High-Performance Stock Trading System

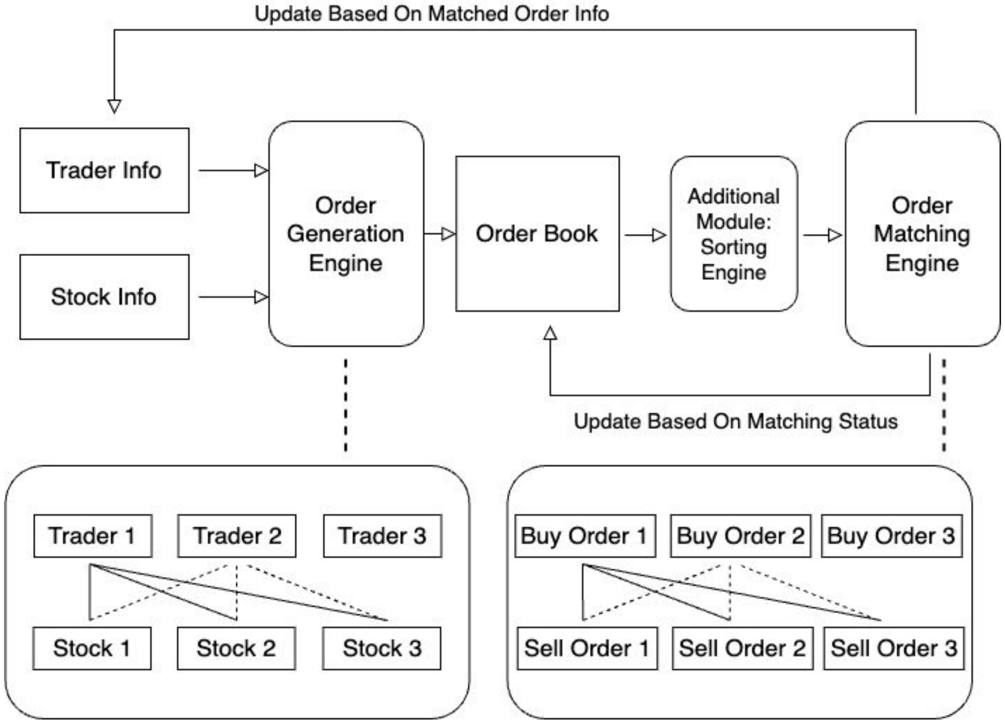
Final Presentation

CS205 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 Bottleneck

- Order Generation
  - Iterate through every stock, and for each stock, iterate through every trader
  - Time complexity: O(NM), N = number of orders, M = number of traders
- Order Matching
  - Need to exhaustively compare each possible pair of buy and sell orders
  - Time complexity is O(P^2), P = number of orders
- Potential Space of Performance Improvement
  - Order creation can be done individually for each stock
  - Order matching can be done separately and in parallel for individual stocks.
  - This would also benefit I/O efficiency as we read from stock files and write back generated orders

