

# Exploiting Concurrency in Domain-Specific Data Structures: A Concurrent Order Book and Workload Generator for Online Trading

Raphaël P. Barazzutti<sup>(✉)</sup>, Yaroslav Hayduk, Pascal Felber,  
and Etienne Rivière

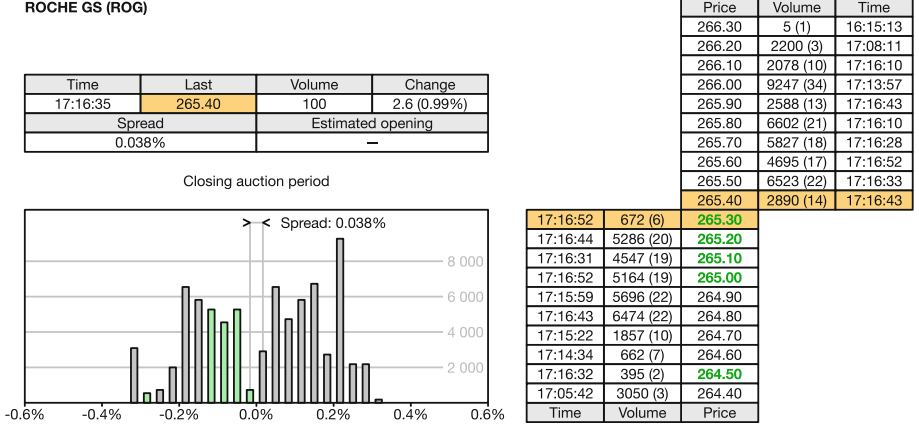
University of Neuchâtel, Neuchâtel, Switzerland  
{raphael.barazzutti,yaroslav.hayduk,pascal.felber,  
etienne.riviere}@unine.ch

**Abstract.** Concurrent programming is essential to exploit parallel processing capabilities of modern multi-core CPUs. While there exist many languages and tools to simplify the development of concurrent programs, they are not always readily applicable to domain-specific problems that rely on complex shared data structures associated with various semantics (e.g., priorities or consistency). In this paper, we explore such a domain-specific application from the financial field, where a data structure—an *order book*—is used to store and match orders from buyers and sellers arriving at a high rate. This application has interesting characteristics as it exhibits some clear potential for parallelism, but at the same time it is relatively complex and must meet some strict guarantees, notably w.r.t. the ordering of operations. We first present an accurate yet slightly simplified description of the order book problem and describe the challenges in parallelizing it. We then introduce several approaches for introducing concurrency in the shared data structure, in increasing order of sophistication starting from lock-based techniques to partially lock-free designs. We propose a comprehensive workload generator for constructing histories of orders according to realistic models from the financial domain. We finally perform an evaluation and comparison of the different concurrent designs.

## 1 Introduction

Stock exchanges provide fully automated order matching platforms to their clients. For each security available on the market, a stock exchange broker maintains a structure called an *order book*, that agglomerates orders received from clients (see Fig. 1). Orders can be of two kinds. *Bid* orders offer to buy a given security at a target (maximal) price, while *ask* orders propose to sell it, also at a target (minimal) price. A *matching engine* is in charge of comparing incoming bid and ask orders, triggering trade operations when a match exists.

With the advent of high-frequency trading, clients expect very low latencies from order matching platforms. The offered latency is actually a key commercial argument for stock exchange services [1]. Brokers and traders expect the latencies



**Fig. 1.** An order book, as seen on real trading platform.

to be in the order of a few milliseconds. This means that the stock exchange matching service needs to have internal latencies that are at least one order of magnitude lower. To achieve such low latencies, designers of brokers started using new communication mechanisms [2] to gain advantages of a few milliseconds to even microseconds, sometimes resorting to dedicated hardware and custom algorithms running on FPGAs [3].

Instead of concentrating on communication mechanisms, we focus in this paper on the effectiveness of the matching engine in order to minimize service latency and maximize throughput. The matching engine improvements are largely independent from those of communication mechanisms: as an incoming order must be processed by the matching engine before a response can be sent to the clients, a reduced matching time will improve end-to-end latency. State-of-the art matching engines thus far work sequentially [4], which means that, despite the system capacity to receive multiple orders concurrently, the processing of orders is handled one after the other. There is a great potential for obtaining performance gains for the matching operation, by taking advantage of the parallel processing capabilities of modern multi-core CPUs. We investigate in this paper the support of concurrent order processing, and explore different design strategies to introduce parallelism in the non-trivial data structure that is the order book, starting from basic lock-based techniques to more sophisticated partially lock-free algorithms. The primary objective of this study is to demonstrate how one can turn a sequential data structure into a concurrent one by carefully combining different synchronization mechanisms and reasoning about concurrency under domain-specific constraints.

Order matching has interesting characteristics as it exhibits some clear potential for parallelism: there are multiple clients and two types of orders, and matching takes place only at the frontier between the two. At the same time, it is not trivial and presents a number of challenges that must be carefully addressed.

