O(NLogN) Approach Output (54.552 seconds) Using Heap and hashMap

```
Processing started at: 2023-11-07 16:29:10.412055
Book: book-1
      Buy -- Sell
=======================
49@100.00 -- 33@100.10
97@100.00 -- 20@100.20
38@100.00 -- 37@100.30
63@99.90 -- 7@100.30
10@99.80 -- 97@100.40
40@99.70 -- 4@100.40
42@99.70 -- 85@100.40
51@99.70 -- 41@100.40
9@99.60 -- 59@100.40
89@99.60 -- 16@100.50
92@99.50 -- 30@100.60
61@99.50 -- 59@100.70
18@99.50 -- 41@100.80
74@99.40 -- 49@100.80
59@99.40 -- 30@100.90
93@99.40 -- 42@101.30
16@99.30 -- 32@101.60
92@99.10 --
6@99.10 --
54@98.80 --

Book: book-3
      Buy -- Sell
=======================
33@100.10 -- 90@100.20
44@100.00 -- 45@100.30
47@99.90 -- 34@100.30
56@99.90 -- 51@100.30
58@99.90 -- 4@100.40
49@99.80 -- 25@100.40
8@99.80 -- 32@100.50
38@99.80 -- 62@100.50
20@99.80 -- 36@100.50
15@99.70 -- 19@100.60
64@99.70 -- 45@100.60
7@99.70 -- 85@100.70
20@99.60 -- 96@100.70
99@99.60 -- 94@100.90
71@99.50 -- 28@100.90
93@99.40 -- 67@100.90
41@99.20 -- 69@101.10
30@99.20 --
6@99.20 --
75@99.10 --
5@99.00 --
```

```
Book: book-2
      Buy -- Sell
=======================
1@99.90 -- 3@100.00
1@99.80 -- 15@100.00
60@99.80 -- 21@100.20
30@99.70 -- 57@100.20
2@99.70 -- 59@100.20
32@99.70 -- 13@100.20
1@99.70 -- 89@100.20
85@99.70 -- 9@100.20
14@99.70 -- 12@100.20
2@99.60 -- 28@100.20
4@99.60 -- 12@100.30
14@99.60 -- 33@100.30
65@99.60 -- 93@100.30
83@99.50 -- 56@100.40
11@99.50 -- 10@100.60
37@99.50 -- 65@100.60
48@99.40 -- 77@100.60
35@99.40 -- 67@100.60
86@99.10 -- 28@100.60
57@98.70 -- 50@100.60
         -- 69@101.00
         -- 25@101.20

Processing completed at: 2023-11-07 16:30:04.963856
Processing Duration: 54.552 seconds
```

O(N^2) approach (41.311 seconds)

Processing started at: 2023-11-07 16:31:23.330065
book: book-1
Buy -- Sell
==========
49@100.00 -- 33@100.10
97@100.00 -- 20@100.20
38@100.00 -- 37@100.30
63@99.90 -- 7@100.30
10@99.80 -- 97@100.40
40@99.70 -- 4@100.40
42@99.70 -- 85@100.40
51@99.70 -- 41@100.40
9@99.60 -- 59@100.40
89@99.60 -- 16@100.50
92@99.50 -- 30@100.60
61@99.50 -- 59@100.70
18@99.50 -- 41@100.80
74@99.40 -- 49@100.80
59@99.40 -- 30@100.90
93@99.40 -- 42@101.30
16@99.30 -- 32@101.60
92@99.10 --
6@99.10 --
54@98.80 --

