Workload Balancing Strategies in Parallelized Computation of TA Indicators for Predictive Analysis of Stock Data

Varin Jaggi and Shubh Agarwal
University of Southern California

Introduction

Stock market prediction and analysis are some of the most difficult jobs to complete. There are numerous causes for this, including market volatility and a variety of other dependent and independent variables that influence the value of a certain stock in the market. These variables make it extremely difficult for any stock market expert to anticipate the rise and fall of the market with great precision.

Our proposed application uses technical analysis (TA) indicators SMA(14) and SMA(30), and implements them using parallel computing techniques, namely multithreading. SMA stands for Simple Moving Average. It is the average price over a specified period. The average is called "moving" because it is plotted on the chart bar by bar, forming a line that moves along the chart as the average value changes.

$$SMA(\alpha) = \frac{\Sigma\, Closing\ price\ of\ past\ \alpha\ seconds}{\alpha}$$

The parallel implementation of the computation of these indicators gives rise to a wide variety of workload imbalance issues. For example, some stocks are more frequently traded than others and this means there are more ticks in α seconds, which corresponds to more computation needing to be done to calculate these indicators due to a larger number of data points. As a result, some processes may have a larger amount of computation assigned to them at compile time compared to other processes which may be assigned stocks that are not frequently traded. This can be solved using static (compile time) scheduling of tasks based on historical trends.

In this project, several work was done:

- Serial implementation of the retrieval of stock data and computation of TA indicators using the Python language
- Implementation of a naive Parallel version of the retrieval of stock data and computation of TA indicators using the Python language
- Identification and analysis of the workload imbalance caused by the volatility of the stock market
- Further optimization of the parallel computation of TA indicators through a special scheduling algorithm
- The comparative analysis of the serial approach vs. the simple parallel approach vs. combinatorially optimized compile time scheduling

Algorithm

There are two major parts of parallelization that we leverage:

1. We attempt to achieve ~n times speedup using n threads to compute n TA indicators. After reading the input, within each thread we first parallelize the computation of the two TA indicators using two threads. Then we parallelize the computation of the two TA indicators for all 50 stocks using 2 x 50 = 100 threads. The speedup is calculated based on the average computation time taken in each second (i.e. each iteration).

2. We attempt to achieve ~p times speedup using p processes, where number of processes = number of cores. To optimize this algorithm and to eliminate workload imbalance we deploy our scheduling algorithm. We first analyze the number of data points for each stock and then we assign stocks to different processes in a way that tries to balance the workload between each process. Briefly, if a stock has a high number of data points then its computation is matched with a stock that has a low number of data points.

