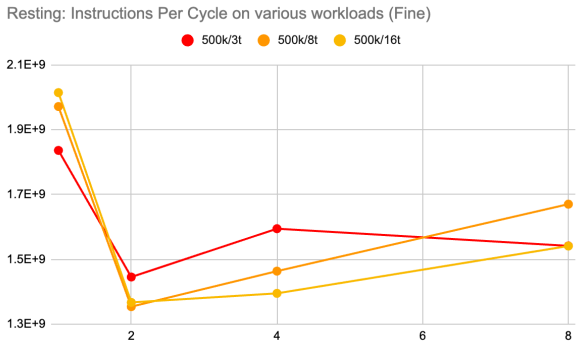
The sequential dependency chain in order matching further undermines fine-grained locking's theoretical advantage. Matching a single market order across N price levels is inherently sequential: the thread must acquire level k's lock, check for liquidity, release level k, then acquire level k+1-a dependency chain that cannot be parallelized even with per-level locks. The crossing workload, which exhibits 60% market orders, shows the largest regression from coarse-grained (2.80x compared to 2.95x at 8 threads i.e. 5% slowdown), directly reflecting the hand-over-hand overhead. Notably, the resting workload where 70% of orders skip matching entirely and call rest() directly shows identical performance between fine-grained and coarse-grained (5.50x at 8 threads), because fine-grained's ability to parallelize insertions at different price levels exactly compensates for its synchronization overhead. However, in realistic mixed workloads, the cost of matching operations makes coarse-grained locking much more well suited due to less cache coherence burden.

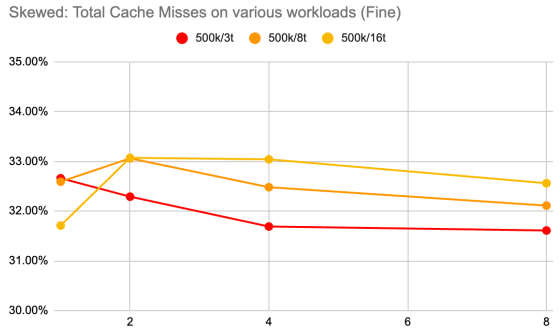
Cache Misses. Fine-grained locking's per-level mutexes reveals significant cache coherence overhead. The balanced and resting workloads show IPC peaking at 2 threads then degrading at 4+ threads despite stable cache miss rates, indicating that rather than memory latency, coherence traffic becomes the bottleneck. Each lock acquisition triggers cache invalidation across cores holding that lock's cache line, and with fine-grained's distributed locks, this overhead multiplies. We observe with the skewed workload that cache misses remain relatively the same and IPC remains invariant across all thread counts at 500k, reflectly further coherence cost. Thus, we reason that fine-grained locking trades lock contention for coherence traffic, which dominates now when threads frequently access different locks and work is serialized.

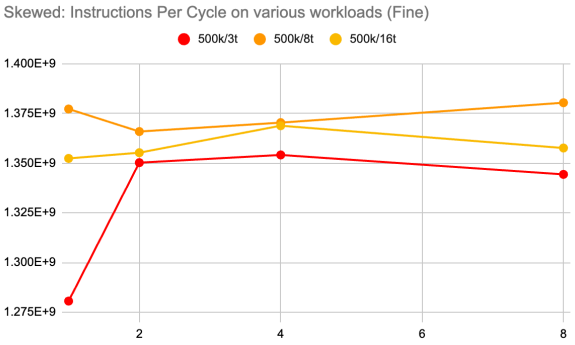












**Note: we omit certain traces if they don't make sense to be compared in a given setting.*

False Sharing. The false sharing optimization regressed 2.4% in wall-time with only a slight improvement (negligible) in cache misses.

Metric	**Original Fine**	**Padded Fine**	**Delta**
Wall Time (us)	60,650	62,129	+2.4% slower
Cycles	1,495M	1,529M	+2.3%
Instructions	2,592M	2,635M	+1.7%
IPC	1.73	1.73	No change
L3 cache misses	12.96M	12.91M	-0.4% improvement
L1 dcache load misses	14.92M	15.39M	+3.1%

We reason that the padding added up to 64 bytes of empty space to each PriceLevel struct to force cache-line alignment. This increased the memory footprint of every level object and worsened spatial locality: price levels stored in the map were now more spread out in memory, requiring longer pointer chases during traversal. The map structure itself (ordered by price, storing level pointers) became less cache-friendly. Additionally, the level mutex itself, when aligned to a 64-byte boundary within the struct, shifted the layout in ways that increased L1 load misses despite reducing aggregate cache conflicts.