book: book-3
Buy -- Sell
==========
33@100.10 -- 90@100.20
44@100.00 -- 45@100.30
47@99.90 -- 34@100.30
56@99.90 -- 51@100.30
58@99.90 -- 4@100.40
49@99.80 -- 25@100.40
8@99.80 -- 32@100.50
38@99.80 -- 62@100.50
20@99.80 -- 36@100.50
15@99.70 -- 19@100.60
64@99.70 -- 45@100.60
7@99.70 -- 85@100.70
20@99.60 -- 96@100.70
99@99.60 -- 94@100.90
71@99.50 -- 28@100.90
93@99.40 -- 67@100.90
41@99.20 -- 69@101.10
30@99.20 --
6@99.20 --
75@99.10 --
5@99.00 --

```
book: book-2
    Buy -- Sell
=======================
1@99.90 -- 3@100.00
1@99.80 -- 15@100.00
60@99.80 -- 21@100.20
30@99.70 -- 57@100.20
2@99.70 -- 59@100.20
32@99.70 -- 13@100.20
1@99.70 -- 89@100.20
85@99.70 -- 9@100.20
14@99.70 -- 12@100.20
2@99.60 -- 28@100.20
4@99.60 -- 12@100.30
14@99.60 -- 33@100.30
65@99.60 -- 93@100.30
83@99.50 -- 56@100.40
11@99.50 -- 10@100.60
37@99.50 -- 65@100.60
48@99.40 -- 77@100.60
35@99.40 -- 67@100.60
86@99.10 -- 28@100.60
57@98.70 -- 50@100.60
                -- 69@101.00
                 -- 25@101.20

Processing completed at: 2023-11-07 16:32:04.640984
Processing Duration: 41.311 seconds
```

Reflect and comment on data consistency if a multithreaded approach is used for processing the order flow.

Using a multithreaded approach to process the order flow can significantly improve performance, particularly if the order flow is heavy and the system needs to handle a high throughput of orders. However, this introduces complexity regarding data consistency, as multiple threads may try to access and modify the order books

simultaneously. Here's how data consistency issues may arise and some ways to handle them:

Race Conditions: If two threads try to update the same order book at the same time, you could end up with a race condition. For example, Thread A might be adding an order while Thread B is deleting another order in the same book. Depending on which operation completes first, the final state of the order book could be different.

Solutions:

Locks/Mutexes: Use locks to ensure that only one thread can modify an order book at a time. This approach, while simple, can become a bottleneck and potentially negate the benefits of multithreading if not managed carefully.

Atomic Operations: Some operations can be made atomic, such as updating the volume of an order. Atomic operations ensure that intermediate states are not visible to other threads, and the operation completes without interruption.

Optimistic Concurrency Control: Use versioning for order books. Before committing any change, check if the version has been altered by another thread since it was read. If it has, the operation can be retried.

Deadlocks: When using locks, there's a risk of deadlocks, where two or more threads are each waiting for the other to release a lock before they can proceed.

Solutions:

Lock Ordering: Impose a strict order in which locks must be acquired.

Lock Timeout: Implement a timeout when attempting to acquire a lock. If the lock isn't acquired in the given time frame, the operation is aborted and can be retried later.

Deadlock Detection: Have a mechanism to detect deadlocks and a strategy to recover from them, such as aborting one of the transactions and retrying it.

Consistency Models: If you relax the consistency requirements, you can allow for more concurrency. This is a trade-off where you allow the system state to be slightly "stale" in exchange for higher throughput.

Solutions:

Eventual Consistency: Accept that the order books might not be perfectly in sync at all times, but will become consistent eventually.

Read-Write Locks: Use separate locks for reading and writing operations. Multiple threads can read at the same time, but only one can write.

Thread-Safe Data Structures: Instead of using regular data structures, use ones that are designed to be safe for concurrent access by multiple threads.

Solutions:

Concurrent Collections: Use thread-safe variants of collections, which are provided by many programming languages in their standard libraries.

Testing for Concurrency Issues: It's notoriously difficult to test multithreaded applications because issues may only appear under certain conditions or loads.

Solutions:

Stress Testing: Run the application under heavy loads for extended periods to try to trigger any concurrency issues.

Static Analysis Tools: Use tools that can analyze the code for potential concurrency issues.

Transaction Processing: You could also use transactional memory or database transactions to handle concurrency.

Solutions:

Software Transactional Memory (STM): Use STM to ensure that changes to shared data are atomic and isolated from other transactions.

Database Transactions: If the order book is maintained in a database, leverage the database's transaction capabilities to ensure consistency.

In summary, while a multithreaded approach can improve performance, it requires careful design to maintain data

consistency. It's essential to choose the right strategies based on the specific requirements and characteristics of the order processing system, such as the expected load and the acceptable level of consistency.