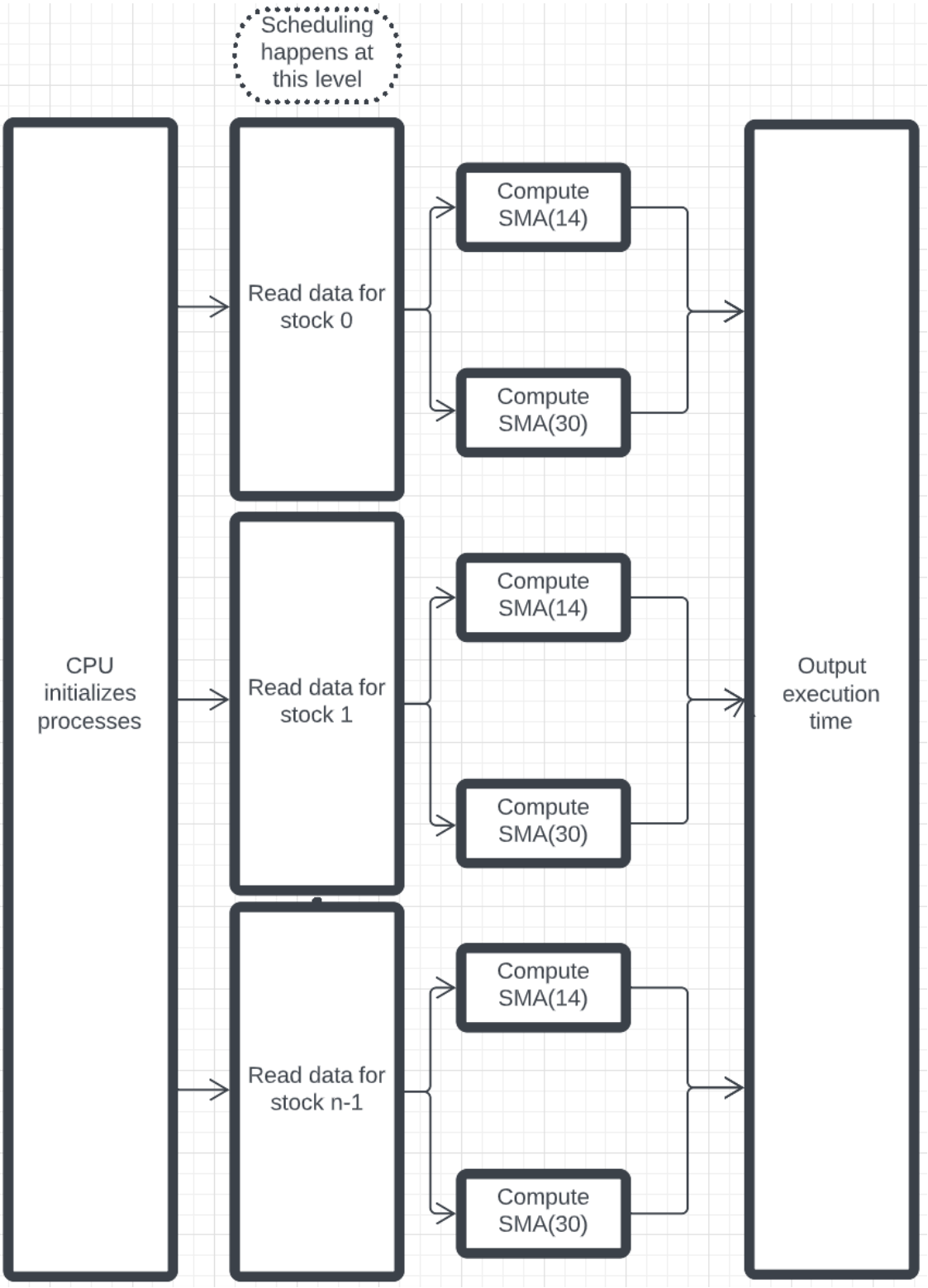

Scheduling happens at this level
CPU initializes processes
Read data for stock 0
Read data for stock 1
Read data for stock n-1
Compute SMA(14)
Compute SMA(30)
Compute SMA(14)
Compute SMA(30)
Compute SMA(14)
Compute SMA(30)
Output execution time

Figure 2

Static Combinatorially Optimized Scheduling Algorithm (SCOSA):

```
# Get list of stocks sorted based on liquidity from previous trading day
order = []
for i in range(0, len(result2)):
    order.append(tuple((len(result2[i]), i)))
order = sorted(order)

array1 = []
array2 = []
array3 = []
array4 = []
array5 = []
array6 = []
array7 = []
array8 = []
# Use multiway number partitioning to schedule tasks based on historical trends
for i in range(0, 6):
    array1.append(sma_14[order[i * 4][1]])
    array2.append(sma_30[order[i * 4][1]])
    array1.append(sma_14[order[49 - (i * 4)][1]])
    array2.append(sma_30[order[49 - (i * 4)][1]])
    array3.append(sma_14[order[(i * 4) + 1][1]])
    array4.append(sma_30[order[(i * 4) + 1][1]])
    array3.append(sma_14[order[49 - ((i * 4) + 1)][1]])
    array4.append(sma_30[order[49 - ((i * 4) + 1)][1]])
    array5.append(sma_14[order[(i * 4) + 2][1]])
    array6.append(sma_30[order[(i * 4) + 2][1]])
    array5.append(sma_14[order[49 - ((i * 4) + 2)][1]])
    array6.append(sma_30[order[49 - ((i * 4) + 2)][1]])
    array7.append(sma_14[order[(i * 4) + 3][1]])
    array8.append(sma_30[order[(i * 4) + 3][1]])
    array7.append(sma_14[order[49 - ((i * 4) + 3)][1]])
    array8.append(sma_30[order[49 - ((i * 4) + 3)][1]])
array1.append(sma_14[order[24][1]])
array2.append(sma_30[order[24][1]])
array3.append(sma_14[order[25][1]])
array4.append(sma_30[order[25][1]])
```

Scheduling Strategy

The multiprocessor scheduling problem consists of finding an optimal distribution of tasks on a set of processors. The number of processors and number of tasks are given. Time taken to complete a task by a processor is also provided as data. Each processor runs independently, but each can only run one job at a time. We call an assignment of all jobs to available processors a

"schedule". The goal of the problem is to determine the shortest schedule for the given set of tasks.

For 50 stocks there can be 50! permutations and hence using a brute-force approach to find the best possible schedule is not feasible. Instead we focused on devising our own scheduling algorithm to combat this problem. We were inspired by the multi way number partitioning problem. In computer science, multi way number partitioning is the problem of partitioning a multiset of numbers into a fixed number of subsets, such that the sums of the subsets are as similar as possible.