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

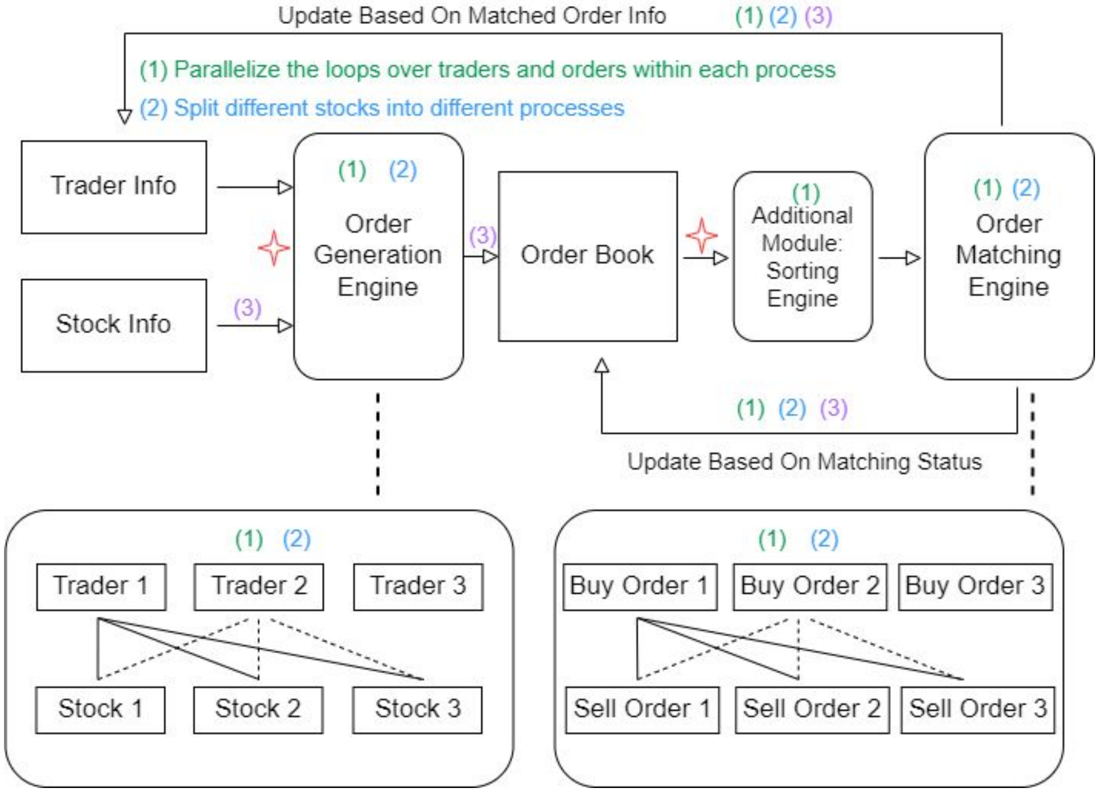
Parallelization Techniques:

(1) Shared memory model: OpenMP

(2) Distributed memory model: MPI

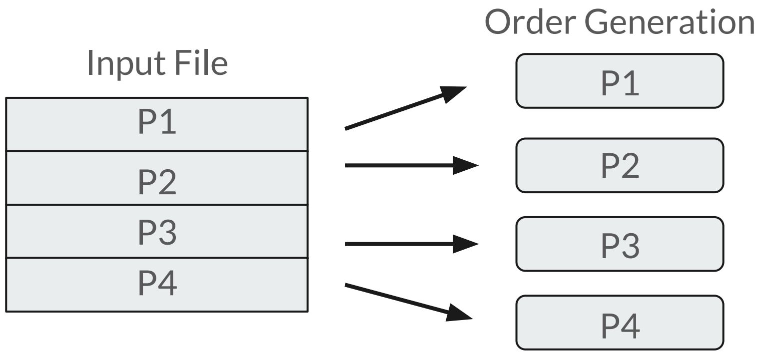
(3) Distributed Memory model: MPI I/O

✧ Synchronization point



Parallelization Implementation - MPI I/O

- Challenge: each line represents a single stock/trader/order information, and the *getline()* functionality is inherently sequential
- Solution: divide the file size by the number of processes, and assign each process a contiguous chunk of bytes using offsets
  - Need to handle a few edge cases when the break point is in the middle of a line
  - Within each chunk, we can safely use *getline()*



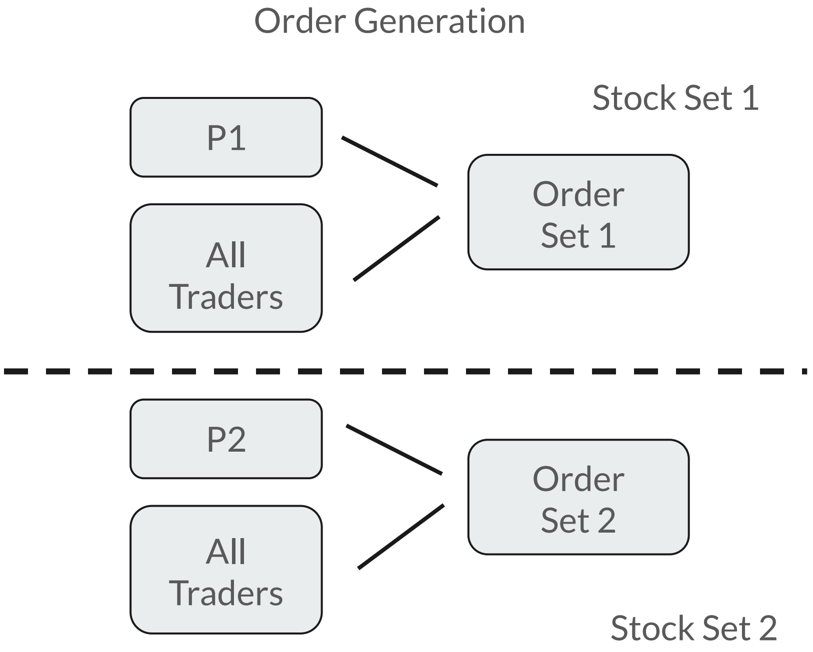
This step initiates the distributed memory model (over stocks) throughout the system

```
// Read a bit extra to not cut off lines in the middle
end_idx += (rank != world_size - 1) ? 20 : 0;  // Additional bytes, adjust as necessary

// Adjust start to skip partial initial line unless at the beginning of the file
std::string data_str(buffer);
if (start_idx != 0) {
    size_t first_newline = data_str.find('\n');
    data_str = data_str.substr(first_newline + 1);
}
```

