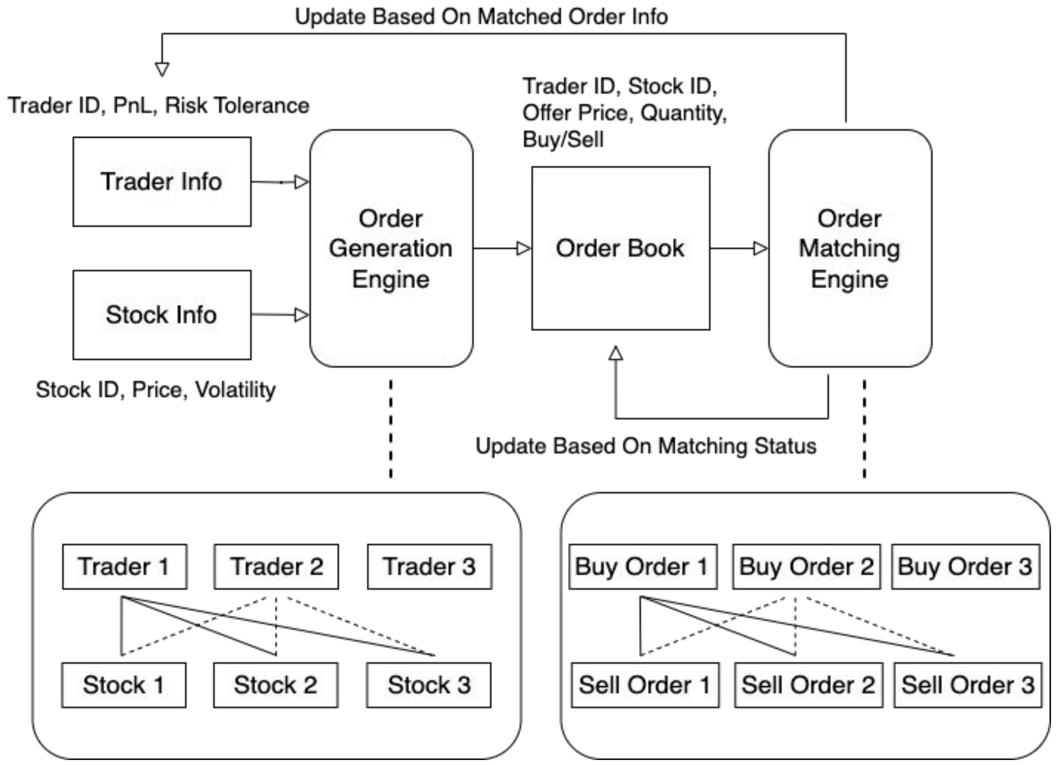
High Performance Stock Trading System - System Design

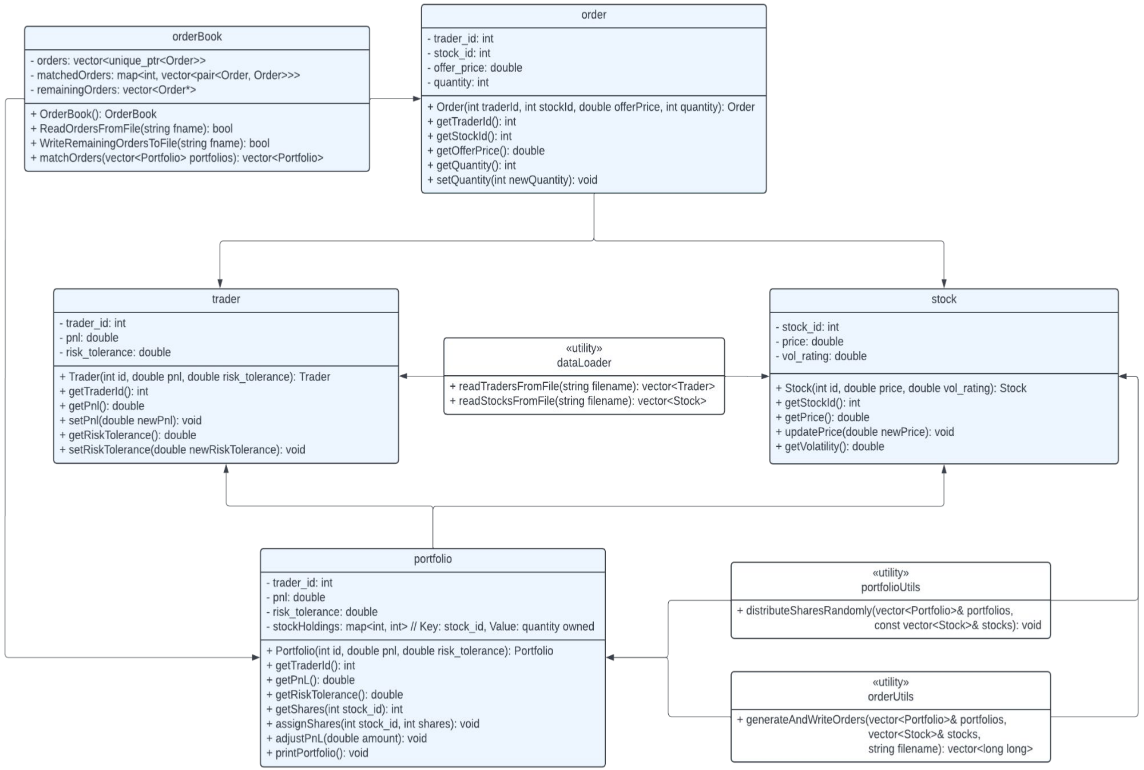
CS205 Final Project - Group 8
Aaron Li, April Zhang, Catherine Gai, Susannah Su, Yixuan Qiu