First, we need to ensure that the output of the matching process in the concurrent case is the same as in the sequential case, notably when it comes to processing orders exactly once and according to arrival rank,<sup>1</sup> because clients are paying customers and real money is being traded. Second, as the system handles a variety of messages types (add/remove, sell/buy), it is not clear how to safely capture all message interactions in the concurrent case. Lastly, to fulfil an order, the matching engine can potentially access more than one existing order already stored in the book. In the concurrent case this might lead to several matching operations simultaneously accessing the same shared state, and special care needs to be taken to avoid possible data corruption associated with concurrency hazards. As such, the implementation need to be carefully designed so that the synchronization costs and algorithmic complexity do not outweigh the benefits associated with concurrent processing.

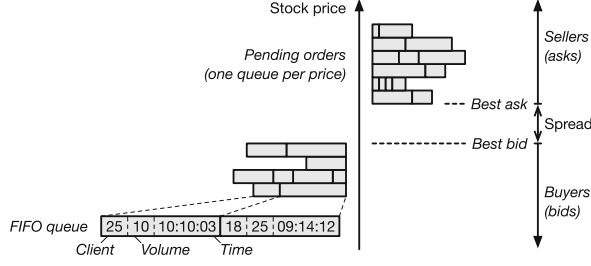
The first contribution of this work is the proposal and the evaluation of domain-specific strategies for processing orders concurrently in the order book. Specifically, the concurrent strategies we explore include: (1) a baseline thread-safe design based on a single global lock; (2) a fine-grained design for locking parts of the order book; and (3) several variants of partially lock-free designs, which trade runtime performance for weaker consistency guarantees. The second contribution of this work is the implementation of a synthetic workload generator that complies with widely-accepted models [5,6]. We further use this workload generator to assess the effectiveness of our concurrent matching algorithms.

## 2 Online Trading and the Order Book

We first describe the principle and guarantees for trading operations. We start by defining some domain-specific terms. An **order** is an investor’s instruction to a broker to buy (*bid*) or sell (*ask*) securities. There are two types of orders: *limit* and *market* orders. A **limit order** specifies a maximum purchase price or minimum selling price. A **market order** does not specify a price and will be immediately matched with outstanding orders, at the best available price for this security. The **volume** indicates the amount of securities in an order as an integer value. The **order book** is a data structure for storing unfulfilled limit orders sent for a particular security. It features two queues, one for **asks** and one for **bids** orders. Orders stored in the book can be cancelled with a specific command. Finally, the **top of the book** consists of the ask with the lowest price and the bid with the highest price, and the difference between these two prices is the **spread**.

An order book maintains two separate queues, one for *bid* orders, and one for *ask* orders. Both data structures are organized in a way that facilitates the fast extraction of the best order as well as the quick insertion of new orders. In each of the queues, orders with the same price are aggregated, i.e., queues are

<sup>1</sup> To avoid possible confusion with the word “order” used to designate trading requests and for prioritizing operations (arrival and processing order), we will only use it in the former sense and resort to alternative expressions for the latter.



**Fig. 2.** Internal structure of the order book.

organized as maps, where keys represent prices (with a granularity going up to the cent) and values are pending limit orders for a particular price. Pending orders for a particular price are sorted according to arrival rank and, thus, upon arrival are stored in a first-in first-out (FIFO) queue (Fig. 2).

Matching occurs only during order processing, when an incoming ask order satisfies some bid(s) or, vice versa, an incoming bid can be satisfied by some ask(s). When a match occurs, the associated existing orders are removed from the book. The priority of matching is driven by price, i.e., lowest selling prices (resp., highest) are sold (bought) first. If multiple orders have the same price, they are matched according to arrival time starting with the oldest. If there are no pending order to process, the system is in a *stable* state where the spread is positive and the two types of orders do not overlap.

To fulfil an order, the matching engine can “consume” more than one order on the other side of the book. This happens when an incoming order matches the best order on the opposite side of the order book, but it is not completely fulfilled and continues to match the next best order. This aggregation process stops once the incoming order has been filled completely, or when there are no more orders that can be consumed given the price and volume constraints. The remaining part of the incoming order is then added to the order book. Similarly, when the already existing order in the order book cannot be fully matched with the incoming order, the partially-matched order remains in the book with its volume decreased by the volume subsumed by the transactions.

The pseudo-code for the baseline sequential matching algorithm is shown in Algorithms 1 and 2. Sell and buy orders are stored in two separate *heaps*, each holding FIFO queues with orders of identical price sorted according to their arrival time. Queues are sorted by increasing price in the asks heap, and by decreasing price in the bids heap. The algorithm matches incoming orders against existing ones from the opposite heap, possibly adding them to the book if they are not completely fulfilled. To keep the pseudo-code as simple as possible, we assume that the heap at other side of the book is not empty when inserting an order and we do not explicitly handle the creation and removal of the queues in the heaps. This code is used as a basis for the concurrent variants presented in Sect. 3.

