A Proposal for High-Performance Parallelized Stock Trading System

Aaron Li, April Zhang, Catherine Gai, Susannah Su, and Yixuan Qiu

Harvard University

Abstract

In this paper, we present an exploration of parallel computing techniques applied to a stock trading simulation system, aiming to enhance computational efficiency and deepen insights into financial market dynamics. We begin by elaborating on the algorithms and discussing the limitations of traditional sequential methodologies. We then outline code parallelization strategies employed, including the use of OpenMP and MPI to parallelize critical components such as order generation and order matching. Roofline and scaling analyses are presented to demonstrate the effectiveness and scalability of our parallelized simulation framework. Through this research endeavor, we contribute to the intersection of computational science and finance, offering insights into practical applications in financial markets. The code is available at `https://code.harvard.edu/CS205/team08_2024`.

1. Background and Significance

In the contemporary financial landscape, simulation models play an important role in capturing the ever-evolving market dynamics [1, 2]. They help us understand market behaviors, evaluate trading strategies, and understand the intricacies of financial systems. However, the efficacy of traditional sequential simulation methodologies in addressing the computational demands inherent to complex financial environments remains limited.

Faced with these challenges, using parallel computing is a promising way to make financial simulations faster and more scalable. By using parallelization techniques to distribute computational tasks across multiple processing units, the simulation processes can be accelerated. This not only optimizes computational performance but also deepens our understanding of financial systems and their complexities.

The project aims to parallelize critical components, such as decision-making and order processing, within a stock trading simulation framework, allowing for a broader exploration of market scenarios and enhancing our understanding of market behaviors and trading strategies. From an academic standpoint, the project contributes to the intersection of computational science and finance.

2. Scientific Goals and Objectives

The scientific goal of this project is to enhance the computational efficiency of stock trading systems through parallelization and to deepen our understanding of financial market dynamics through the development and application of parallelized stock trading simulations.

The project has several key objectives. Firstly, it aims to optimize computational efficiency by employing parallelization techniques to accelerate stock trading simulations, resulting in reduced computation time and improved performance. Secondly, it seeks to use parallelized simulations to model trading strategies, providing insights into the factors influencing trader decisions and market behavior.

Given the complexity and scale of these objectives, the project requires substantial computing resources and access to a High-Performance Computing (HPC) architecture. The computational power provided by an HPC system is essential for executing the intensive simulations and analyses required to achieve the project's scientific goals effectively.

3. Algorithms and Code Parallelization

3.1 Workflows and Algorithms

The trading system encompasses several classes including 1) **Stock**, which has attributes such as stock ID, price, and volatility rating; 2) **Trader**, including trader ID, profit and loss (PnL), and risk tolerance influencing trading behavior; 3) **Order**, representing trading instructions with parameters like trader ID, stock ID, offer price, and quantity; 4) **Portfolio**, detailing a trader's stock holdings and quantities; and 5) **OrderBook**, managing orders and executing buy and sell matches. Additionally, the system incorporates utility functions to facilitate tasks such as data initialization, order generation, and file I/O. These functions streamline the simulation process by automating repetitive tasks and ensuring the availability of realistic data for analysis.

The workflow of the trading system is structured into sequential steps:

1. Initialization: The system initializes by reading trader and stock information including trader IDs, profit and loss, risk tolerance, stock IDs, prices, and volatility ratings.

2. Order Generation: Once initialized, the system generates orders based on the available trader portfolios and stock attributes. This process involves calculating offer prices and order quantities, considering trader risk tolerance and stock volatility. Each order specifies an offer price at which the trader is willing to buy or sell a certain quantity of stock. The order generation engine iterates through each trader and stock, randomly allocating shares to traders until all stock is distributed.