Trading System Workflow

1. Read trader and stock information
2. Generate orders
3. Write orders to order book
4. Match orders and execute trades
5. Update trader portfolios and balances



Sequential Baseline



```
orderBook
- orders: vector<unique_ptr<Order>>
- matchedOrders: map<int, vector<pair<Order, Order>>>
- remainingOrders: vector<Order*>
+ OrderBook(): OrderBook
+ ReadOrdersFromFile(string fname): bool
+ WriteRemainingOrdersToFile(string fname): bool
+ matchOrders(vector<Portfolio> portfolios): vector<Portfolio>
```

```
order
- trader_id: int
- stock_id: int
- offer_price: double
- quantity: int
+ Order(int traderId, int stockId, double offerPrice, int quantity): Order
+ getTraderId(): int
+ getStockId(): int
+ getOfferPrice(): double
+ getQuantity(): int
+ setQuantity(int newQuantity): void
```

```
trader
- trader_id: int
- pnl: double
- risk_tolerance: double
+ Trader(int id, double pnl, double risk_tolerance): Trader
+ getTraderId(): int
+ getPnl(): double
+ setPnl(double newPnl): void
+ getRiskTolerance(): double
+ setRiskTolerance(double newRiskTolerance): void
```

```
«utility»
dataLoader
+ readTradersFromFile(string filename): vector<Trader>
+ readStocksFromFile(string filename): vector<Stock>
```

```
stock
- stock_id: int
- price: double
- vol_rating: double
+ Stock(int id, double price, double vol_rating): Stock
+ getStockId(): int
+ getPrice(): double
+ updatePrice(double newPrice): void
+ getVolatility(): double
```

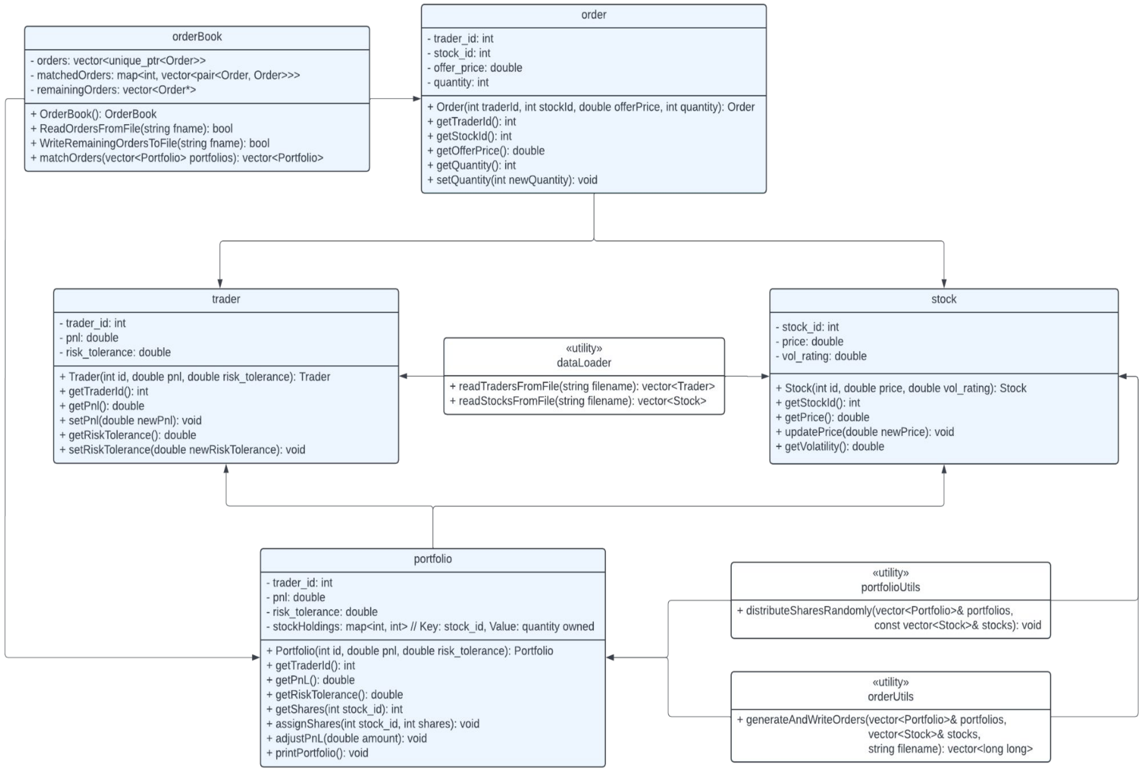
```
portfolio
- trader_id: int
- pnl: double
- risk_tolerance: double
- stockHoldings: map<int, int> // Key: stock_id, Value: quantity owned
+ Portfolio(int id, double pnl, double risk_tolerance): Portfolio
+ getTraderId(): int
+ getPnL(): double
+ getRiskTolerance(): double
+ getShares(int stock_id): int
+ assignShares(int stock_id, int shares): void
+ adjustPnL(double amount): void
+ printPortfolio(): void
```

```
«utility»
portfolioUtils
+ distributeSharesRandomly(vector<Portfolio>& portfolios,
                           const vector<Stock>& stocks): void
```

```
«utility»
orderUtils
+ generateAndWriteOrders(vector<Portfolio>& portfolios,
                         vector<Stock>& stocks,
                         string filename): vector<long long>
```