**Algorithm 1.** Helper functions.

---

```

1: Type ORDER is:
2:   type: {LIMITED, MARKET}                                ▷ Limited or market price?
3:   operation: {BUY, SELL}                                   ▷ Buy or sell?
4:   volume: integer                                          ▷ How many securities?
5:   price: float                                             ▷ At what price?
6:   ...
7:   id: integer                                              ▷ Timestamp (for concurrent algorithms)
8:   status: {IDLE, MATCHING, REMOVED}                       ▷ Status (for concurrent algorithms)

9: Type BOOK is:
10:  asks: heap of FIFO queues (orders of same price) ▷ Sorted by increasing price
11:  bids: heap of FIFO queues (orders of same price) ▷ Sorted by decreasing price
12:  ...

13: function CAN_MATCH(node, order)  ▷ Can incoming order match node in book?
14: if order.operation = node.operation then                ▷ Need order of opposite type
15:   return false
16: if order.type = MARKET then                                ▷ Market orders always match
17:   return true
18: if order.operation = SELL then
19:   return order.price ≤ node.price
20: else
21:   return order.price ≥ node.price

```

---

**Algorithm 2.** Sequential order insertion (single-threaded).

---

```

1: function HANDLE_ORDER_SEQ(order)
2:   sell ← (order.operation = SELL)
3:   while order.volume > 0 do
4:     q ← TOP(sell ? book.bids : book.asks)  ▷ Non-empty top queue on other side
5:     n ← FIRST(q)                             ▷ Top order in the queue
6:     if ¬CAN_MATCH(n, order) then
7:       q ← GET(sell ? book.asks : book.bids, order.price)  ▷ Queue at price
8:       PUSH(q, order)                                ▷ Store order in book (append to queue)
9:       break
10:    if n.volume > order.volume then
11:      n.volume ← n.volume − order.volume
12:      break
13:    order.volume ← order.volume − n.volume
14:    POP(q)                                             ▷ Remove top order from other heap
15:   return SUCCESS

```

---

### 3 A Concurrent Order Book

We now describe different strategies for supporting concurrency in the order book. This data structure is interesting because it is non-trivial and the matching operation may be time-consuming (e.g., when an incoming order matches and “consumes” many existing orders from the book). Hence, taking advantage of the parallel processing capabilities of recent multi-core architectures is obviously desirable.

It is, however, not easy to perform concurrent operations on the order book while at the same time preserving consistency. Some synchronization is necessary for correctness, but too much synchronization may hamper performance. We will start by discussing simple synchronization techniques and gradually move to more sophisticated strategies that achieve higher levels of concurrency.

In all concurrent approaches discussed below, requests to the order book are handled and processed by a pool of threads. As we would like the order book to yield the same output in a concurrent execution as when processing operations one at a time, we need to process requests in the same sequence as they have been received. We therefore insert incoming requests in a FIFO queue<sup>2</sup> and assign to each request a unique, monotonously increasing timestamp that we use to sort operations (see Algorithm 3, lines 1–8). We will discuss later scenarios where we can process some requests in a different sequence while still preserving the linearizability of the order book operations.

Before discussing our strategies for handling concurrent operations, let us first consider some observations about the specific properties of the order book. First, matching always occur at the top of the book. Therefore, the matching operation has interesting locality properties, and it will not conflict, for instance, with operations that are “far enough” from the top of the book. Second, an order that is matched upon insertion only needs to be inserted in the book if it is not fully matched. We can thus identify two interesting common cases: (1) an order is not inserted at the top of the book and hence no matching occurs, and (2) an order inserted at the top of the book is fully subsumed by existing orders and therefore does not need to be inserted in the book.

Finally, there are several scenarios where we can straightforwardly determine that two concurrent limit orders do not conflict. For instance, insertions of an ask and a bid can take place concurrently if there is no price overlap between them, i.e., the ask has a higher price than the sell, as they cannot both yield a match. As another example, insertions of two limit asks or two limit bids can take place concurrently if they have different prices (i.e., they are in different queues) and they do not both yield a match. These observations will be instrumental for the design of advanced concurrency strategies.

### 3.1 Coarse-Grained Locking

We first consider the trivial approach of using a single lock (SGL) to serialize accesses to the shared data structure. To simplify the presentation of concurrent algorithms, we assume that threads from the pool repeatedly execute the function `THREAD_PROCESS` to process one order from the incoming queue, and the result of this function is then returned to the corresponding client. The basic operating principle of the coarse-grained approach is shown in Algorithm 3, lines 9–14. Threads from the pool acquire the main lock, process the next order

---

<sup>2</sup> We assume that this queue is thread-safe as processing threads may dequeue orders concurrently with one another and with the (unique) thread that enqueues incoming orders.

from the queue, and release the lock before the response is sent back to the client. Hence, the processing of orders is completely serialized and no parallelism takes place for this operation. The main advantage of this approach is its simplicity, which also makes the algorithm easy to prove correct. It will serve as a baseline for the rest of the paper.

---

**Algorithm 3.** Coarse-grained locking and common functions.

---

```

1: Variables:
2:  incoming: FIFO queue (orders)           ▷ Thread-safe queue for incoming orders
3:  ts: integer                             ▷ Timestamp for incoming orders (initially 0)
4:  sgl: lock                               ▷ Single global lock (initially unlocked)
5:  ...
6: upon RECEIVE(order):                     ▷ Reception of an order from a client (single thread)
7:   order.id  $\leftarrow$  ts                     ▷ Assign unique timestamp
8:   PUSH(incoming, order)                 ▷ Append order to queue
9:   ts  $\leftarrow$  ts + 1
10: function THREAD_PROCESSsgl              ▷ Processing of an order by a thread
11:  order  $\leftarrow$  POP(incoming)              ▷ Take next order from queue
12:  LOCK(sgl)                               ▷ Serialize processing
13:  r  $\leftarrow$  HANDLE_ORDER_SEQ(order)       ▷ Use sequential algorithm
14:  UNLOCK(sgl)
15:  return r

```

---

### 3.2 Two-Level Fine-Grained Locking

We now explore opportunities for finer-grained locking to increase the level of concurrency. We start from the observation that two threads accessing limit orders from the book with different prices, i.e., located in different queues in the heaps, can do so concurrently without conflicts. Therefore, it is only necessary to control access to the queues that are accessed by both threads.