Parallelization Implementation - Order Generation

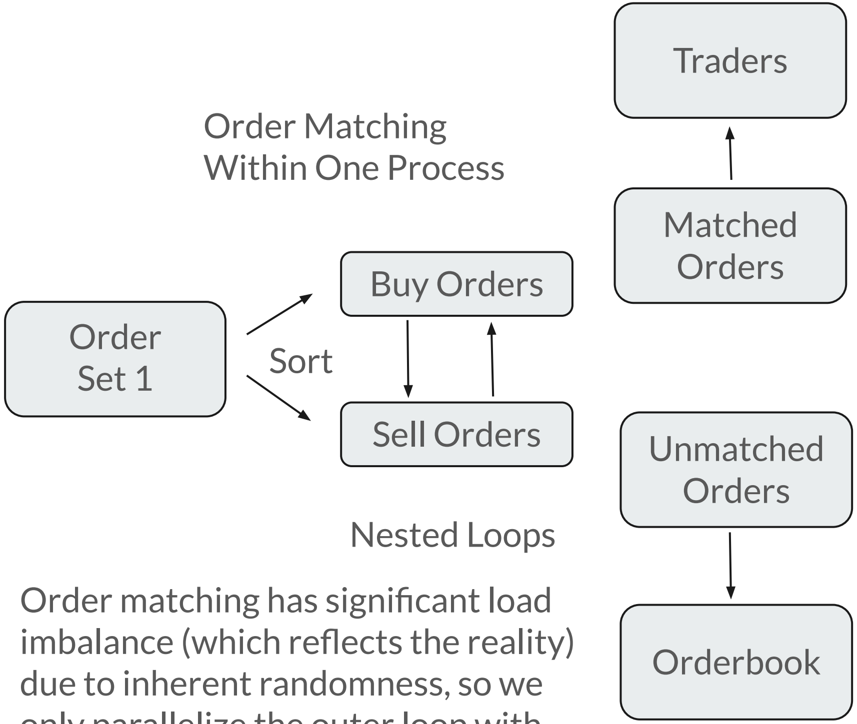
- We need to iterate over all pairs of (stock, trader), so all trader info must be shared among all processes
- We can make a copy of all trader info for each process because it's read-only during the order generation process
- Within each process, we can safely exploit thread-level parallelism using OpenMP
- MPI I/O allows all processes write to the same orderbook file simultaneously
    - Each process needs to know the offsets of all other processes




```
// Gather sizes of each local data to calculate offsets
int* sizes = new int[world_size];
MPI_Allgather(&dataSize, 1, MPI_INT, sizes, 1, MPI_INT, MPI_COMM_WORLD);
```

Parallelization Implementation - Order Matching

- Now each process handles all orders related to a subset of stocks, and still has a copy of all trader information
- OpenMP is used extensively
  - Merge Sort
  - Buyer/Seller Matching
- MPI I/O is used for writing back unmatched orders
- MPI communication is used for aggregating the changes in trader info (i.e. PnL) across all processes (only once after all matchings are done)



Order matching has significant load imbalance (which reflects the reality) due to inherent randomness, so we only parallelize the outer loop with dynamic scheduling given limited number of threads