We tried to solve our scheduling problem using the same principle used to solve the following problem:

How do you divide the number 1 to 100 into 10 groups that have the same value?

We begin by first breaking them down into 50 groups that all have the same sum

(1 and 100)=101

(2 and 99)=101

(3 and 98)=101

(4 and 97)=101

….. (46 and 55)=101

(47 and 54)=101

(48 and 53)=101

(49 and 52)=101

(50 and 51)=101

Now we have all the numbers arranged into 50 grouped pairs that each sum to 101. Since all of them have the same sum, then any 5 of them picked and grouped together will have a sum of 505. We will end up with ten groups, and each group will contain ten numerical values that sum to 505 in each group.

In a similar way, our newly devised Static Combinatorially Optimized Scheduling Algorithm (SCOSA) first arranges the stock data available to us in ascending order of the number of data points. We then use the same strategy as above to assign stocks to processes such that the workload is balanced. We treat the number of data points as the numbers we need to partition (these are 1 to 100 in the example problem above) and we partition the data into our 4 processes (analogous to the 10 groups we divide the 100 numbers to in the problem above). In this way, we achieve a Static Scheduling Algorithm which achieves workload balancing through Combinatorially optimizing the distribution of stocks to processes.

The reason such a scheduling algorithm is imperative to improving performance is that a workload imbalance causes uneven distribution of work across processes. The figure below shows how load imbalance can impact efficiency. In this figure, tasks were assigned to processes in a round-robin approach. The time taken by the longest-running task contributes to the span, which limits how fast the parallelized portion can run. Load imbalance can be mitigated by over-decomposition, dividing the work into more tasks than there are workers. Like packing suitcases, it is easier to spread out many small items evenly than a few big items. This is shown in Figure 3. Some processors have fewer tasks than others. There is still a possibility that a very long task will get scheduled at the end of a run, but our static scheduling algorithm mitigates this risk and ensures load balancing of tasks and an optimized execution time.

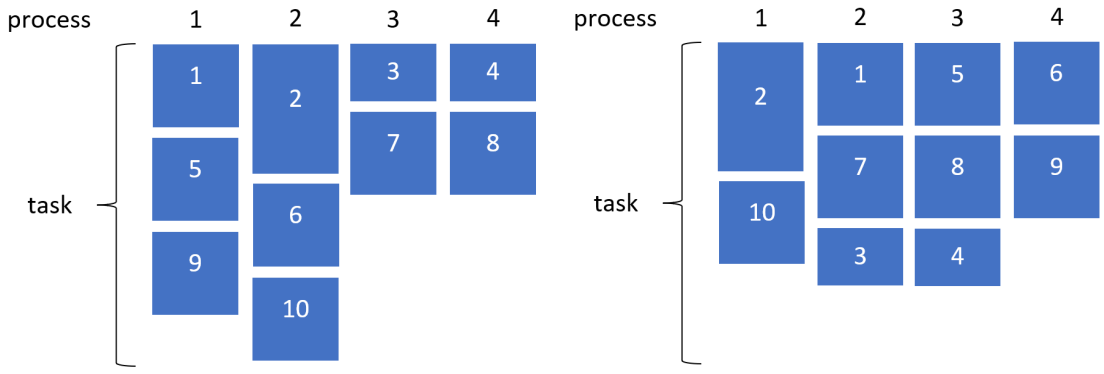


Figure 3

Hypothesis

Predictive analysis of stock data consists of one major kernel: computation of the TA indicators. This computation can be parallelized using multithreading as there is no data dependency between the computations of different TA indicators of different stocks. However, the problem of load balancing arises. So, we should be able to speed up the parallelized computation of TA indicators using scheduling algorithms to minimize workload imbalance (as shown in figure 3). These scheduling algorithms leverage the size of the data we have to better assign tasks to threads and schedule the underlying computations.

Thus our hypothesis are:

- The use of scheduling algorithms can further accelerate the parallelized computation of TA indicators for predictive analysis of stock data by exploiting information about the stock data
- Furthermore, Compile time scheduling should be more efficient than Run time scheduling of tasks due to the increased overhead of dynamically assigning and creating threads and more efficient than a Round Robin cyclic distribution approach to scheduling in parallel systems

Experimental Setup

All tests were carried out on a single-socket server equipped with the following:

1. Intel(R) Xeon(R) CPU E3-1225 v5 @ 3.30 GHz (4 cores, 4 threads)
2. 8 GB DDR4 RAM @ 2667 MHz

- **Data Preparation**

For the naive version of the algorithm, we use stationary (tick history) financial data for Nifty 50 stocks on the National Stock Exchange (NSE), India pulled directly from a MySQL database.