The principle of the two-level locking strategy is shown in Algorithm 4. As before, threads first attempt to acquire the main lock (line 3). Once a given thread acquires the main lock, it traverses the queues in the opposite heap of the book in sequence, starting from the top, and locks each visited queue individually (lines 7–12). This process stops as soon as the accumulated volume of orders in already traversed queues reaches the volume of the incoming order. In case of a limit buy or ask order, the process also stops whenever visiting a queue at a price that is higher, respectively lower, than the price of the incoming order. Finally, if the incoming order has not been fully matched, it needs to be inserted in the book and we also lock the queue associated with the price of the incoming order (lines 13–15). The algorithm then releases the main lock (line 16). It can now safely perform the actual matching operations on the previously locked queues, including the optional insertion of the incoming order in the book if some unmatched volume remains, and release the individual locks as soon as they are

**Algorithm 4.** Fine-grained locking.

---

```

1: function THREAD_PROCESSFGL                                ▷ Processing of an order by a thread
2:   order  $\leftarrow$  POP(incoming)                                ▷ Take next order from queue
3:   LOCK(sgl)                                                  ▷ Serialize traversal of heap
4:   sell  $\leftarrow$  (order.operation = SELL)
5:   v  $\leftarrow$  order.volume
6:   q  $\leftarrow$  TOP(sell ? book.bids : book.asks)                ▷ Top queue at other side
7:   while v > 0 do
8:     if  $\neg$ CAN_MATCH(FIRST(q), order) then
9:       break
10:    LOCK(q)                                                    ▷ Acquire lock on queue
11:    v  $\leftarrow$  v - VOLUME(q)                                ▷ Subtract volume of all orders in queue
12:    q  $\leftarrow$  NEXT(q)                                         ▷ Next queue from heap
13:  if (v > 0) then
14:    q  $\leftarrow$  GET(sell ? book.asks : book.bids, order.price)  ▷ Queue at price
15:    LOCK(q)                                                    ▷ Acquire lock on queue
16:  UNLOCK(sgl)
17:  r  $\leftarrow$  HANDLE_ORDER_SEQFGL(order)                      ▷ Use sequential algorithm
18:  return r

function HANDLE_ORDER_SEQFGL  $\equiv$  HANDLE_ORDER_SEQSGL          ▷ Algorithm 2
... UNLOCK(q) ...                                             ▷ All locks released once no longer needed (before line 15)

```

---

no longer needed. To that end, we simply reuse the sequential algorithms with the addition of lock release, which happen right before line 15 in Algorithm 2 or whenever a queue becomes empty.<sup>3</sup>

This approach provides higher concurrency than coarse-grained locking because the algorithm holds the main lock for a shorter duration, when determining which queues from the book will be accessed. To ensure consistency, it uses a second level of locks for concurrency control at the level of individual queues. Therefore, multiple orders can execute concurrently if they operate in different parts of the order book, but they are serialized if the sets of queues they access overlap.

### 3.3 Toward Lock-Free Algorithms

The final stage in our quest for concurrency is to try to reduce the dependencies on locks, whether coarse- or fine-grained, as they introduce a serial bottleneck and may hamper progress. In particular, a thread that is slow, faulty, or pre-empted by the OS scheduler while holding a lock may prevent other threads from moving forward.

Our objective is thus to substitute locking operations by lock-free alternatives. To that end, we first need to remove the locks protecting the queues and permit threads to enqueue and dequeue orders concurrently. We do so by replacing the queues in Algorithm 1, lines 9–10, by a concurrent heap structure for

<sup>3</sup> Some implementation details, such as avoiding a second traversal of the heap by keeping track of locked queues, are omitted for simplicity.



backing the order book. Specifically, we use a concurrent map,<sup>4</sup> which imposes a custom sorting of the orders it contains. First, orders are sorted according to prices, and then, according to the timestamp.

To handle concurrent accesses explicitly, we also add in the order object an additional status flag that we use to indicate whether the order is being processed or has been removed by some thread. We modify this flag in a lock-free manner using an atomic compare-and-set (CAS) operation.

The *order.status* flag (Algorithm 1, line 7) can be in one of three states: **IDLE** indicates that the order is not being processed by any thread; **MATCHING** specifies that some thread is processing the order; and **REMOVED** means that the order, although still present in the order book, has been logically deleted.

We have developed three variants of the concurrent, almost<sup>5</sup> lock-free algorithm, with each having different guarantees. The first algorithm, which we call **LF-GREEDY**, provides the least guarantees in terms of the sequence in which orders are processed. The second algorithm, **LF-PRIORITY**, prevents an incoming order from consuming new orders that have arrived later. The last algorithm, **LF-FIFO**, additionally prevents incoming orders arriving later from stealing existing orders from incoming orders arriving earlier.

For the sake of simplicity, the pseudo-code as presented further does not show the handling of market orders. Instead it only considers the more general case of limit orders. In the case of market orders, if there is no or only a partial match, the unmatched orders are returned back to the issuer. We furthermore omit obvious implementation-specific details, e.g., an incoming order is naturally matched against the opposite side of the order book.

