Project Report: Parallel Automated Trading System for Low-Latency Stock Trading

Shauryasikt Jena* **Shubham Maheshwari***
EE451/CSCI452 - Prof. Viktor K Prasanna
University of Southern California
{jenas, sm57628}@usc.edu

1 Introduction

Low-latency trading systems are concerned with executing trades at the fastest possible speeds, with their success dependent on optimal data processing and rapid order execution. In the quest for faster systems, distribution of data processing and parallelizing these processes is at the forefront of recent developments. The aim is to reduce the time for pre-processing raw data and related risk assessment to effect trade executions as speedily as possible to gain a competitive edge in the market.

This project explores the development of an automated low-latency trading system, that leverages distribution of data and tasks that can be parallelized for accelerated executions of trades. This work identifies the parallelizable parts of a serial trading algorithm, and investigates the improvement in terms of reduced latency and increased order execution rates when said parts are run in parallel.

2 Strategy Formulation

The primary objective of this project is realized by analyzing the properties of the dataset, identifying parallelizable components of the serial implementation and effecting strategies on how to exploit maximum parallelism.

2.1 Dataset

The stock prices of a given number of company portfolios, 30 for this project, in the NASDAQ exchange from one market day are used (Dukascopy, 2023). Each data sample has 6 attributes.

1. **Time:** the timestamp of when a stock's market evaluation is updated
2. **Symbol:** NASDAQ stock symbol of the stock
3. **Bid:** The price at which brokers (user) expect to buy 1 stock, i.e., cost price
4. **Bid Volume:** The maximum number of stocks available to be bought at the instant
5. **Ask:** The price at which brokers (user) expect to sell 1 stock, i.e., selling price
6. **Ask Volume:** The maximum number of stocks that can be sold at the instant

Timestamp	Ask	Bid	Ask Vol.	Bid Vol.
06:30:00.625	186.004	185.776	0.012	0.012
06:30:00.676	185.993	185.787	0.012	0.012
06:30:00.828	185.864	185.786	0.012	0.012

Table 1: Example: Apple (AAPL) first 3 ticks

The bid volumes and ask volumes are regulated to maintain the stability of the market. Each data sample for a company is provided in ticks. A tick is a minimum measure of upward or downward movement, here US$0.01, in the price of a stock.

2.2 Algorithm

Here α, Relative Strength Index (RSI), and Risk Reward Ratio (RRR) are intermediary tools processed from the data to decide trades.

The automated trading system will — (a) handle real-time market data, (b) analyze it for trading signals, and (c) execute orders.

Machine Learning model weights were formerly used in automated trading systems to calculate the above mentioned tools. However, such large update steps were removed once statisticians derived simpler formulae to closely replicate their performance. Although it was proposed to parallelize such ML architectures for this project in the first draft, the implementation was dropped in favor of recent methods to maintain the importance of parallelism in relevant algorithms.

2.2.1 Momentum α

The momentum of a stock is the directional upwards or downwards movement of its value over the most recent fixed window of time or events (ticks). It is quantified by the momentum α that is the average directional trend of the stock's value.

As a general rule, positive alpha would denote expected future growth and negative alpha would denote expected future depreciation.

$$\alpha = \frac{\text{Bid}_{\text{now}} - \text{Bid}_{\text{past}}}{\text{Bid}_{\text{past}}} \quad (1)$$

Algorithm 1 Automated Trading System

Input: Historical Information of 30 stocks from NASDAQ

Output: Number of stocks retained and profit accounted at the end of a market day

Create Classes to store Ask,Bid,Ask Vol, Bid Vol, derivatives for each company

Function `find_`α (*combined file*, p) **:**

  α calculation, look over p past ticks

return α map

End Function

Function `find_RSI` (*combined file*, T) **:**

  RSI calculation, look into past T ticks

return RSI map

End Function

Function `find_RRR` (*combined file*, p) **:**

  RRR calculation, look into past p ticks

return RRR map

End Function

Function `SmartOrderRouting` (*combined file*, α, *RSI, RRR*) **:**

  Trigger trade (buy/sell) signal

  Determine order quantity

return net profits, net stocks

End Function

Net profits should be maximized, net stocks should be 0

For a running market day, sliding window is a suitable algorithm to continuously evaluate α in $\mathbf{O(1)}$.