- **Simplified** after Ignacio's feedback, only kept main functionality
- Orders are now **sorted based on stock_id** → possible to perform **parallelization for each stock for matching, avoid conflicts**
- Added a portfolio class for recording each trader's stock holdings in a map<int, int> (key: stock_id, value: quantity owned)
- Order matching occurs **after all orders have been placed**, similar to a **batch processing** system.

Sequential Baseline


orderBook
- orders: vector<unique_ptr<Order>>
- matchedOrders: map<int, vector<pair<Order, Order>>>
- remainingOrders: vector<Order*>
+ OrderBook(): OrderBook
+ ReadOrdersFromFile(string fname): bool
+ WriteRemainingOrdersToFile(string fname): bool
+ matchOrders(vector<Portfolio> portfolios): vector<Portfolio>
order
- trader_id: int
- stock_id: int
- offer_price: double
- quantity: int
+ Order(int traderId, int stockId, double offerPrice, int quantity): Order
+ getTraderId(): int
+ getStockId(): int
+ getOfferPrice(): double
+ getQuantity(): int
+ setQuantity(int newQuantity): void
trader
- trader_id: int
- pnl: double
- risk_tolerance: double
+ Trader(int id, double pnl, double risk_tolerance): Trader
+ getTraderId(): int
+ getPnl(): double
+ setPnl(double newPnl): void
+ getRiskTolerance(): double
+ setRiskTolerance(double newRiskTolerance): void
«utility»
dataLoader
+ readTradersFromFile(string filename): vector<Trader>
+ readStocksFromFile(string filename): vector<Stock>
stock
- stock_id: int
- price: double
- vol_rating: double
+ Stock(int id, double price, double vol_rating): Stock
+ getStockId(): int
+ getPrice(): double
+ updatePrice(double newPrice): void
+ getVolatility(): double
portfolio
- trader_id: int
- pnl: double
- risk_tolerance: double
- stockHoldings: map<int, int> // Key: stock_id, Value: quantity owned
+ Portfolio(int id, double pnl, double risk_tolerance): Portfolio
+ getTraderId(): int
+ getPnL(): double
+ getRiskTolerance(): double
+ getShares(int stock_id): int
+ assignShares(int stock_id, int shares): void
+ adjustPnL(double amount): void
+ printPortfolio(): void
«utility»
portfolioUtils
+ distributeSharesRandomly(vector<Portfolio>& portfolios, const vector<Stock>& stocks): void
«utility»
orderUtils
+ generateAndWriteOrders(vector<Portfolio>& portfolios, vector<Stock>& stocks, string filename): vector<long long>

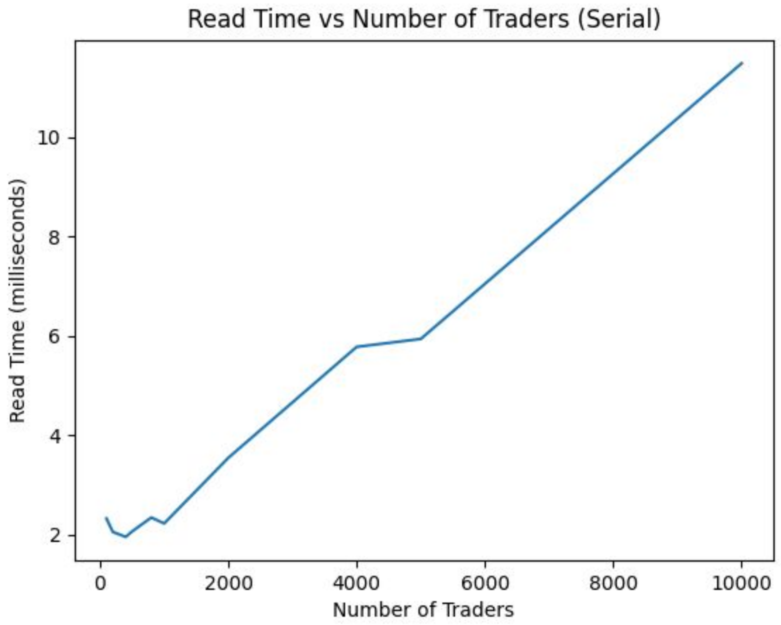
- **Order generating logic:** iterate through each portfolio, retrieve base price and volatility of the stock and trader's risk tolerance → generate order offer price and order quantity based on risk & volatility, random selection for buy or sell action → output generated orders to a file
- **Order matching logic:** buy (- quantity) and sell (+ quantity) **orders are sorted by stock ID**. Matches are sought by comparing offer prices and quantities **within each stock ID category**.
  - Successful matches: Adjust order quantities & update the traders' profit and loss. Fully matched orders are removed, while partial or unmatched orders are recorded. The system tallies all matches, **maintains unmatched orders, and updates portfolios accordingly**.