Parallelization Implementation - Miscellaneous

- Changed the data structure of orders from dynamic vectors to preallocated arrays, as the large amount of vector-append operations cannot be efficiently parallelized

```
class OrderBook {
public:
        OrderBook(int max_num_orders);
        Order* orders;
        int max_num_orders;
        bool ReadOrdersFromFile(std::string fname);
        bool WriteRemainingOrdersToFile(std::string fname);
        void addOrder(int idx, int trader_id, int stock_id, double offer_price, int quantity);
        Portfolio* matchOrders(Portfolio* portfolios, int num_portfolios);
```

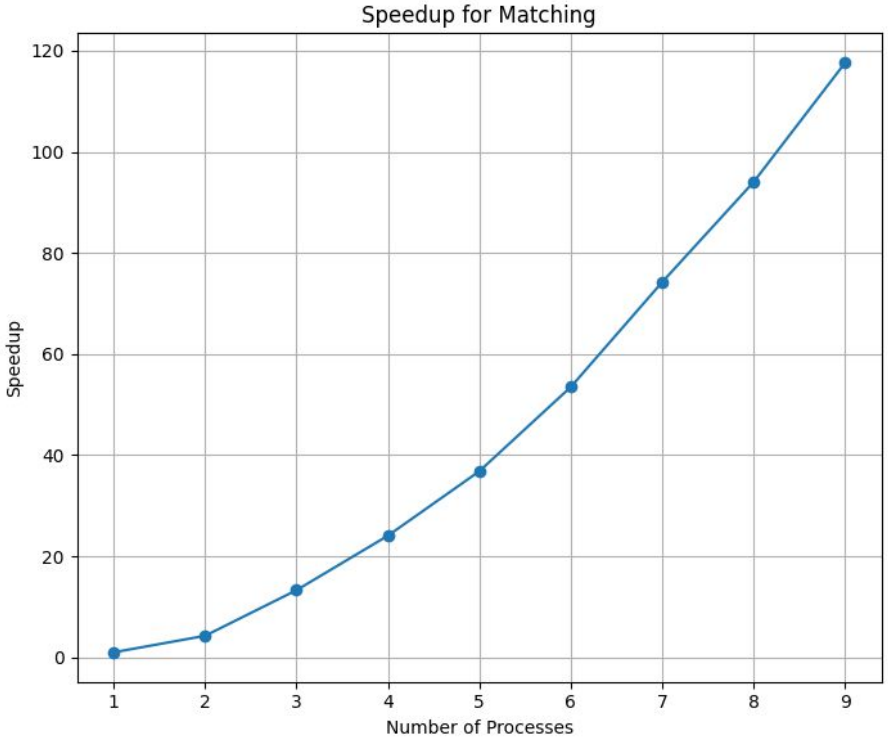
- Pass the entire orderbook from order generation to matching to reduce file-based I/O

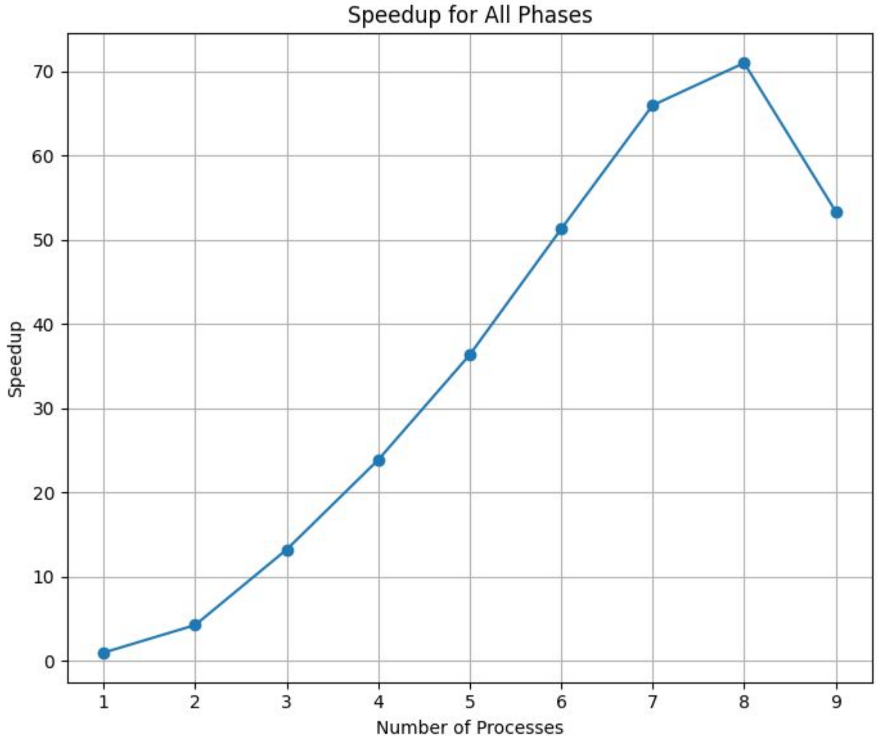
```
if (isBuyOrder) {
    order_book.addOrder(idx, portfolio.getTraderId(),
        stock.getStockId(), offerPrice, -std::abs(quantity));
```