2.2.2 Relative Strength Index (RSI)

Relative Strength Index (RSI) (Moroșan, 2011) is a momentum oscillator that denotes if a stock is overbought or oversold. It gives useful information about predicting reversion in the value of stocks, crucial for timing buy or sell signals. RSI is given by equation 2, and as a rule the value of 50 is balanced. Depending on the behavior of the market, the RSI conditions for buy or sell are set.

$$\text{RSI} = 100 - \frac{100}{1 + \frac{\text{Avg. gains}}{\text{Avg. losses}}} \quad (2)$$

RSI can be evaluated suitable by the moving averages algorithm in $\mathbf{O(1)}$.

2.2.3 Risk-Reward Ratio (RRR)

Risk-Reward Ratio (RRR) is evaluated by the trends of all individual gains and losses over a fixed window, the same size as that for evaluating α. However, it involves complex Fibonacci convergence algorithms to estimate the target inherent value of a stock, irrespective of its current evaluation.

$$\text{RRR} = \frac{\text{Potential Profit}}{\text{Potential Loss}} \quad (3)$$

α and RSI are used to determine the buy/sell/hold trade signal and RRR determines the order quantity depending on the bid/ask quantities. The trades are executed as per the Smart Order Routing (SOR) (Wikipedia, 2023) algorithm.

If all the conditions of any column of the Table 2 are not satisfied then the no order is executed and the trade is held for that tick.

Term	Buying condition	Selling condition
α	$\alpha > 0$	$\alpha < 0$
RSI	RSI < 30	RSI > 70
RRR	RRR > 1	RRR < 1

Table 2: Simple strategy metrics

2.3 Hypothesis

The objective is to lower the latency between reading tick information and executing the subsequent trade by exploiting parallelism in the algorithm 1. This project leads as per the following hypothesis.

- **Data Parallelism:** Each newly free thread is responsible for the next incoming tick.
- **Task Parallelism:** Finding the intermediate values, α, RSI, and RRR, for each tick update can be parallelized.
- **Algorithmic Parallelism:** This can be implemented in 2 places.
  - Synchronous: Different components of one function call for SOR can be parallelized.
  - Asynchronous: Reading the next tick information, and executing the trade based on that information can be parallelized.

3 Method

First, a serial version of the code is designed as per algorithm 1. Then the parallelizable components of the algorithm as identified in Section 2.3 are parallelized and they are benchmarked against several metrics of performance, namely — **latency**, **throughput**, and **order execution success rate** for the speedup. The two versions are also be compared on the basis of their profitability as a measure of correctness.

For uniformity in the complexity of total task, the following hyperparameters are set:

- **10 ms** = Delay in reading a tick from file in real time
- **10 ms** = Delay in reflecting order execution to the market exchange
- **10** = Difference in ticks to monitor for α and RRR
- **50** = Number of ticks in the window to consider for RSI
- **10 minutes** = Initial monitoring period of the market day to define above hyperparameters and the conditions in Table 2

All computations are done on the d14 nodes from the *gpu* partition on discovery HPC cluster of CARC, USC. The specifications of the CPUs on any such node are:

- Model: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
- Architecture: x86_64
- Thread(s) per core: 1
- Core(s) per socket: 16
- Socket(s): 2

3.1 Serial Implementation

The flow of algorithm 1 is described in Figure 1. There is only one thread of the main program in this implementation. Figure 1 shows the flow of different functions from reading a tick to executing a trade for each trade signaled. We simulate the time difference between ticks by letting the program sleep for the corresponding amount between reading the two ticks.

Orders cannot be canceled here as the cancellation is triggered by another tick of the same company registering before the execution of an order based on the current tick, and this implementation does not allow for such parallel reading.