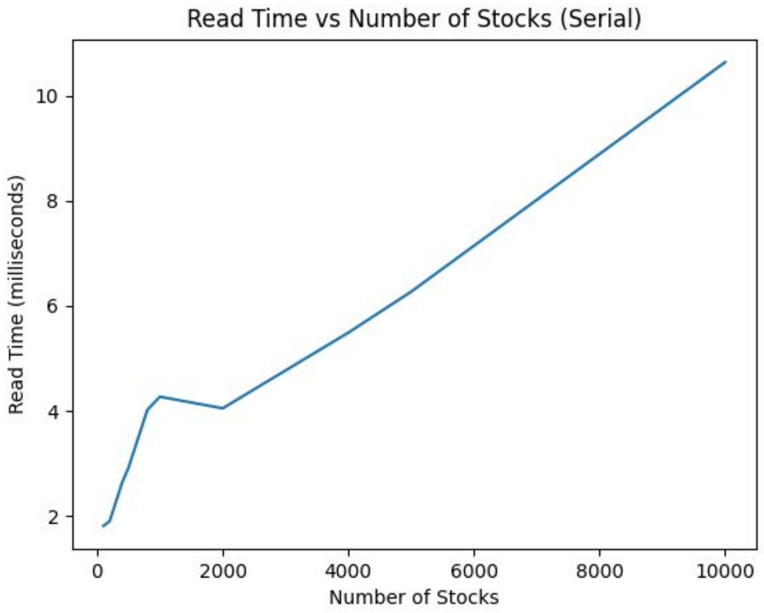
Profiling for Order Generation

Order generation procedure:

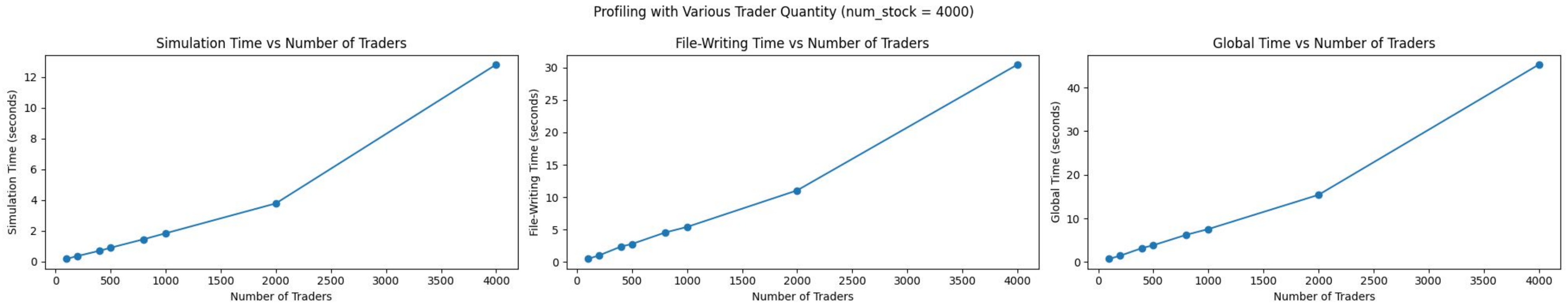
- Iterate through all traders, and for each trader, all possible stocks.
- No trading activity (including buy *and* sell) if trader has no shareable stock.
- Randomly generate trading quantity and buy/ sell activity.
- Write relevant information into order CSV file.

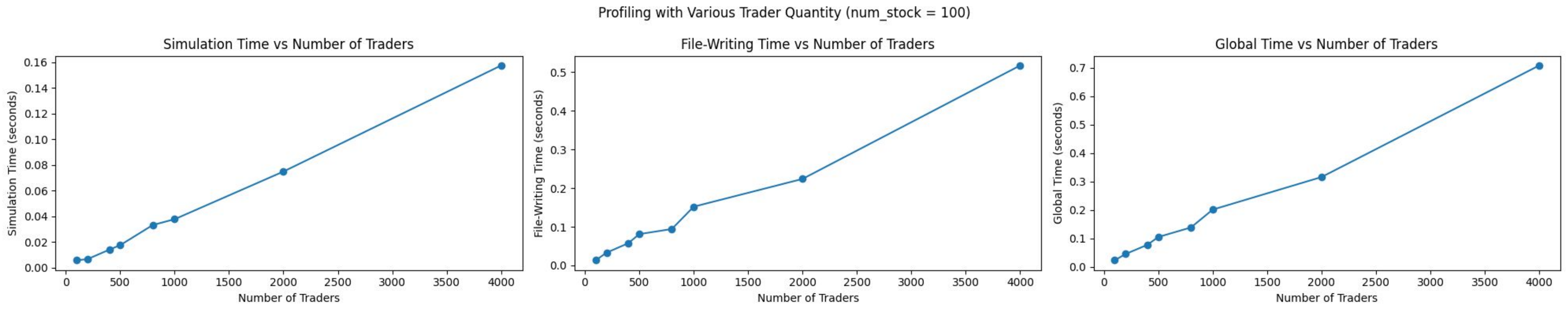
Read Time vs Number of Traders/Stocks (Sequential)