Approach to Scaling Analysis

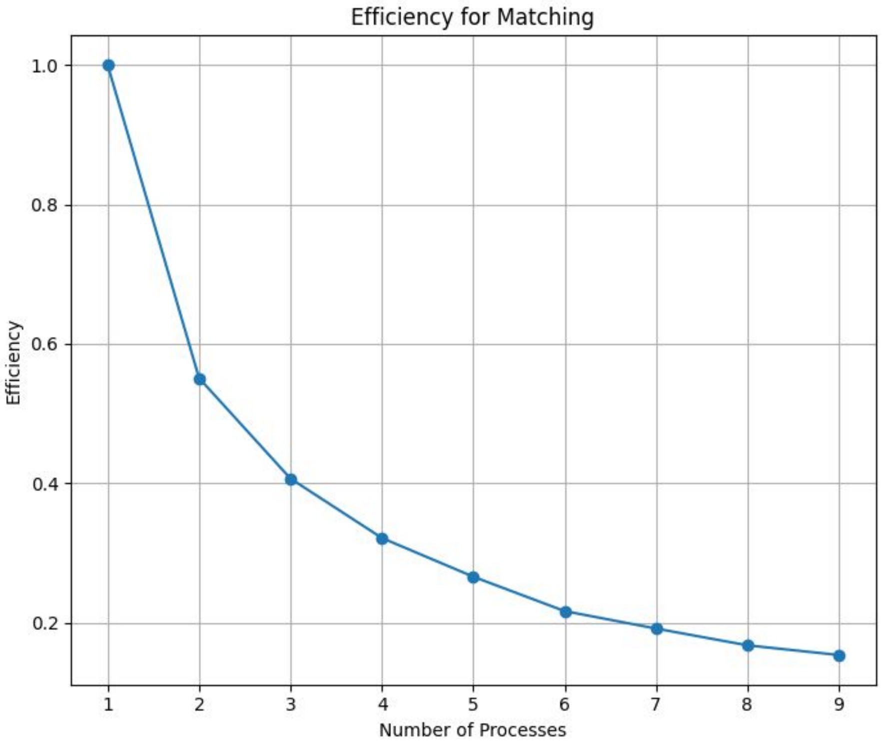
- Strong scaling:
  - 500 stocks, 10,000 traders
  - 1 to 9 processes
  - 4 threads
- Weak scaling:
  - # processes proportional to size of stocks and traders combined
  - 40 kB = 1 process with 4 threads
  - 500 stocks, change traders (2000 to 20,000)
- Threading:
  - 500 stocks, 10,000 traders
  - 1 process, 1 to 36 threads