3. Order Matching: Orders are managed by the OrderBook, which maintains a collection of buy and sell orders. The OrderBook attempts to match buy and sell orders based on their details, such as trader IDs, stock IDs, offer prices, and quantities. Specifically, we use merge sort to arrange both buy and sell orders based on quantity to maximize the total number of matched orders. Buy orders are sorted in descending order of quantity, while sell orders are sorted in ascending order of quantity. This sorting helps optimize the matching process by prioritizing larger buy orders and smaller sell orders. Subsequently, an order matching engine is utilized to pair sell orders with buy orders. The matching process considers the offer prices and quantities of buy and sell orders. If the offer price of a sell order is less than or equal to the offer price of a buy order, and there is sufficient quantity to fulfill the transaction, a match is found. The matched orders are then recorded for trade execution. Ultimately, the updated trader and stock information is written back into the OrderBook.

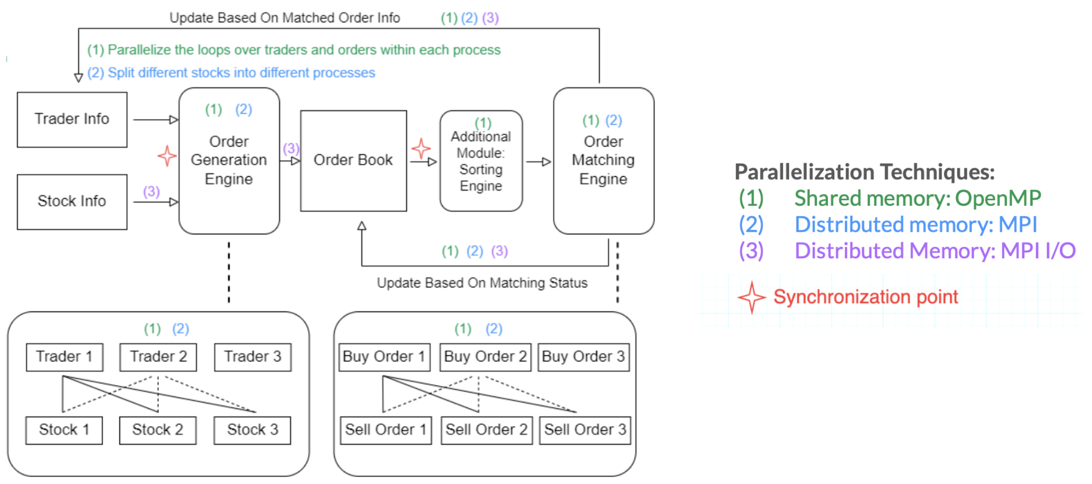
4. Trade Execution: Matched orders result in trade executions, where stocks are exchanged between traders according to the agreed-upon prices and quantities. These trades update the trader portfolios and their respective balances accordingly.

5. Portfolio Update: After trade execution, the system updates trader portfolios to reflect the changes in stock holdings and PnL. Traders may gain or lose capital based on the outcomes of their trades.

We designed and implemented this sequential processing framework from scratch, with some inspirations taken from existing implementations such as `https://github.com/chronoxor/CppTrader`.

3.2 Parallelization

In considering places for parallelization, the system uses both OpenMP [3] and MPI [4]. OpenMP parallelizes tasks within a shared-memory environment, such as share distribution, order generation (where frequent data access is required), and portfolio updating, which facilitates efficient parallel processing within a single node by enabling multiple threads to access a shared memory space. MPI enables parallel processing across multiple processes in a distributed memory setting, facilitating parallel order matching and file I/O operations. For components such as the OrderBook and order matching engine, we use MPI to distribute tasks across different nodes, allowing effective data management across the system without the constraint of shared memory. Additionally, we enhance performance by using MPI I/O to handle large data volumes efficiently, thereby reducing wait times and improving throughput.



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)
(1) (2)
Order Generation Engine
(3)
Order Book
(1)
Additional Module: Sorting Engine
(1) (2)
Order Matching Engine
(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
Parallelization Techniques:
(1) Shared memory: OpenMP
(2) Distributed memory: MPI
(3) Distributed Memory: MPI I/O
Synchronization point

Figure 1: Parallel Design

Figure 1 presents an overview of code parallelization techniques used in the stock trading platform.