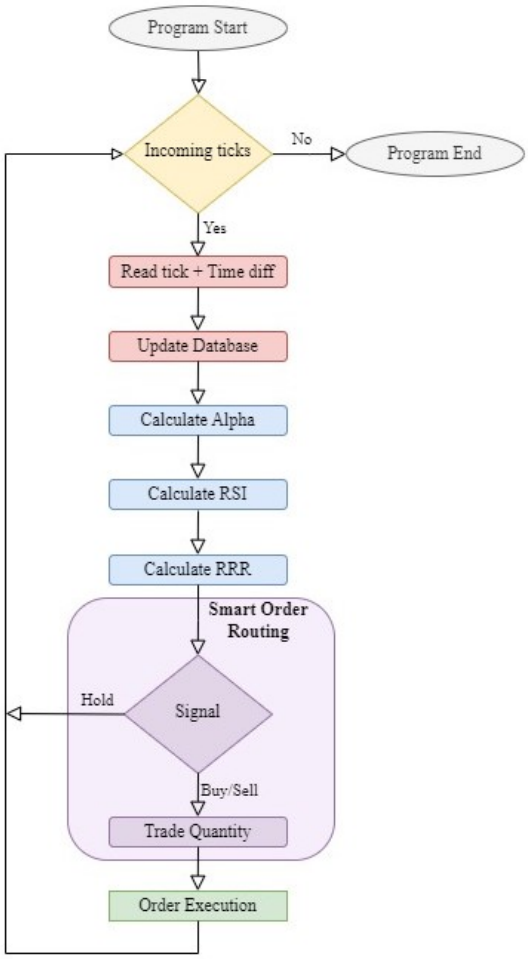

Figure 1: Serial Automated Trading System on 1 thread

3.2 Parallel Implementation

Figure 2 shows the overall structure for parallel implementation. One main thread is responsible for reading a tick, maintaining the time difference between ticks, updating ticks in the database, and adding the tick to the shared task queue similar to producing a resource in the queue. All these steps are repeated till ticks are incoming. There are n ($n \geq 1$) worker thread sets that are created before starting the tick reading to perform data parallelism. Suppose for a particular tick, one of the worker thread sets started its computation but on the main thread, another tick for that same stock symbol appears, then the previous tick won't be executed and that order will be canceled.

In each worker thread set, there are 3 threads:

- **Worker thread** Responsible for the computation of a tick. This thread waits for the main thread to add a tick to the task queue or end

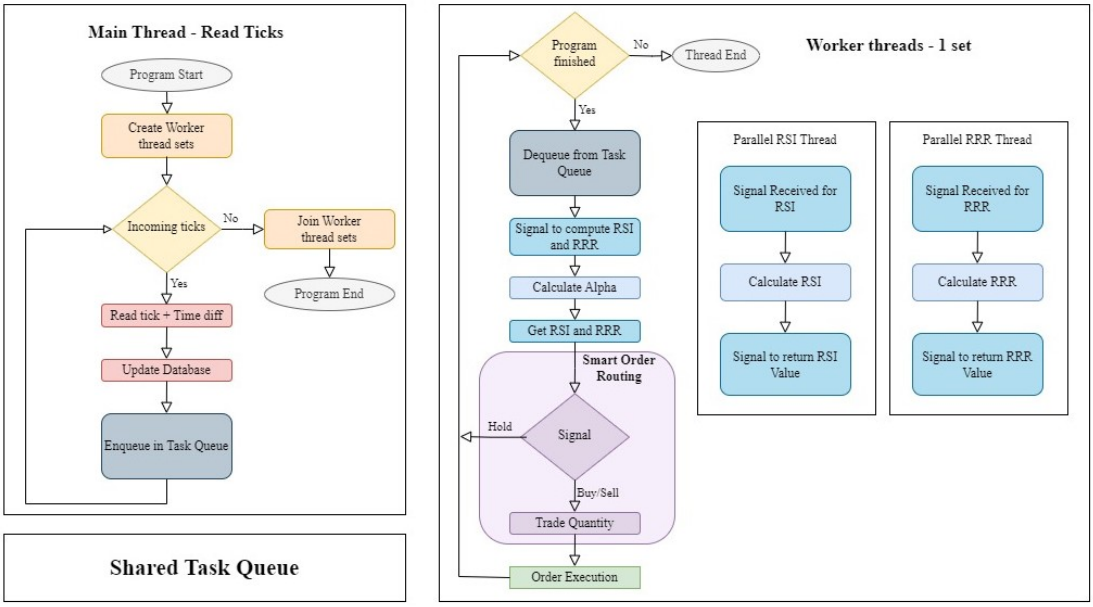

Figure 2: Parallel Automated Trading System on 1 thread

Implementation	Threads	Total Ticks	Ticks Read	Orders Executed	Orders Cancelled	Average Latency	Profits (US$)
Serial	1	667218	380741	357931	-	833.1μs	178.64
Parallel	1 + **1**×3	667218	667218	588389	37630	254.4μs	189.02
Parallel	1 + **2**×3	667218	667218	622844	3174	249.6μs	132.74
Parallel	1 + **4**×3	667218	667218	626006	12	247.6μs	137.06
Parallel	1 + **8**×3	667218	667218	626019	0	245.5μs	139.08

Table 3: Performance metrics for all the implementations. For parallel implementations, there is one main thread alongside triads of worker threads. The latency is reduced by 10 ms or 10,000 μs to account for replicating the real life communication delay to the markets for reflecting a change in the portfolio.

the program, as soon as it sees that the queue has been updated, it fetches a tick from the queue like consuming the produced resource. If a tick is consumed by 1 worker thread, other workers will not consume that tick, and it will be removed from the queue.