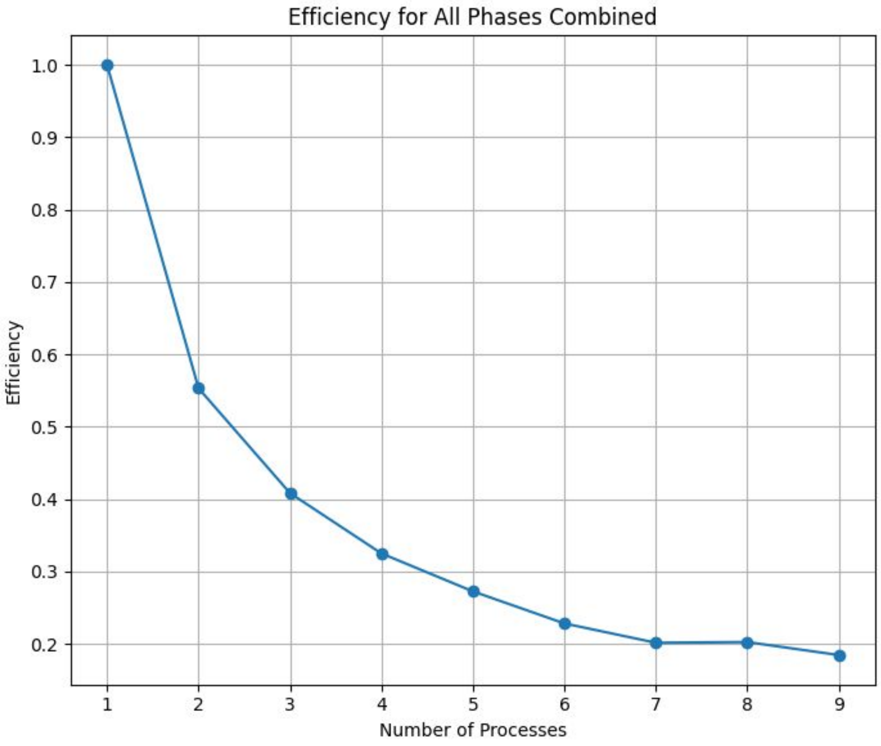
Strong Scaling Results



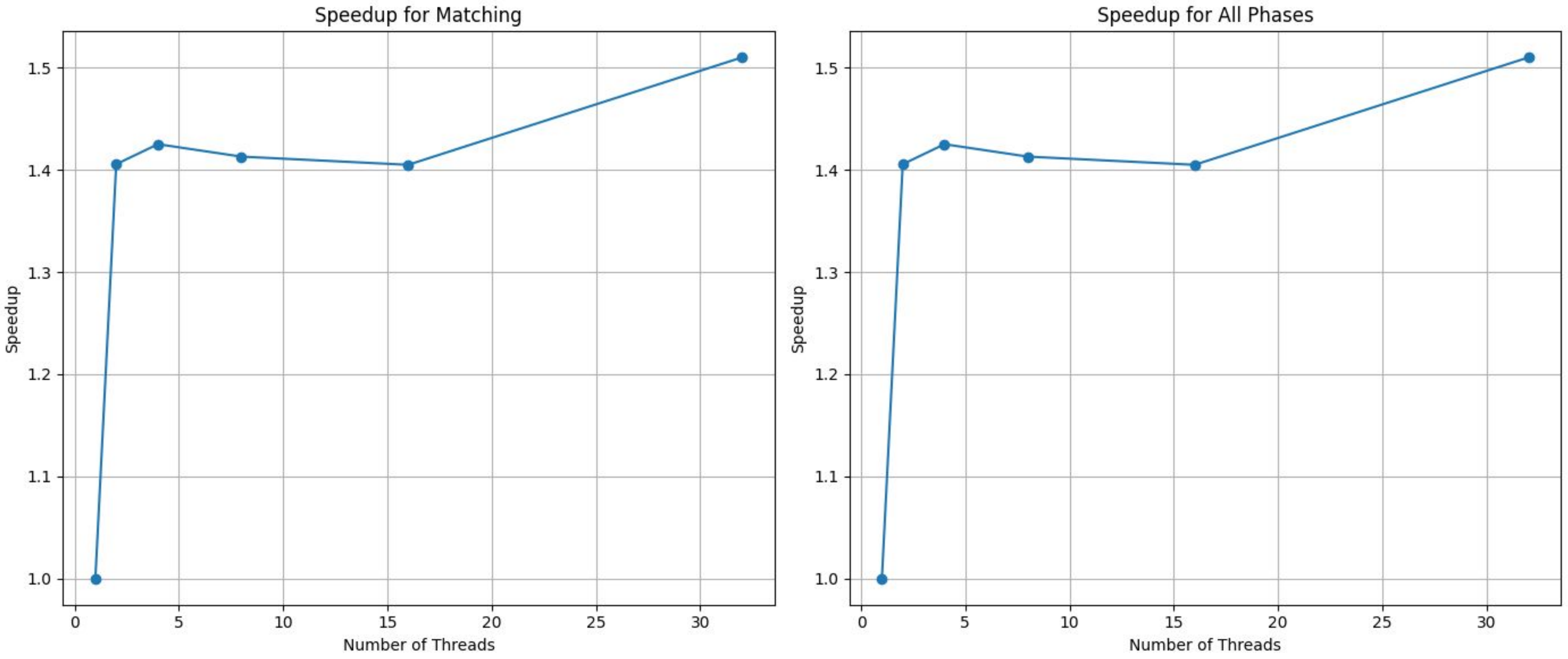


Weak Scaling Results

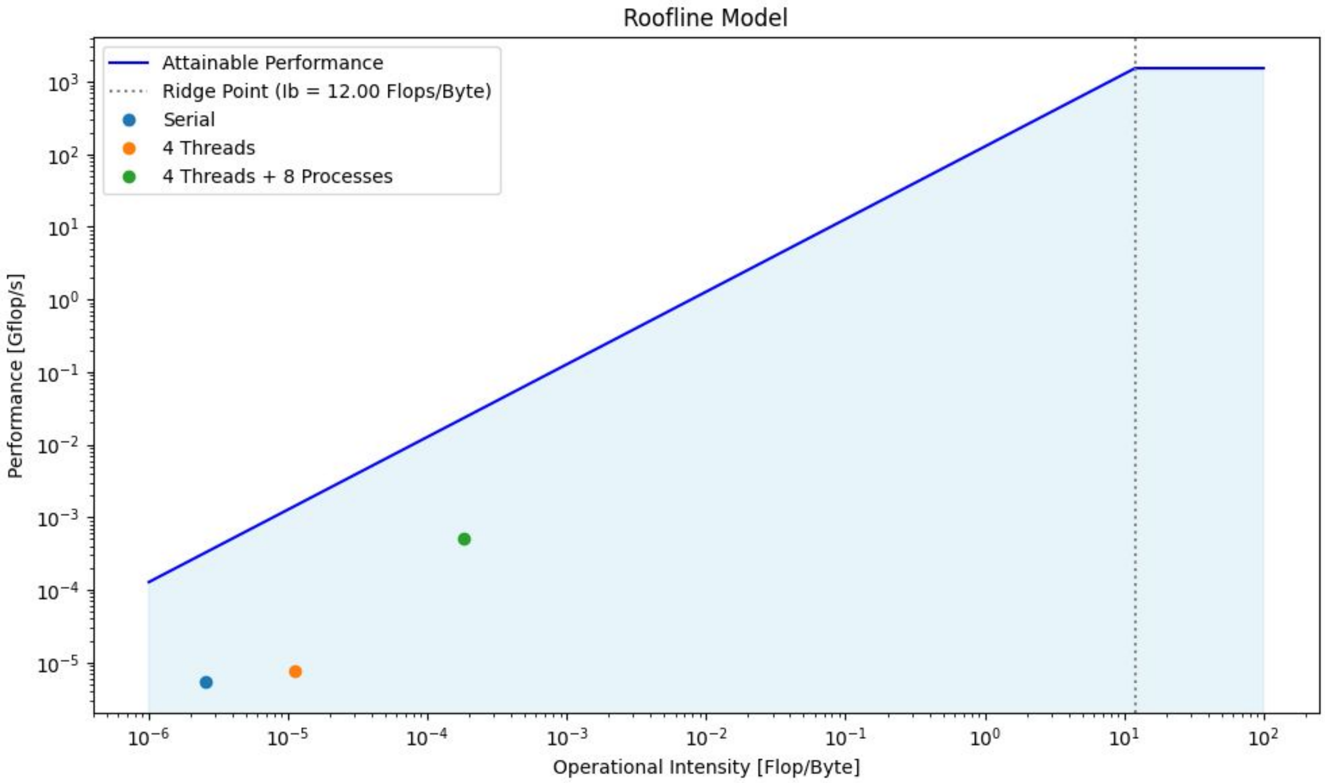




Threading Performance



Roofline Analysis



Future Work

- Order Matching with Timestamping
  - Implement realistic order matching strategies — consider order timestamps
  - Explore techniques such as price-time priority
- Investigate techniques for better scalability and performance
  - e.g., order book partitioning, replication, or hierarchical organization
- Integration with financial libraries/frameworks for more realistic simulations
  - e.g., libraries for market data processing, risk management, or advanced order matching algorithms
- Real-Time Order Matching

Thank you!