The key parallelization techniques employed are:

1. Shared memory parallelization using OpenMP: This technique allows multiple threads to access and modify shared memory concurrently, enabling parallel execution of certain tasks within a single process.

2. Distributed memory parallelization using MPI: Message Passing Interface (MPI) is used to split the workload across multiple processes, each with its own private memory space. Processes communicate and exchange data through message passing.

3. Distributed memory parallelization with MPI I/O: In addition to using MPI for process-level parallelism, MPI I/O is utilized for parallel input/output operations.

Parallelization techniques are applied at different stages of the order processing pipeline and across components including the order generation Engine, OrderBook, sorting engine, and order matching engine. We implement multi-threading by parallelizing the loops over traders and orders within each process. Different processes handle different stocks to effectively distribute the workload across multiple compute nodes. Synchronization points, denoted as stars in Figure 1, are placed at specific stages to ensure data consistency and proper coordination between the parallel processes and threads. The parallelization strategies aim to maximize resource utilization and throughput while maintaining the correctness of the order matching system.

3.2.1 MPI I/O

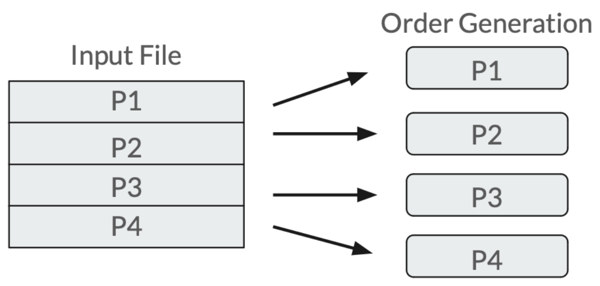


Figure 2: Parallelization Implementation - MPI I/O

The main challenge arises from the fact that each line represents individual stock, trader, or order information, and the use of the getline() function in the serial version inherently operates sequentially. The proposed solution involves dividing the file size by the number of processes and assigning each process a contiguous chunk of bytes using offsets, as depicted in Figure 2, which initiates the entire distributed memory model via MPI across the system. However, there are certain edge cases, such as when the number of lines cannot be evenly divided by the number of processes. To address this issue, we can read in additional data at the beginning and end of the file.

3.2.2 Order Generation

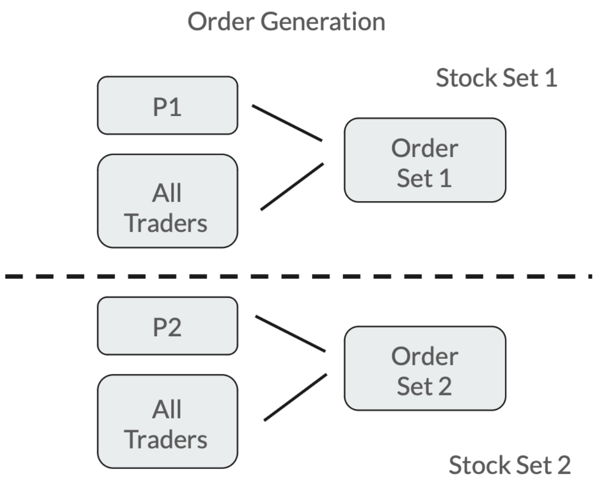


Figure 3: Parallelization Implementation - Order Generation

In our system, we iterate over all stock/trader pairs, sharing trader information among all processes. As trader information remains read-only during the order generation process, we replicate all trader data for each process. Within each process, we utilize thread-level parallelism using OpenMP, enabling concurrent order calculations over portfolios and stocks. Each thread autonomously generates a local list of order strings to prevent resource contention. To maintain reproducibility without race conditions, we employ a thread-local random number generator. Offer prices and trade quantities are computed within each thread using local data, with thread-specific decisions to create buy or sell orders, stored in thread-local storage. Thread outputs are synchronized and concatenated into a single string for efficient file writing. MPI is utilized to collect data sizes from each process and compute offsets for combined file writing. A collective write operation with MPI is then performed to output all order strings to a CSV file in parallel, ensuring accurate data alignment. Finally, dynamically allocated resources for MPI operations are cleaned up, and the file handle is closed.