- **Serial implementation (baseline)**

We first implemented the retrieval of stock data and computation of the TA indicator on Python without any parallelization. The computation was performed

in a serial fashion. The execution time for this was measured and that served as our baseline.

- **Multithreading acceleration**

We used multithreading to examine the speedup of parallelism. As mentioned in the algorithm section, we parallelize the computation of the TA indicators where each thread is responsible for computing one indicator for one stock. We evaluated the performance using 2 threads vs. 100 threads.

- **Scheduling acceleration**

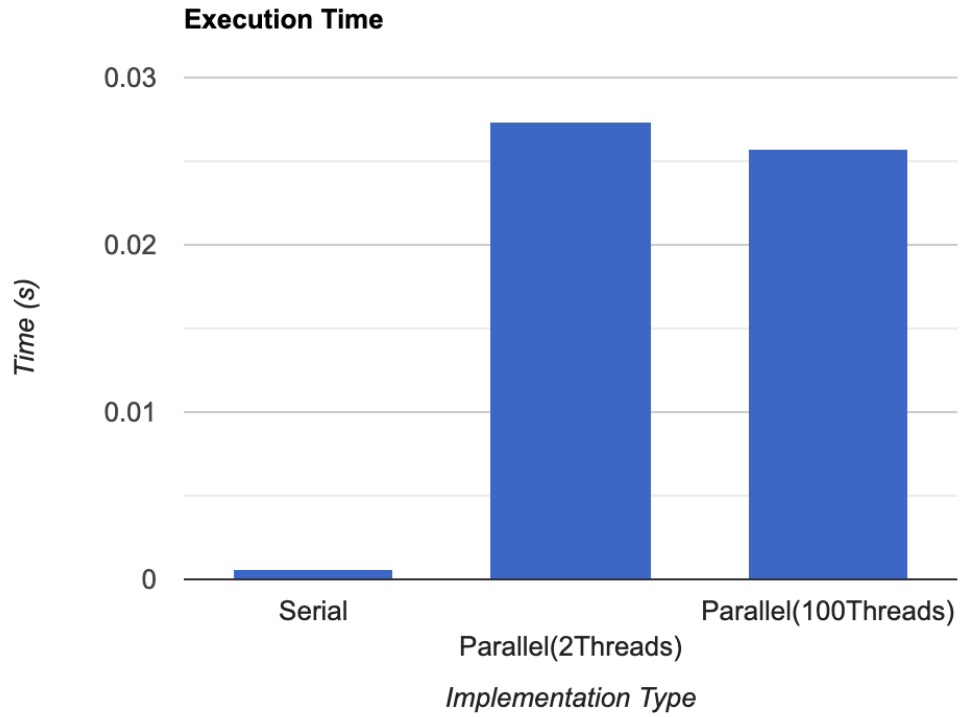
As mentioned in the compile time scheduling algorithm, we assign tasks to threads according to the size of the data that needs to be processed for the particular computation. This helps us evaluate the speedup in performance we get by rectifying the workload imbalance issue and this can further be compared to our serial implementation baseline.

Results and Analysis

- **Evaluating Multithreading Acceleration**

We had hypothesized that parallelizing the computation of the two technical indicators using multithreading will bring about a speedup of about n times (where n = # of threads used). However, we found that the overhead of spawning and destroying threads every time a new tick data is received is time-wise more expensive than the benefit that parallelism provides.

The following are our results:

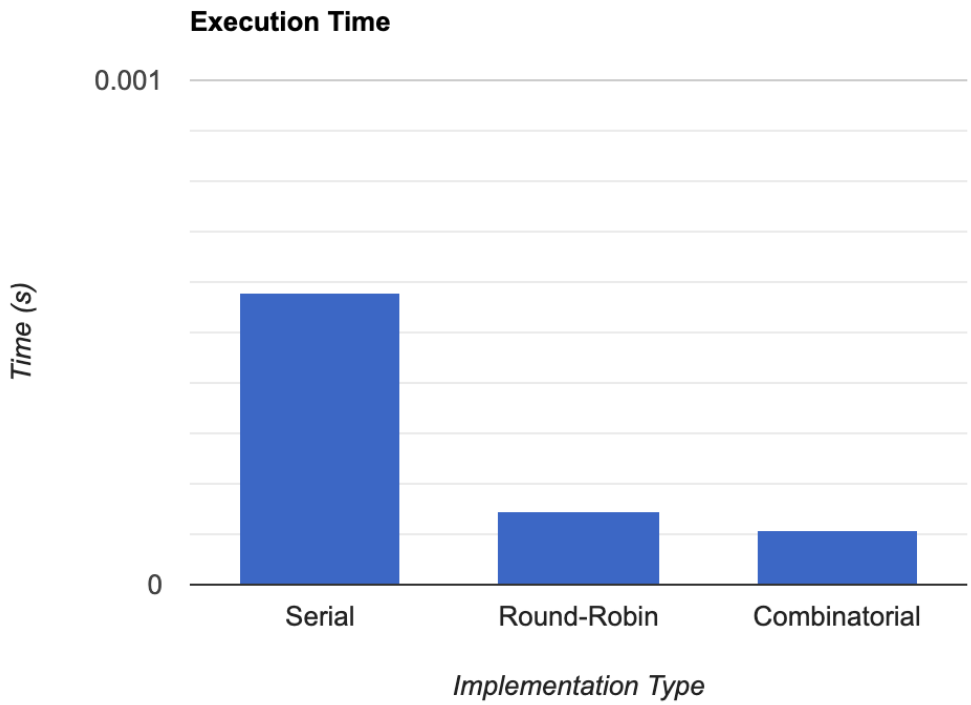


- **Evaluating Scheduling Acceleration**

According to the Concurrency Glossary, True parallelism requires hardware support (a multi-core processor), while pseudo-parallelism means that the parallel computations are an abstraction over serially interleaved sub-computations. We believe that we achieved pseudo parallelism through the use of multithreading which is why the overhead of creating and destroying new threads became a bottleneck that could not be overcome by the speedup achieved through multithreading. In a bid to try to speedup our parallel implementation we devised our scheduling algorithm (as described before). These algorithms enabled us to get a better performance through the optimized scheduling of the computations of the TA indicators of different stocks to different processes rather than trying to parallelize the computation of each individual TA indicator. We also implemented a Round-robin scheduling approach that uses cyclic distribution to randomly assign stocks to processes and does not take load balancing into account. We then compared this to our combinatorial scheduling

algorithm and found that it gives a 14.48% speedup over the round robin scheduling and a speedup of 78.62% over our serial baseline implementation.

The following are our results:



Conclusion

In this project, we first implemented a serial version of computing SMA(14) and SMA(30) using the python language. We measured the execution time for this serial computation as our baseline. We then implemented a naive parallel version of these computations using multithreading (as described in part 1 of the Algorithm section). After evaluating the performance we realized that multithreading only gave us pseudo-parallelism and so, the overhead of spawning and destroying new threads could not be compensated for. As a result, we proceeded to optimize our parallel approach by using multiprocessing along with our scheduling algorithm. We implemented a Round-robin scheduling version and measured the execution time. This, however, did not take into account the workload imbalance that arises due to parallel computation. To

further optimize our results we devised the Static Combinatorially Optimized Scheduling Algorithm (SCOSA).

After evaluating the performance of our scheduling algorithms we found that we achieved a 4.67x speedup over our serial baseline and a 1.2x speedup over Round Robin scheduling using SCOSA. This was achieved through the workload balancing attributes of SCOSA.

Further Scope

An issue arises, however, in the static scheduling of real time predictive analysis of stock data. During the realtime computation of these indicators, there can be a surge in the number of trades in a particular stock because of various reasons such as company/industry specific news, earnings, etc. This would increase the amount of work that a process assigned to that stock is responsible for. As a result dynamic scheduling is required to reschedule some tasks in real life to smooth out this workload imbalance. This is an ideal starting point for anyone interested in continuing this project. Further scope could include comparing the loss in efficiency in moving stocks from one process to another vis-a-vis the gain in efficiency from a more robust scheduling mechanism.

References

[1] Rajakumar, S., Arunachalam, V. P., & Selladurai, V. (2004, January 14). *Workflow balancing strategies in parallel machine scheduling*. Springer-Verlag London. Retrieved May 5, 2022,from http://www.me.nchu.edu.tw/lab/CIM/www/courses/Flexible%20Manufacturing%20Systems/FMS%20Paper%20review/12Workflow%20balancing%20strategies%20in%20parallel%20machine%20scheduling.pdf

[2] Load imbalance. Load Imbalance - an overview | ScienceDirect Topics. (n.d.). Retrieved May 8, 2022, from https://www.sciencedirect.com/topics/computer-science/load-imbalance

[3] *Concurrency glossary - slikts.github.io*. (n.d.). Retrieved May 9, 2022, from https://slikts.github.io/concurrency-glossary/?id=overlapping-vs-interleaved-lifetimes-true-parallelism-vs-pseudo-parallelism