At the creation of this thread, it creates two child threads, one for computing RSI and one for computing RRR, these threads help in task parallelism. After fetching a tick from the task queue, this thread stores the tick in a shared space and signals both the child threads to start computing RSI and RRR values respectively. Then it proceeds to calculate the alpha value using the tick's information and wait for the child threads to complete their execution.

After receiving RSI and RRR values, it computes the signal and order quantity in parallel performing algorithmic parallelism and finally checks whether this tick is the latest tick for the stock or not and then it performs the order execution according to the signal if it is the latest tick.

These steps are repeated for incoming ticks till the program-ended signal is sent by the main thread.

- **RSI thread** This thread is for computing RSI value for the tick being shared by the worker thread. It starts the computation when it receives the signal to start it. Upon computing the RSI value, this thread sends a signal back to the worker thread, informing it that the RSI value is computed.

- **RRR thread** This thread is for computing RRR value for the tick being shared by the worker thread. It starts the computation when it receives the signal to start it. To perform

the RRR computation of a tick, this thread also accesses the history of that stock's prices in the database. Upon computing the RRR value, this thread sends a signal back to the worker thread, informing it that the RRR value is computed.

At the end of reading all ticks, all the threads are joined, and the final outputs are stored by the main thread.

4 Results and Discussion

The performance of the serial implementations and all the parallel implementations are described in Table 3.

4.1 Observations

Ticks Read: This is directly related to the throughput of the system as only threads that are read can be utilized for further processing. With the introduction of data parallelism in the form of asynchronous reading, the primary algorithm of the program is also parallelized in reading and processing. Hence, only the serial implementation fails to read all the ticks.

Orders Executed: This is a measure of the throughput, corresponding to how many trade signals (buy/sell/hold) were handled. Order execution success rate also increases with an increase in the number of threads, as multiple worker blocks can handle processing incoming ticks with fewer chances of waiting until the next tick of a stock symbol under processing is queued.

Orders Cancelled: This is the number of orders canceled because of the next incoming tick of the company under processing. Fewer orders canceled imply a higher order execution success rate.

Average Latency: This is the average latency between the completion of reading a tick and completing a trade order based on it for all the companies under study. This decreases along with an increase in the number of threads. The clear difference between serial and parallel latency is due to the introduction of threads for computing RSI and RRR values in parallel with α.

Profits: A measure of correctness so that parallelizing the system does not incur severe costs to the application it is aiding.

4.2 Performance Considerations

Since this project is concerned with latency being in the order of microseconds, we had to carefully consider introducing parallelism with minimal overhead costs. We tried several parallel implementations and finalized the one that gave the best results.

One of the implementations was to create a thread for a tick if the total number of threads is less than the number of worker blocks and then join the existing threads to create new ones for new ticks. This approach showed poor results as the overhead was too large in creating and joining threads repeatedly.

Another implementation was to send the tick data to running worker threads, this showed some promising results but some worker threads had to wait if they finished their work earlier and the main thread had not reached to them in the round-robin fashion. This paved the way for our current implementation which was thread pooling and maintaining a shared task queue. Each running thread consumed a tick from that task queue as soon as they finished their work, minimizing all overheads.

5 Conclusions

Our hypothesis of introducing data parallelism via asynchronous reading and processing tasks, task parallelism for computing the trading intermediaries, and algorithmic parallelism in Smart Order Routing was successful. The parallel implementation clearly reduced the latency for processing the information of one tick to execute a trade based on that, as well as allowed for increased reception towards continuous fast-paced inputs leading to better throughput.

The profits suffered to some extent during parallelization, as the hyperparameters need to be changed suitably for optimal gains. However, the hyperparameters directly influence computational load and hence, had to be in agreement with the serial implementation for fair comparisons.

Limitations

Given the domain of microseconds for this application, introducing parallelism is not scalable due to the overhead costs, implying a point of saturation with increasing the number of threads.

Also, the hyperparameters need to be changed as per different behaviors of market. These changes might introduce slight irregularity in the observed trends but should not affect the benefits of introducing parallelism.

Appendix

The code to our project can be found on this link github.com/jenashauryasikt/EE451-Project.

References

Dukascopy. 2023. Dukascopy - historical data feed.

Adrian Moroșan. 2011. The relative strength index revisited. *African journal of business management*, 5:5855.

Wikipedia. 2023. Wikipedia - smart order routing.