Speedup Limitations. The primary bottleneck was hand-over-hand matching lock operations. A market order matching across M price levels required 2M lock operations (side-lock and level-lock for each level), whereas coarse-grained matching required only two (one global lock for the entire operation). This translated directly to a cycle count difference, where fine-grained consumed roughly 6% more cycles despite identical IPC, proving the difference was lock overhead, not computation.

Execution Time. On 8 threads over a balanced 5M/8t workload, we found that 20.1% of the time was spent partitioning messages by ticker, 0.05% was spent on launching threads, and 79.85% was spent on actual computing. Here, a major point of improvement would be reducing the lock overhead within the per thread parallel time (79.85%) as explained in Speedup Limitations.

Batching

Speedup Comparison. Recall that batching groups consecutive non-crossing limit orders to reduce lock acquisitions, and it matches coarse-grained performance within 1% across all workloads and thread counts. Wall times, cycles, and IPC remain invariant. This can be reasoned because the coarse-grained locking already holds a single shard lock for the entire critical section, so eliminating per-level contention is already achieved. Batching's grouping by side and price doesn't improve this since it still executes under the same global lock. The theoretical benefit of fewer lock acquire+release pairs is negated because coarse-grained was likely not paying high costs.

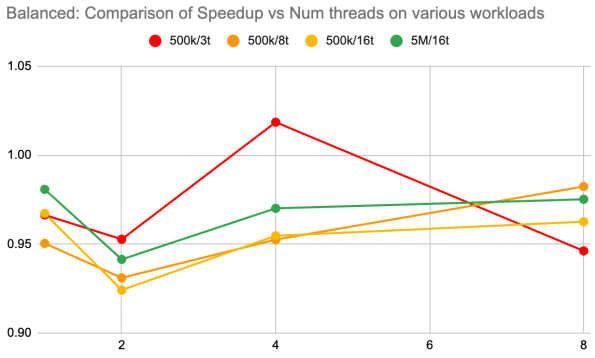
Cache Misses. Furthermore, L1 cache miss rates are indistinguishable across all three engines (i.e. Coarse, Fine, and Batching), which reveals that batching didn't provide strict cache locality benefit as we had expected. This can be due to the fact that coarse-grained's atomic shard processing already achieves optimal cache access patterns and batching's grouping strategy actually introduces more per-candidate overhead due to additional snapshot checks.

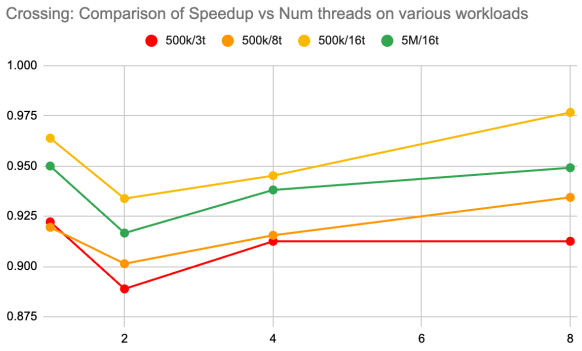
Limitations. We can conclude from little change from Coarse to Batch matching engine that coarse-grained already exposed all potential leverage or exploitation on cache coherence and locality, and that lock-count reduction alone doesn't improve performance when lock hold times are short and granularity is already rather optimal.