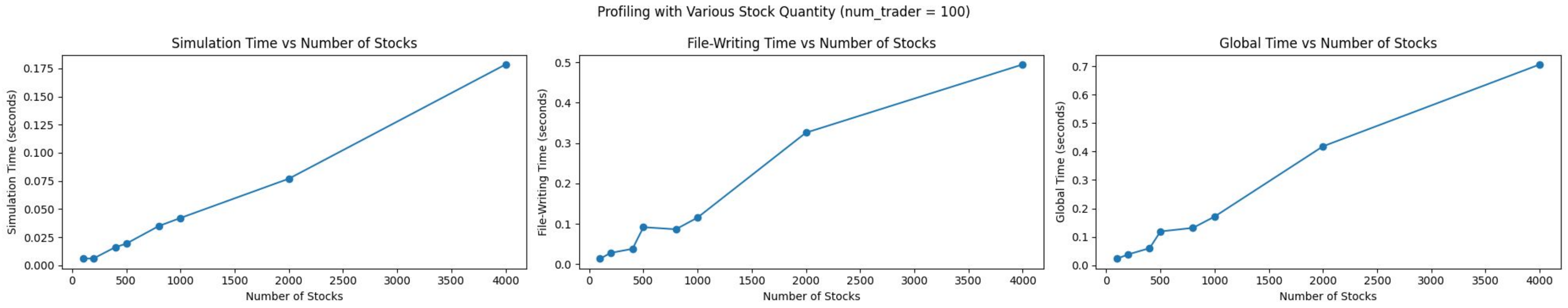


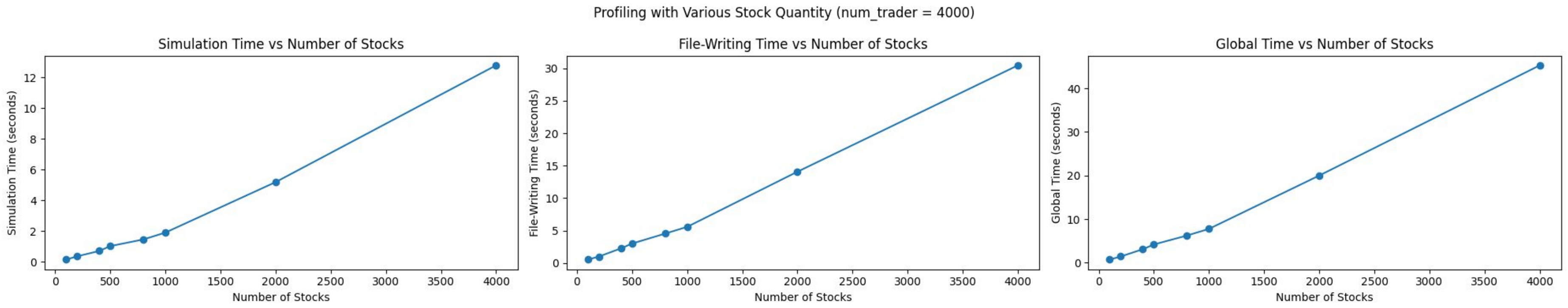
Runtime Plots w.r.t Traders





Runtime Plots w.r.t Stocks



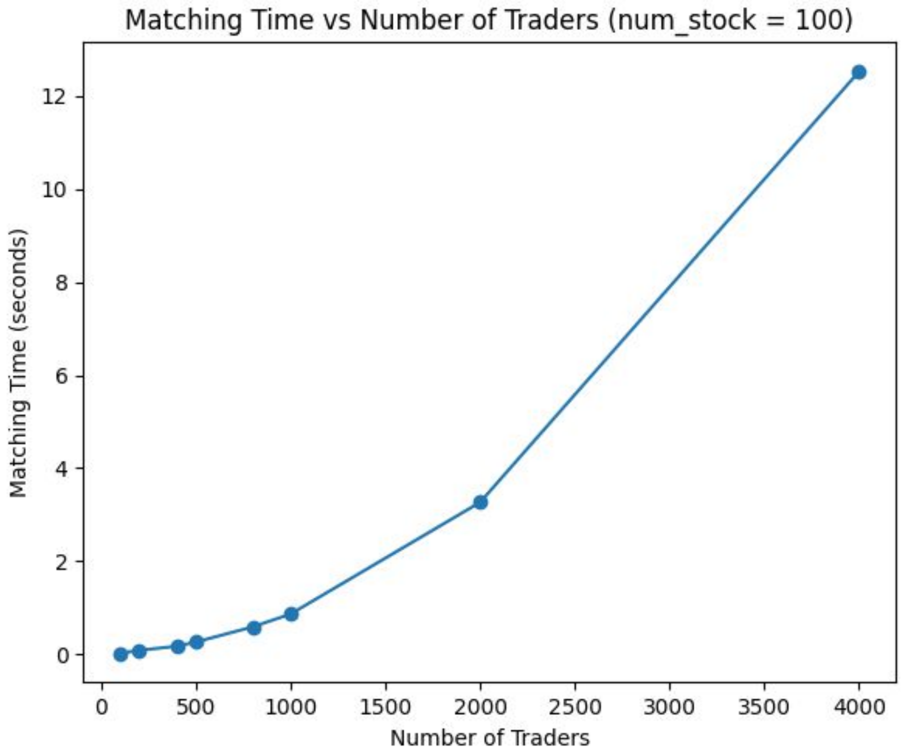


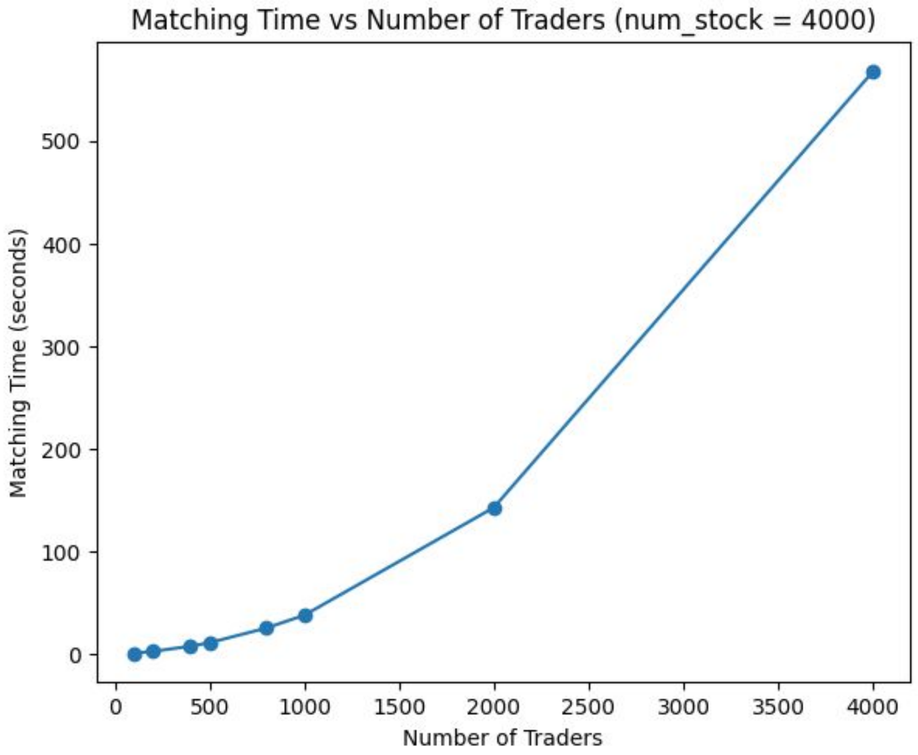
Profiling for Order Matching

Order matching procedure:

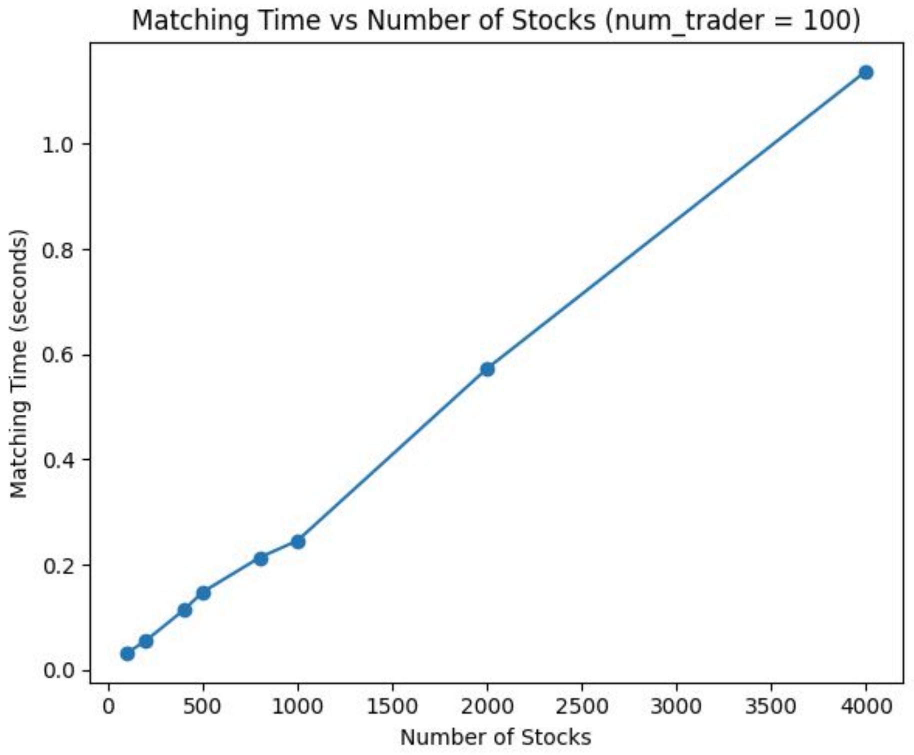
- Load orders from files, iterate over all orders.
- Categorize orders into buy (- quantity) and sell (+ quantity) based on stock ID.
- For each stock ID, attempt to match buy and sell orders by:
    - Match if the buy order price >= sell order price.
    - Determining if a full or partial match is possible based on order quantities.
- Adjust quantities of matched orders accordingly.
- Update the profit and loss (PnL) for both buyers and sellers based on the executed trades.
- Collect and list unmatched or partially matched orders as remaining orders.

