CS 5220 Parallel Programming Project: Algorithmic Trading

By,
Atheendra P Tarun (ap778)
Harsh Pandey (hp349)
Prasannjit Kumar (pk435)
Santoshkalyan Rayadhurgam (scr96)
Shashank Adimulam (sa793)

INTRODUCTION & BACKGROUND TO TRADING

" I can calculate the movement of the stars,
but not the madness of men."

– Sir Isaac Newton after losing a fortune in the South Sea Bubble

Financial Markets

Financial markets are mechanisms that allow people to trade liquid and fungible assets at relatively low transaction costs.

Financial markets facilitate raising capital, transfer of risk, transfer of liquidity and international trade. Capital is raised by acquiring investors. Risk can be spread, or transferred, by investing in uncorrelated assets.

Stocks

Stocks are instruments that represent an ownership share in the equity of a company. One can say a stock is divided into multiple shares, but for semantic purposes these words are equivalent [1]. The stock market is a public entity for the trading of such company stocks. Stocks are frequently traded on the stock market both for investment purposes and speculation.

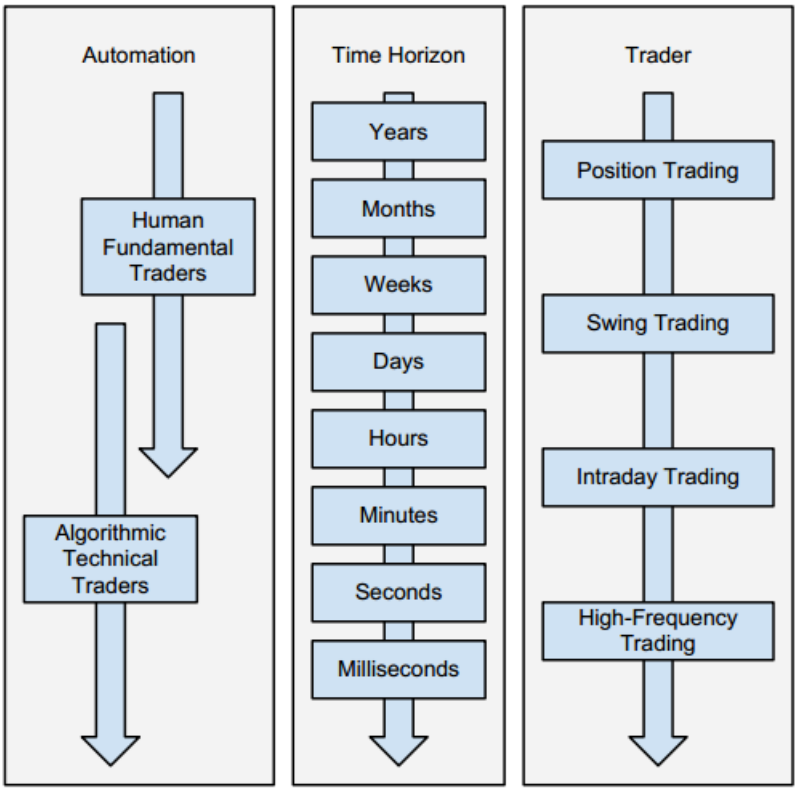
Stocks listed on the market can change value drastically for many reasons. In essence, the company is valued based on expected future yields. The fluctuations arise from events affecting the company or market, some theories that it even has are reflective property where price changes affect price levels in a feedback loop. These are properties that are expected in day-to-day trading.

Trading

Most assets in the world are traded. Where there is trade, there can be speculation. Speculation is often called active trading and is usually done with a short time horizon.

Until about the middle of the last century the notion of finance as a scientific pursuit was inconceivable. Trading in financial securities was mostly left to gut feeling and bravado. The emergence of mathematical models not only transformed finance into a quantitative science, but also changed financial markets fundamentally.

A mathematical model of a financial market typically leads to an algorithm for sound, automated decision-making, replacing the error-prone subjective judgment of a trader. Certainly, not only do algorithms improve the quality of financial decisions, they also pave the ability to handle the sheer amount of transactions of today's financial world.