3.2.3 Order Matching

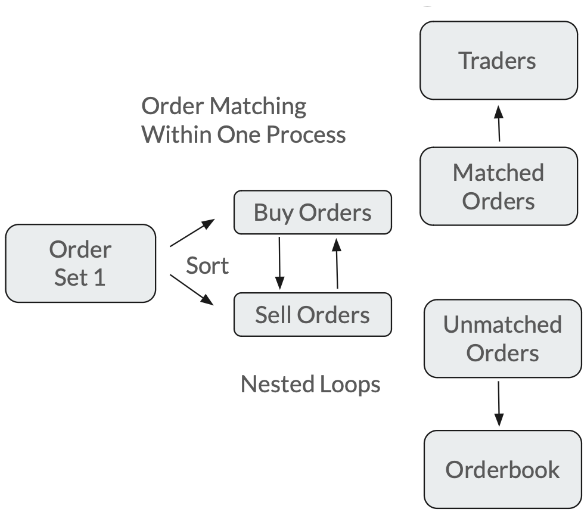


Figure 4: Parallelization Implementation - Order Matching

The most significant bottleneck arises during the order matching stage. Post-generation, each process is responsible for managing orders pertaining to a subset of stocks while still retaining a copy of all trader information. Using MPI previously allows us to employ OpenMP within each MPI process at this stage. Our focus now centers on optimizing merge sort, which sorts buy and sell orders based on quantity. Subsequently, buyer-seller matching entails a nested loop to compare each pair of orders. After the matching procedure, we need to update matched orders within the trader information to reflect their profit or loss and write unmatched orders back to the OrderBook using MPI I/O. MPI communication is used for aggregating changes in trader information, i.e., the profit and loss, across all processes, albeit only after completing all matchings. To optimize order matching, we focus on the outermost loop with dynamic scheduling to mitigate load imbalance. Order matching experiences significant load imbalance primarily due to inherent randomness. We only parallelize the outer loop with dynamic scheduling, considering the limited number of threads available.

4. Performance Benchmarks and Scaling Analysis

4.1 Roofline Analysis

In our roofline analysis, parallel execution achieves higher performance compared to serial execution, as expected. The operational intensity, measured in FLOPs/byte, for all the data points is relatively low, indicating a smaller proportion of floating-point operations relative to