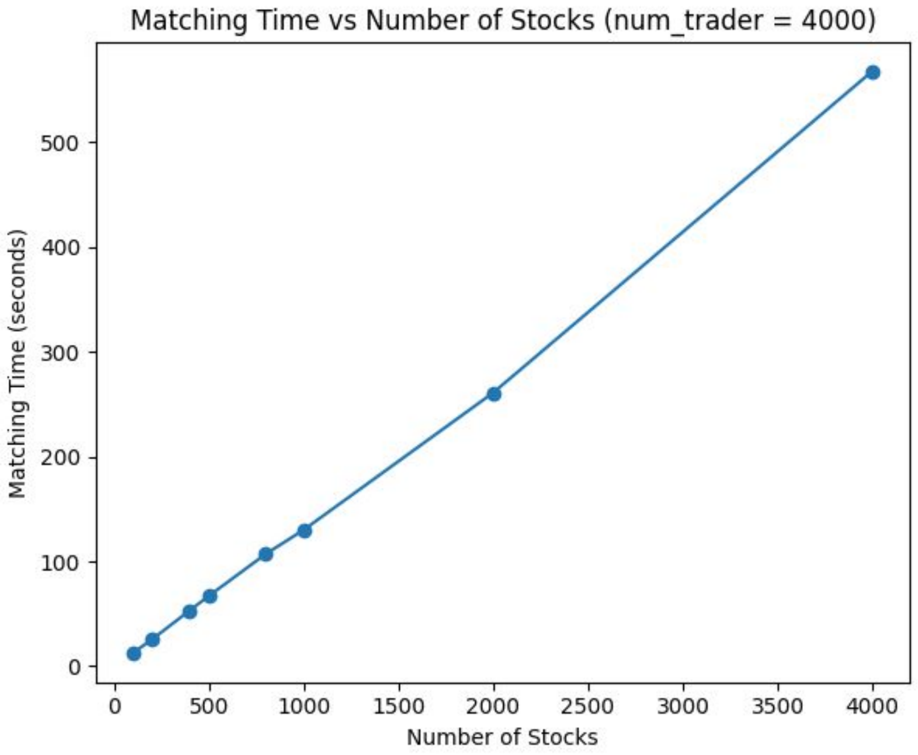
Runtime Plots w.r.t Traders





Runtime Plots w.r.t Stocks





Roofline Analysis

CPU: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz

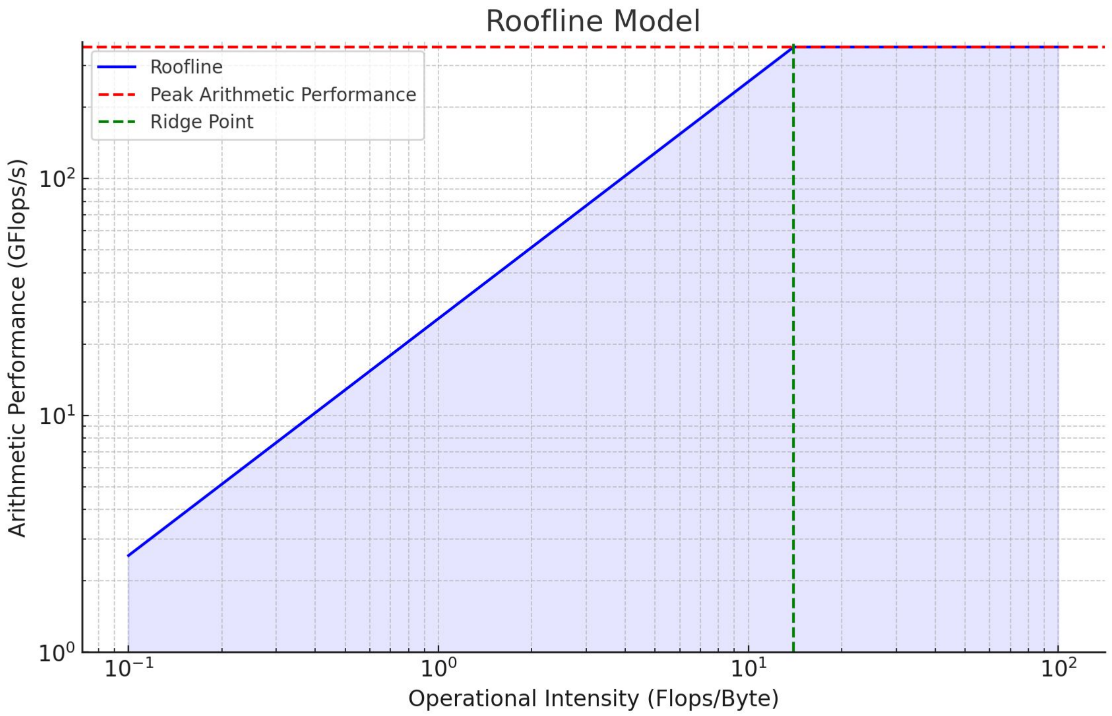
- Nominal peak arithmetic performance

  = base clock rate * # cores * SIMD lanes * Flops per cycle

  = 2.8 GHz * 4 * (512 bit / 64 bit) * 4 Flops per cycle

  = 358.4 GFlops/s

- Nominal peak memory performance: 25.6 GB/s



Roofline Analysis (Sequential)