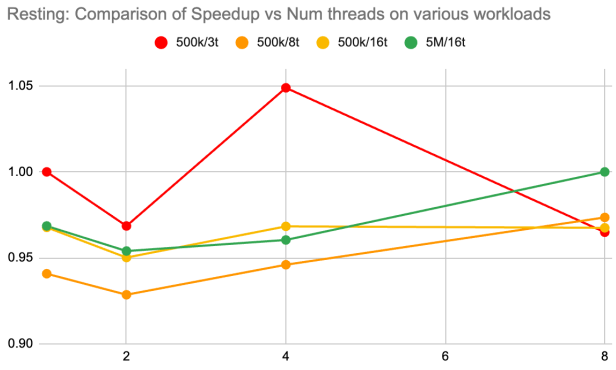
Cumulative Analysis: Comparison & Limitations

Comparison between Fine and Coarse. For the 500k order benchmark utilizing 8 threads, the coarse-grained implementation achieved a wall time of 38.9 µs, operating 6.7% faster than the fine-grained approach (41.5 µs). This difference in time is directly reflected in the total cycle count: coarse-grained required 1.22 billion cycles, while fine-grained required 1.29 billion cycles

(which is a 5.4% penalty). This confirms that the fine-grained approach requires strictly more work to accomplish the same task, which can be explained by the number of lock operations. Specifically, the coarse grained engine requires 1 global lock operation per order to access the ticker's LOB while the fine grained engine requires between 3 to 5 lock operations per market order due to hand over hand locking during price level traversal. This overhead compounds, generating the excess cycles observed above. Similarly, while both approaches exhibit sublinear scaling as thread counts increase, coarse-grained maintains a higher scaling efficiency. Specifically, we found that fine grained time was 0.9-1x that of coarse grained time for all workload distributions.







We can interpret these graphs as the ratio of speedup under different workload distributions (balanced, crossing, or resting) calculated as fine speedup divided by coarse speedup for their corresponding workload size and thread count. Most notably, crossing consistently has 2-3% poorer speedup compared to fine, whereas there appears to be spikes in better speedup for thread count 4 under 500k/3t for balanced and resting workloads.

Therefore, while hand-over-hand locking avoids holding the side mutex across the entire traversal and thus avoids stalling other threads that need to access the map structure, we see that in reality the traversal itself must be sequential because price-time priority requires visiting levels in order from best to worst, and this sequential requirement cannot be parallelized.

Moreover, each lock acquire and release operation is expensive in terms of CPU cycles, involving cache coherency protocol messages and memory barriers. Since the bid-ask spread where 80% of matches occur consists of only two to three price levels, all market orders

contend for the same level locks, and the hand-over-hand locking approach simply adds lock acquisition overhead without reducing contention.

Workload Distribution Motivation. The 6x difference between resting and crossing reveals that workload fundamentally determines bottleneck type. Resting workload minimizes hand-over-hand lock traversals, enabling fine-grained's per-level parallelism, while crossing workload maximizes them, creating sequential dependency chains that serialize matching operations. Skewed workload exposes a different bottleneck entirely which is work distribution. Even with perfect locking, if 90% of the work lands on one ticker, only one thread can process it, regardless of thread count.

Machine Target Appropriateness. Our choice to target a multicore CPU (Intel i7-9700 with pthreads) was appropriate for this problem. A GPU would have been poorly suited as the limit order book has irregular memory access patterns (linked list traversal of price levels), data-dependent branching (decide whether order crosses), and complex state updates that would be difficult to parallelize efficiently on a GPU. The sequential matching requirement (price-time priority) and per-ticker independence map naturally to CPU cores rather than GPU threads. The i7-9700's eight cores provided sufficient parallelism to explore data-parallel (cross-ticker) strategies, which proved to be the only viable parallelization approach. Shared memory across cores eliminated the need for explicit data transfer, making pthread-based synchronization appropriate.

References

1. https://github.com/brprojects/Limit-Order-Book
2. https://github.com/devmenon23/Limit-Order-Book

Work Distribution

We split work evenly (50%-50%) for this project. Specifically, our task break down was as follows:

Irene

1. Coarse Grained Implementation
2. Coarse grained profiling + results
3. Expanding golden-trace harness (more messages, tickers, and optionalities)
4. Cache book pointer once per shard for coarse grained implementation
5. Non crossing locks (level lock) for fine grained implementation
6. Fine grained cancel and modify
7. Pad data structures to avoid false sharing

Lillian

1. Serial Implementation
2. Preliminary golden-trace harness
3. Index-based sharding in Coarse grained implementation
4. Crossing locks (hand over hand) for fine grained implementation
5. Fine grained profiling + results
6. Expanding testing suite to evaluate under skewed workloads
7. Batching of non crossing orders before matching