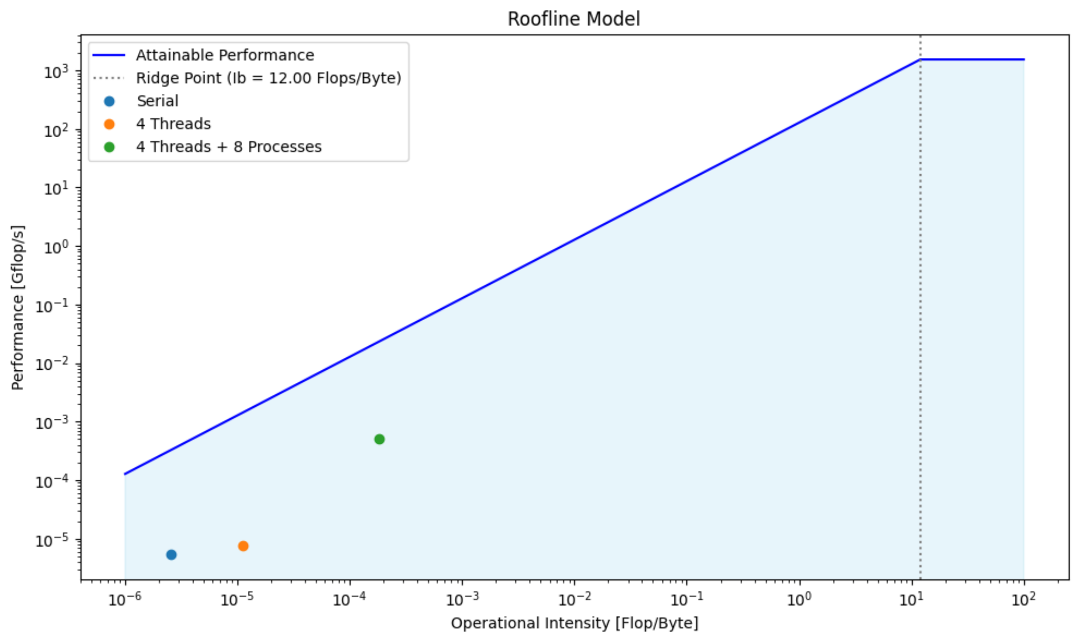


Figure 5: Roofline Model

data transferred and that the workload is memory-bound rather than compute-bound. The performance is limited by data transfer rather than computational throughput. Our system does not involve many floating point operations like those in risk management and high-frequency trading. Furthermore, our system involves substantial data transfer, including tasks such as market data updates, order submissions, and trade executions.

Overall, these observations suggest that there is room for further optimization, potentially by increasing the operational intensity or improving memory bandwidth utilization to approach the peak attainable performance.

4.2 Strong Scaling

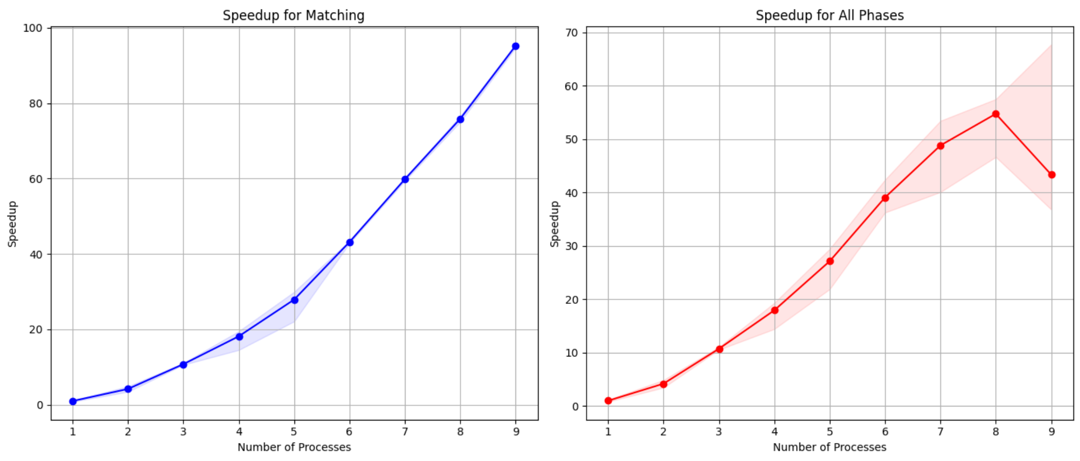


Figure 6: Strong Scaling

We maintained a consistent configuration with 500 stocks and 10,000 traders. In parallel execution, we used 4 threads per process. We plotted Figure 6 by varying the number of processes from 1 to 9 along the x-axis. The y-axis illustrates the average speedup attained across five runs. The shaded regions on the graph indicate error margins. The widened shaded range observed as the number of processes increases results from the combined effect of increased overhead and the randomness inherent in initialization and order generation processes.

In our strong scaling analysis, we isolate order matching as the most computationally intensive component, where the majority of processing occurs. In the order matching phase, our program scales perfectly. However, in other parts of the program, such as reading data, stock distribution, and order generation, they do not scale as well and introduce overheads. This is because the overhead of managing MPI communication and synchronization outweighs the computational demand for these phases when dealing with a relatively small number of stocks, such as 500. Consequently, each distributed task becomes significantly smaller relative to the overhead, leading to a diminished overall speedup as the number of processes exceeds 8. With additional computing resources beyond the cluster's limitations, we anticipate achieving better scaling results by experimenting with a larger number of stocks.