**The LF-Greedy Algorithm.** The **LF-GREEDY** algorithm (see Algorithm 5, omitting text within square brackets) works as follows. After the incoming order *order* has been received and scheduled for processing, the worker thread obtains the best order *n* from the order book’s heap. Orders in our lock-free algorithms can be marked as **MATCHING** to indicate that they are being processed, or as **REMOVED** when logically deleted but still physically present in the order book. As such, the thread first checks if *n* has been marked as removed (line 5). If so, it removes *n* from the order book (line 6) and continues to another iteration of the algorithm. Otherwise, we know that *n* has not been removed, and we need to check whether some other thread has already started processing *n*, in which case we wait until the processing has finished by polling the *n.status* flag (line 8). Thereafter, we attempt to change the status of *n* from **IDLE** to **MATCHING** using CAS (line 9). If the CAS operation succeeds, then we know that the order was indeed idle (note that it could have been removed in the meantime, or taken over for matching by another thread) and the thread has successfully taken exclusive

<sup>4</sup> `java.util.concurrent.ConcurrentSkipListMap`.

<sup>5</sup> While the algorithms do not use explicit locks, they are not completely “lock-free” as in some situations a thread may be blocked waiting for the status flag to be updated by another thread. Techniques based on “helping” could be used to avoid such situations, at the price of increased complexity in the algorithms. We therefore slightly abuse the word “lock-free” in the rest of the paper.

**Algorithm 5.** Greedy [and priority] order insertion algorithm.

---

```

1: function THREAD_PROCESSGREEDY/PRIORITY      ▷ Processing of an order by a thread
2:   order ← POP(incoming)
3:   while order.volume > 0 do
4:     n ← 1st node from heap [ such that n.id < order.id ]
5:     if n.status = REMOVED then                                ▷ Order logically removed?
6:       heap ← heap \ {n}                                     ▷ Yes: remove order from book
7:       continue
8:       wait until n.status ≠ MATCHING                          ▷ Avoid useless CAS
9:       if ¬CAS(n.status, IDLE, MATCHING) then                 ▷ Take ownership of node
10:        continue
11:        if ¬CAN_MATCH(n, order) then                            ▷ Can we match order?
12:          heap ← heap ∪ {order}                               ▷ No: store order in book
13:          n.status ← IDLE
14:          break
15:          if n.volume > order.volume then                      ▷ Order fully satisfied
16:            n.volume ← n.volume − order.volume
17:            n.status ← IDLE
18:            break
19:            order.volume ← order.volume − n.volume           ▷ Node fully consumed
20:            n.status ← REMOVED
21:  return SUCCESS

```

---

ownership over it. If the CAS operation fails, then some other thread must have just changed the order's status to either `MATCHING` or `REMOVED` and we continue to another iteration of the algorithm.

After taking ownership of *n*, we need to check if the price of the incoming order *order* could be matched with the price of *n*. If not, we store *order* in the order book and release *n* by setting its status to `IDLE` (line 13), effectively finishing the matching process of the incoming order. Otherwise, if the prices of *n* and *order* can be matched, we check if the incoming order *order* could fully consume *n*. If so, we decrease the incoming order's volume (line 19) and mark *n* as `REMOVED`. Note, that we do not physically remove *n* from the *heap* at this step; instead, we rely on other threads' help for removing it lazily (lines 5 and 6). If the volume of *n* is larger than *order* can consume, we decrease it and unlock *n* (line 17). This implies that the outstanding volume of *n* remains in the order book and can be consumed by other threads.

If the incoming order has a large volume, it can potentially consume multiple orders from the book. In this version of the algorithm we do not enforce any restrictions on which existing orders can be consumed by the incoming order, i.e., concurrent threads might consume existing orders that are interleaved in the heap.

**The LF-Priority Algorithm.** The LF-PRIORITY algorithm provides more guarantees in terms of sequence in which incoming orders consume existing orders stored in the book. Specifically, when matching an incoming order *order* with the content of the book, we want to only consider existing orders that have

been received strictly before *order* has been received. To that end, we rely on the timestamp *order.id* assigned to each order upon arrival (Algorithm 3, lines 6). We then modify Algorithm 5 by adding an extra condition (line 4 between square brackets), which restricts *order* to only process orders from the order book having a smaller timestamp. This condition can be supported straightforwardly in our implementation because of the key we use to store orders in the concurrent heap. Indeed, we use the same concurrent map as before and, when retrieving the best order, we apply an extra filter condition to select orders having keys with timestamps that are smaller than the currently processed order.

**The LF-FIFO Algorithm.** To introduce the LF-FIFO algorithm, we first informally discuss where LF-PRIORITY is lacking and how its shortcomings can be addressed. In Algorithm 5, if the thread processing an order is delayed (e.g., preempted by the OS scheduler), an order arriving later might consume the best outstanding orders in the book. This is a problem if one needs to enforce that orders arriving first are given precedence over orders arriving later. Furthermore, concurrent orders may consume interleaved orders from the book, i.e., an incoming order may be matched against a set of existing orders that does not represent a continuous sequence in the book, hence breaking atomicity. The main idea of LF-FIFO is therefore to prevent threads from consuming orders from the book before the processing of incoming orders received earlier has finished.