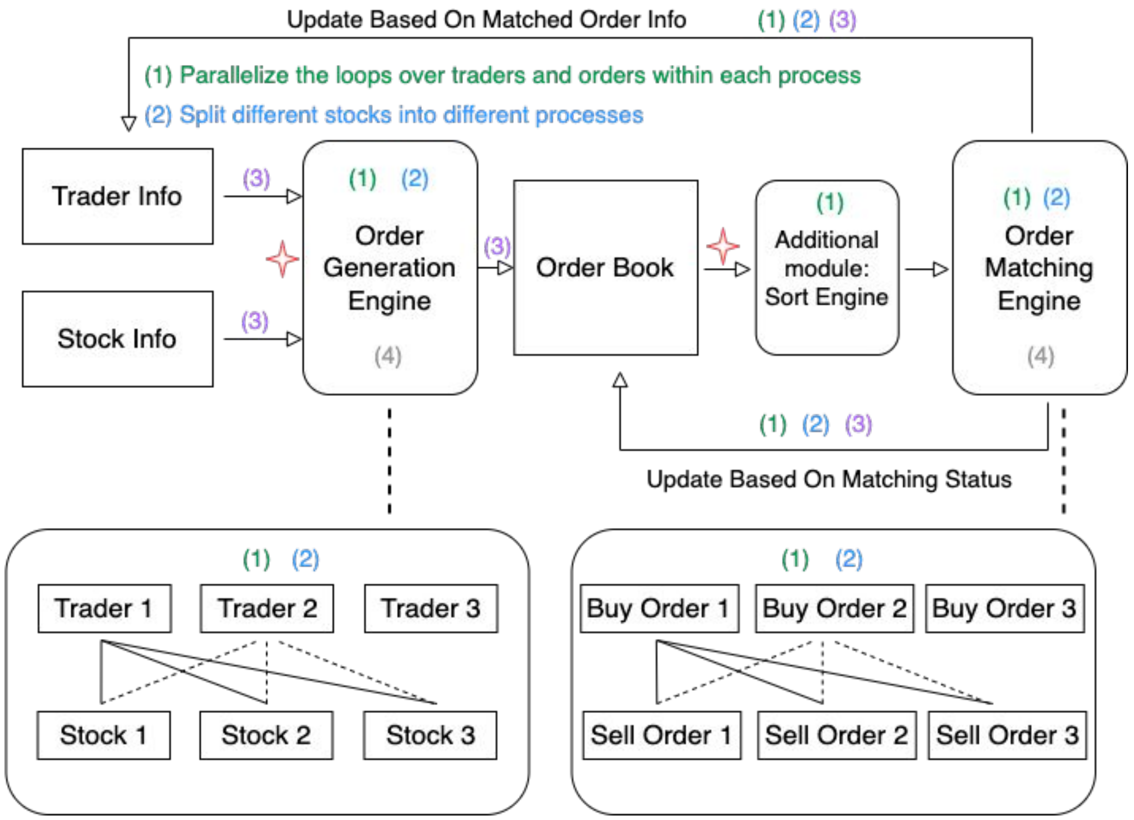
- Read/Write trader, stock, and order data:
  - No computation required, so completely memory bounded
- Order Generation:
  - For each fixed-size loaded (trader, stock) pair, we need to compute a fixed number of fields for the generated order, so it’s relatively compute-memory balanced
- Order Matching:
  - Naive iterative matching incurs a lot of cache misses, which dominates over the arithmetic comparisons and computations for each pair of buy and sell orders, so it’s currently memory bounded

Parallelization Design

- Combination of shared and distributed memory models
- Relaxed Synchronization helps hide latency
- Additional global sort engine (which is also parallelizable) improves matching efficiency and reduces the number of unmatched orders

Parallelization Techniques:

(1) Shared memory model: OpenMP
(2) Distributed memory model: MPI
(3) Distributed Memory model: MPI I/O
(4) Vectorization
✦ Synchronization point



Update Based On Matched Order Info (1) (2) (3)
(1) Parallelize the loops over traders and orders within each process
(2) Split different stocks into different processes
Trader Info
Stock Info
(3)
(3)
(1) (2) Order Generation Engine (4)
(3)
Order Book
(1) Additional module: Sort Engine
(1) (2) Order Matching Engine (4)
(1) (2) (3)
Update Based On Matching Status
(1) (2)
Trader 1
Trader 2
Trader 3
Stock 1
Stock 2
Stock 3
(1) (2)
Buy Order 1
Buy Order 2
Buy Order 3
Sell Order 1
Sell Order 2
Sell Order 3

Thank you!

CS205 Final Project - Group 8
Aaron Li, April Zhang, Catherine Gai, Susannah Su, Yixuan Qiu