4.3 Weak Scaling

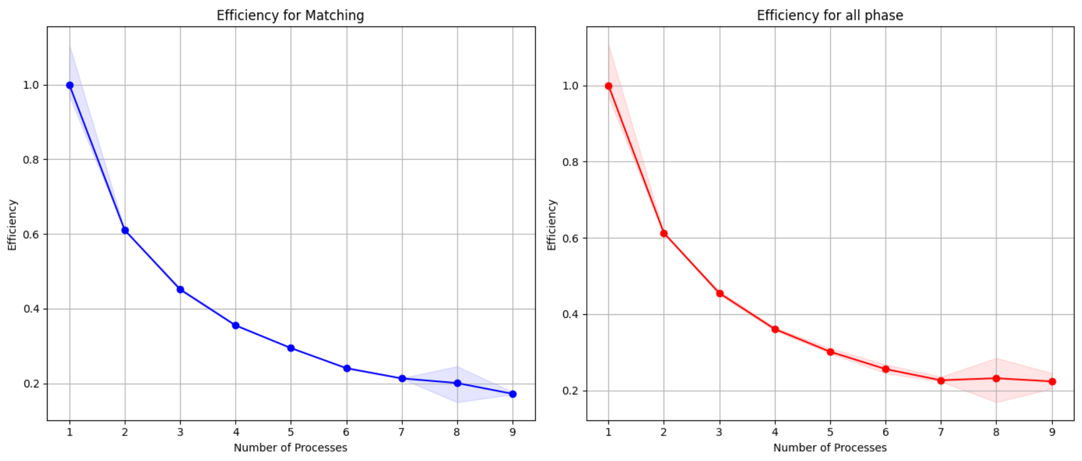


Figure 7: Weak Scaling

Figure 7 shows the efficiency against the number of processes for order matching and for all phases.

The configuration has a fixed number of 500 stocks to simulate a realistic stock trading system where the number of stocks stays constant. The number of traders vary from 2,000 to 20,000. Each process consists of 4 threads, with a data size allocation of 40 kB per process.

In our weak scaling analysis, we measured efficiency by increasing both the workload and the number of processes simultaneously. This time, there is no significant difference between the performance of order matching and the overall performance. In an ideal weak scaling scenario, we would anticipate efficiency to remain constant at 1, indicating that adding resources in proportion to the increased workload would not result in performance loss. However, efficiency diminishes as the number of processes increases due to the escalating complexity of managing communication and synchronization. This overhead consequently leads to decreased efficiency across all components of our program.

4.4 Threading Performance

We examined the impact of threading on program performance within the context of 500 stocks and 10,000 traders. Our experiment spanned from 1 to 36 threads within a single process.