To provide these stricter guarantees, we employ ideas from the hand-over-hand locking [7] technique. The principle is that, when traversing a list, the lock for the next node needs to be obtained while still holding the lock of the current node. That way, threads cannot overtake one another. The pseudo-code of the LF-FIFO is shown in Algorithm 6. A thread processing an incoming order *order*, which would consume multiple existing orders, first performs a CAS on the first best order (line 12), marking it as removed in the end (line 26). Then it saves the first best order in a local variable (line 28) and continues to another iteration, during which it selects the second best node (line 5) and performs a CAS to atomically change its status to **MATCHING**. Upon success and only then do we physically remove the first best node from the heap (line 15).

The process describing order removals is distinctly different from that which was presented in prior algorithms. In LF-GREEDY and LF-PRIORITY algorithms, when an arbitrary thread detects that an order has been marked as **REMOVED** (Algorithm 5, line 5), it helps by removing that order (i.e., lazy removal with helping). In contrast, instead of assisting in the removal of *n* from the heap, the LF-FIFO algorithm restarts from the beginning (line 9), relying on the thread that has marked *n* as **REMOVED** to also physically remove it from the heap (lines 15 and 30). Therefore, the LF-FIFO algorithm provides weaker progress guarantees but better fairness between threads.

**Algorithm 6.** Order insertion algorithm with FIFO properties.

---

```

1: function THREAD_PROCESSFIFO                                ▷ Processing of an order by a thread
2:    $order \leftarrow \text{POP}(\text{incoming})$ 
3:    $p \leftarrow \perp$                                            ▷ Previous node fully matched by thread
4:   while  $order.volume > 0$  do
5:      $n \leftarrow 1^{\text{st}}$  node  $n \neq p$  from  $heap$  such that  $n.id < order.id$ 
6:     if  $n = \perp$  then                                         ▷ Any matching order in book?
7:        $heap \leftarrow heap \cup \{order\}$                      ▷ No: store order in book
8:       break
9:     if  $n.status = \text{REMOVED}$  then                             ▷ Order logically removed?
10:      continue                                              ▷ Yes: wait until physically removed
11:    wait until  $n.status \neq \text{MATCHING}$                        ▷ Avoid useless CAS
12:    if  $\neg \text{CAS}(n.status, \text{IDLE}, \text{MATCHING})$  then         ▷ Take ownership of node
13:      continue
14:    if  $p \neq \perp$  then
15:       $heap \leftarrow heap \setminus \{p\}$                      ▷ Delayed removal
16:       $p \leftarrow \perp$ 
17:    if  $\neg \text{CAN\_MATCH}(n, order)$  then                         ▷ Can we match order?
18:       $heap \leftarrow heap \cup \{order\}$                      ▷ No: store order in book
19:       $n.status \leftarrow \text{IDLE}$ 
20:      break
21:    if  $n.volume > order.volume$  then                           ▷ Order fully satisfied
22:       $n.volume \leftarrow n.volume - order.volume$ 
23:       $n.status \leftarrow \text{IDLE}$ 
24:      break
25:     $order.volume \leftarrow order.volume - n.volume$            ▷ Node fully consumed
26:     $n.status \leftarrow \text{REMOVED}$ 
27:     $n.id \leftarrow order.id$                                    ▷ Prioritize concurrent insertions
28:     $p \leftarrow n$                                            ▷ Keep in book (to avoid being overtaken)
29:    if  $p \neq \perp$  then
30:       $heap \leftarrow heap \setminus \{p\}$                      ▷ Remove last consumed order
31:  return SUCCESS

```

---

## 4 Generating Workloads

Besides algorithms for exploiting concurrency in the order book operation, we contribute in this section a workload generator that allows evaluating the throughput of the matching operation under realistic workload assumptions.

The sensitive nature of financial data and the strict rights of disclosures signed between clients of stock quote operators typically prevent from using real datasets and call instead for appropriate models for synthetic data generation. Models emerged in economics and econophysics (i.e., physicists' approaches to tackle problems in economics) such as the ones by Maslov [5], Bartolozzi [8] and Bak et al. [9]. These models allow understanding the properties of the order book in terms of the total volume of securities available or requested at each price point in the bid and ask queues. This aggregated information is enough for the targeted users of these models, who are interested in modelling and implementing investment strategies based on the total volume of securities at each price point, independently from their origin or destination. The distribution

of individual order sizes has been studied separately, and shown to follow a power law by several authors [6, 10, 11]. Some models that consider individual orders nonetheless use a unit order size rather than a distribution in the interest of simplicity [5, 12].

We implement a variation of the model proposed by Maslov [5], which uses simple rules and which output has been shown to compare well with the behaviour of a real limit order-driven market. The original model assumes however, similarly to [12], that all orders have the same volume of one single security. This simplification is problematic for testing a matching engine, in particular for testing its behaviour and performance in the presence of partially matched orders. We therefore extend the model by allowing orders to feature arbitrary volumes and assign volumes following a power law distribution based on findings made by Maslov and Mills in [6]. We note that another limitation of this model is that it does not consider changes to existing orders stored in the order book, unlike for instance the Bak-Paczuski-Shubik model [9]. We choose not to address this limitation as it does not fundamentally limit the representativeness of the behaviour of clients using the order book for what concerns the matching algorithm itself. The expiry mechanism for existing orders proposed by the model, along with new insertions is indeed enough to model dynamics.