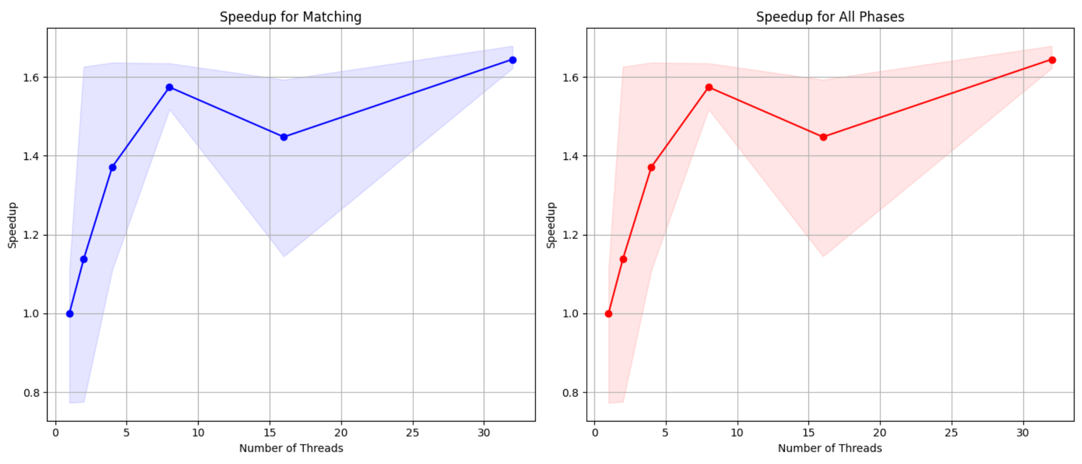


Figure 8: Speedup vs. Number of Threads

Figure 8 illustrates the speedup achieved by thread increments for order matching and all phases. For both graphs, initially, there is a significant gain in speedup as threads are added, up to around 8 threads. Beyond that point, the speedup tapers off, indicating diminishing returns due to factors like communication overhead or limited parallelizable work. This observation underscores the challenge posed by overheads among MPI processes, which emerge as the primary bottleneck limiting scalability.

	Baseline 1	Baseline 2	Parallelized
Typical wall clock time (hours)	2.846575	2.075481	0.037904
Number of processes	1	1	8
Number of threads per process	1	4	4
Maximum number of input files in a job	3	3	3
Maximum number of output files in a job	2	2	2
Library used for I/O	MPI I/O	MPI I/O	MPI I/O

Table 1: Workflow parameters of the three test cases

Notes:

- There are three input files in a job: one for stocks, one for traders, and one for generated orders.
- As for output, there are two files: one for generated orders and another for unmatched orders.

5. Resource Justification

The benchmark tests conducted with different configurations, as outlined in Table 1, show that the parallelized setup leads to significantly reduced wall clock time compared to the baseline configurations, which underscores the effectiveness of parallelization in improving computational efficiency. The observed reduction in wall clock time highlights the necessity of parallelization and the utilization of additional resources to achieve optimal performance.

The order matching process incurs the most significant overhead. As shown in Table 2, there is a substantial reduction in the percentage of runtime attributed to order matching when using 8 processes, each with 4 threads (the optimal job size based on our strong scaling and threading experiments), compared to a single process. Specifically, this percentage decreases from 99.98%

Number of Processes	Number of Threads per Process	Runtime Distribution
1	-	Order Matching: 99.98%
8	4	Read: 0.0143561% Order Generation: 27.81640% Order Matching: 72.1692365%

Table 2: Runtime Analysis

to 72.17%. By distributing the workload across multiple processes and threads, parallelization maximizes resource utilization and computational throughput, ultimately leading to shorter execution times.

Furthermore, with access to more computing resources, we anticipate achieving better scaling results by exploring larger datasets, specifically by conducting tests with a greater number of stocks. The availability of additional resources and processes would facilitate the execution of tests with larger datasets, consequently enhancing the scalability of our simulations and improving overall performance.

Author Contributions

April Zhang: Local and on-cluster roofline analysis (lead), serial code profiling (lead), performance and scaling analysis of parallelized program (lead), serial and parallelized code debugging (supporting)

Susannah Su: Serial code development from scratch (lead), serial code debugging (lead), documentation (lead), data generation script (lead), parallelization design (supporting)

Aaron Li: Serial code debugging and profiling (supporting), parallelization design (lead), parallelization implementation and debugging (lead), roofline and scaling analysis of parallelized code (supporting)

Catherine Gai: Serial code development, order matching logic design, implementation, and debugging (equal contribution, lead); data preparation (lead); serial code profiling: order generation (lead); parallelization design (supporting)

Yixuan Qiu: Serial code testing, debugging, and profiling; roofline analysis; report write-up

References

[1] Alessandro Bigiotti and Alfredo Navarra. ‘Optimizing automated trading systems’. In: *Digital Science*. Springer. 2019, pp. 254–261.

[2] Eugene A Durenard. *Professional Automated Trading: Theory and Practice*. John Wiley & Sons, 2013.

[3] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[4] Edgar Gabriel et al. ‘Open MPI: Goals, concept, and design of a next generation MPI implementation’. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users’ Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer. 2004, pp. 97–104.