We now proceed to detailing the model itself. An average price  $p$  is fixed at the beginning of the generation, which starts by the generation of one bid and one ask limit order. Thereafter, orders are generated by first deciding on their operation (bid or ask), with equal priority. Each order is a *limit* order with priority  $q_{lo}$ , and a *market* order otherwise. The price attached to a limit order is generated based on the base price  $b$  of the best available order on the other side of the order book: the cheapest ask for a bid, and the largest bid for an ask. A random variation  $\Delta$ , generated randomly in  $\{1, 2, \dots, \Delta_{max}\}$  is applied: the price for the order is set to  $p(t) + \Delta$  for a bid, or to  $p(t) - \Delta$  for an ask. The volume  $v$  for each order is generated according to the power law identified in [6]. For market orders,  $P[v] \propto v^{-1-\mu_{market}}$  where  $\mu_{market} = 1.4$ . For limit orders,  $P[v] \propto \frac{1}{v} e^{-\frac{(A-\ln(v))^2}{B}}$ , where  $A = 7$  and  $B = 4$ . These values for  $\mu_{market}$ ,  $A$  and  $B$  are the ones suggested in the original paper [6], as are the values we use for the other parameters:  $q_{lo} = \frac{1}{2}$ , and  $\Delta = 4$ . We use an initial price of  $p = 1,000$ .

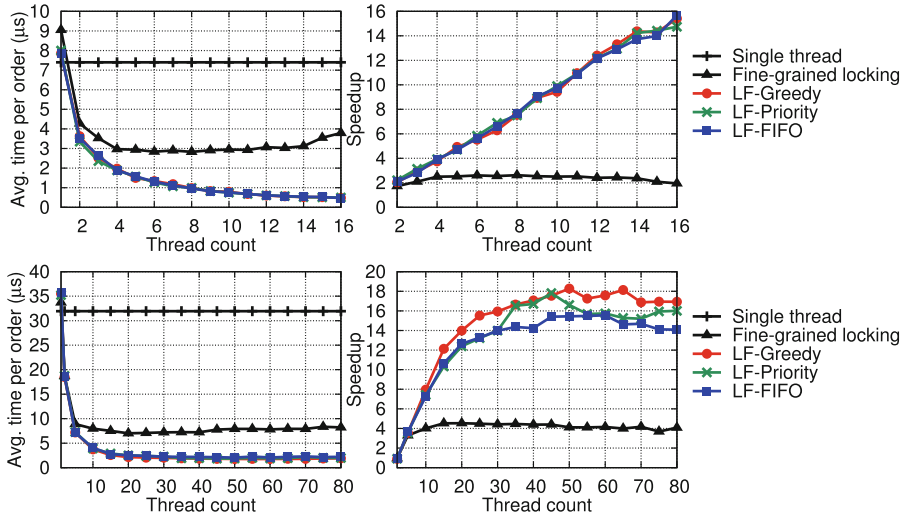
In order to prevent limit orders staying indefinitely in the order book, an expiry mechanism removes unmatched limit orders from the order book after  $\lambda_{max}$  time steps. The expiry mechanism prevents the accumulation of limit orders having prices that differ significantly from the current market price. In the real market, this operation is performed by either traders or by the stock exchange itself. For instance, the New York Stock Exchange purges all unmatched orders at the end of the day. Maslov indicates that for any reasonably large value of the cut-off parameter  $\lambda_{max}$ , the model produces the same scaling properties of price fluctuations. We use  $\lambda_{max} = 1,000$  as in the original paper.

## 5 Evaluation

We experiment on two different architectures. The first is an Intel i7-5960X Haswell CPU (8 cores, 16 hardware threads with hyperthreading enabled) with 32 GB of RAM. The second is an IBM POWER8 S822 server (10 cores, 80 hardware threads) with 32 GB of RAM. We run our experiments in OpenJDK’s Runtime Environment, build 1.8.0\_40, with default options.

For all concurrent order book implementations considered, we process 100,000 orders in total. We vary the thread count from 1 to the maximum number of threads supported by each architecture. We run all experiments 10 times and present the average. The orders are generated offline using the model from Sect. 4, kept in memory and replayed directly to each of the order book implementations. For each of the experiments performed, we also plot the obtained speedup related to the baseline sequential matching engine running with a single thread.

For all the tests, we observe that the lock-free approaches outperform fine-grained locking. The latter approach does not scale beyond 4 threads for the Haswell architecture and 8 threads for POWER8. When more threads are used, however, its performance does not degrade significantly and remains relatively constant. In contrast, when looking at the lock-free approaches, we see that they scale almost linearly. Also, we see that the more guarantees in terms of the sequence in which orders are processed a lock-free algorithm provides, the slower it performs. The variations in performance are, however, minimal.



**Fig. 3.** Order processing time (average over 100,000) and speedup for different order book implementations on the Intel Haswell (top) and IBM POWER8 (bottom) architectures.

## 6 Related Work

Strategies for optimizing the operation of matching platforms can be broadly divided into two categories: the reduction of latency and optimizations related to order processing. To achieve ultra-low latencies, high-frequency trading servers are typically housed in the same building as the matching engine servers [2]. Additionally, novel communication technologies, such as microwaves, are gaining popularity as they promise to convey orders faster than fibre optic [13].

Significant effort was also spent towards efficient middleware systems for order handling, besides the matching operation itself. *LMAX Disruptor* [14] is an integrated trading system running on the JVM. It implements the reception and pre-processing of orders. It stores the received orders in a queue with ordering guarantees similar to the *incoming* queue used in our algorithms. Disruptor features a simple single-threaded matching engine that fetches and process orders from the queue sequentially, but it also allows the implementation of more sophisticated matching or order processing engines including those using multiple-threads implementation. It is therefore complementary to our study, which concentrates on the internal of the matching engine.

Although, to the best of our knowledge, there does not exist concurrent lock-free implementations of matching engines, substantial effort has been dedicated to developing efficient single-threaded implementations. For instance, Shetty et al. [15] propose such an implementation for the .NET platform. The authors detail the steps required for locking the order book when accessing it from multiple threads concurrently, similarly to our baseline coarse-grain locking algorithm.

In addition to the models for generating orders that we mentioned in Sect. 4, several authors investigated the *dynamics* of order books. Huang et al. [16] propose a market simulator to help compute execution costs of complex trading strategies. They do so by viewing the order book as a Markov chain and by assuming that the intensities of the order flows depend only on the current state of the order book. Cont et al. [17] propose using a continuous-time stochastic model, capturing key empirical properties of order book dynamics. Alternatively, Kercheval et al. [18] use a machine learning framework to build a learning model for each order book metric with the help of multi-class support vector machines.

## 7 Conclusion

We proposed in this paper strategies for performing order matching in the order book in a concurrent manner. We started with two lock-based implementations using coarse- and fine-grained locking designs. We then proposed three algorithms that do not use explicit locks and provide different guarantees in terms of the sequence of order processing. We also contributed a workload generator that allows us to evaluate the throughput of order matching under realistic workload assumptions. Experimental results suggest that, although the fine-grained approach scales only up to a few cores, by carefully substituting locking operations

by lock-free alternatives, we can achieve high performance and good scalability. Future work might target combining our concurrent matching engine with *LMAX Disruptor*, forming a cohesive framework where both, order dispatch and matching, are executed in an almost lock-free manner.

## References

1. Ende, B., Uhle, T., Weber, M.C.: The impact of a millisecond: measuring latency effects in securities trading. In: Wirtschaftsinformatik Proceedings, Paper 116 (2011)
2. WIRED: Raging bulls: how wall street got addicted to light-speed trading (2012). <http://www.wired.com/2012/08/ff-wallstreet-trading/2/>
3. Leber, C., Geib, B., Litz, H.: High frequency trading acceleration using FPGAs. In: International Conference on Field Programmable Logic and Applications, FPL (2011)
4. Preis, T.: *Ökonophysik - Die Physik des Finanzmarktes*. Springer, Wiesbaden (2011)
5. Maslov, S.: Simple model of a limit order-driven market. *Phys. A Stat. Mech. Appl.* **278**(3), 571–578 (2000)
6. Maslov, S., Mills, M.: Price fluctuations from the order book perspective - empirical facts and a simple model. *Phys. A Stat. Mech. Appl.* **299**(1), 234–246 (2001)
7. Lea, D.: *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Boston (1996)
8. Bartolozzi, M.: Price variations in a stock market with many agents. *Eur. Phys. J. B* **78**(2), 265–273 (2010)
9. Bak, P., Paczuski, M., Shubik, M.: Price variations in a stock market with many agents. *Phys. A Stat. Mech. Appl.* **246**(3–4), 430–453 (1997)
10. Gabaix, X.: Power laws in economics and finance, Technical report. National Bureau of Economic Research (2008)
11. Bouchaud, J.-P., Mézard, M., Potters, M.: Statistical properties of stock order books: empirical results and models. *Quant. Finan.* **2**(4), 251–256 (2002)
12. Khanna, K., Smith, M., Wu, D., Zhang, T.: Reconstructing the order book, Technical report. Stanford University (2009)
13. Singla, A., Chandrasekaran, B., Godfrey, P.B., Maggs, B.: The internet at the speed of light. In: 13th ACM Workshop on Hot Topics in Networks, HotNets (2014)
14. Thompson, M., Farley, D., Barker, M., Gee, P., Stewart, A.: Disruptor: high performance alternative to bounded queues for exchanging data between concurrent threads. White paper (2011). <http://disruptor.googlecode.com/files/Disruptor-1.0.pdf>
15. Shetty, Y., Jayaswal, S.: The order-matching engine. In: *Practical .NET for Financial Markets*. Apress, pp. 41–103 (2006)
16. Huang, W., Lehalle, C.-A., Rosenbaum, M.: Simulating, analyzing order book data: the queue-reactive model. *J. Am. Stat. Assoc.* **110**(509), 107–122 (2013)
17. Cont, R., Stoikov, S., Talreja, R.: A stochastic model for order book dynamics. *J. Am. Stat. Assoc.* **58**(3), 549–563 (2010)
18. Kercheval, A.N., Zhang, Y.: Modelling high-frequency limit order book dynamics with support vector machines. *Quant. Financ.* **15**(8), 1315–1329 (2015)



Networked Systems

4th International Conference, NETYS 2016, Marrakech,

Morocco, May 18-20, 2016, Revised Selected Papers

Abdulla, P.A.; Delporte, C. (Eds.)

2016, X, 396 p. 118 illus., Softcover

ISBN: 978-3-319-46139-7