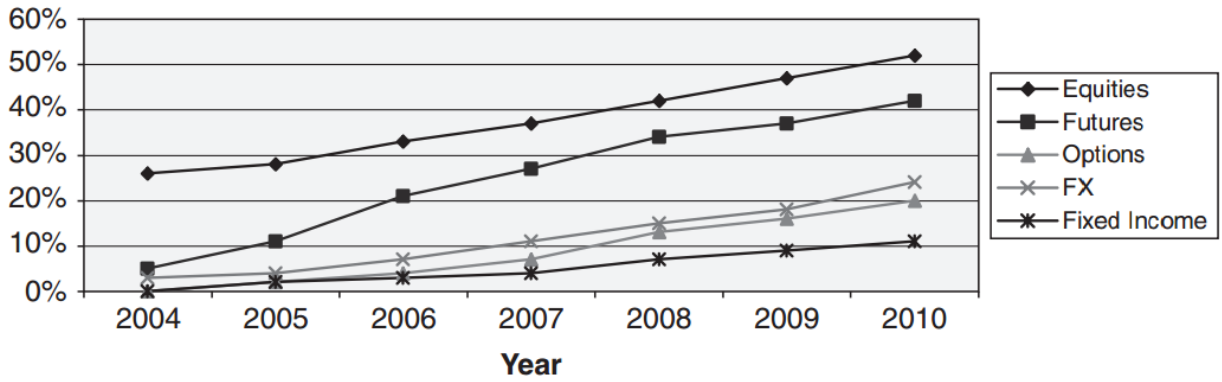


Trading time horizon & automation

Intraday Trading

Intraday trading is trading confined to a single day, most day – traders sell their positions before close.

Intraday trading is becoming the realm of machines, as they can search through much more data. Most machines are however trading on more or less instant strategies where the aim is to hold an actual position as short as possible, called High Frequency Trading. We can observe the trend of the adoption of algorithmic trading and, hence, intraday trading in the following graph.



Adoption of algorithmic execution by asset class.
Source:Aite Group.

Back testing

The process of testing a trading strategy on prior time periods. Instead of applying a strategy for the time period forward, which could take years, a trader can do a simulation of his or her trading strategy on relevant past data in order to gauge the its effectiveness.

When you backtest a theory, the results achieved are highly dependent on the movements of the tested period. Backtesting a theory assumes that what happens in the past will happen in the future, and this assumption can cause potential risks for the strategy. MACD is an algorithm that belongs to the class of Backtesting algorithms and is given below.

Methodology

Our aim in this project has been to prove the efficiency of parallelization in algorithmic trading by implementing the following algorithms:

1. MACD – Moving Average Convergence Divergence
2. Monte Carlo simulation to compare the result of any random trading methodology with our MACD implementation.

These methods are explained in more detail in the following sections.

1. Moving Average Convergence Divergence:

Moving Average Convergence algorithm is a trading indicator that is used in the analysis of stock prices. With this algorithm, we'll be able to predict changes in the direction, momentum and duration of that particular trend in stock's price. MACD oscillator is a collection of three time series calculated from historical price data, often referred to as closing price. The three series are the MACD proper series, the average series and the divergence series that is the difference between the two. The MACD series is the difference between a fast (short period) Exponential moving average (EMA), and a slow EMA of the price series. To clarify what each term signifies:

- *SMA*: Simple moving average (SMA) is an arithmetic moving average that is calculated by adding the closing price of a stock for a number of time periods and dividing this total by a number of time periods. Short-term averages respond quickly to changes in the price of the underlying, on the other hand, long-term averages are slow to react. The average is calculated is over a period of time. Equal weighing is given to each daily price. Traders watch for short-term averages to cross the long-term averages to signal the beginning of uptrend.

- *EMA*: Exponential moving average (EMA) is a type of infinite impulse response that applies weighing factors that decrease exponentially. The weighting for each older datum decreases exponentially, but never reaches zero.

EMA for a series is calculated by the following:

$$\textbf{Precondition: } S_1 = Y_1$$

$$\textbf{For } t > 1,\ S_t = \alpha * Y_{t-1} + (1-\alpha) * S_{t-1}$$

Where,

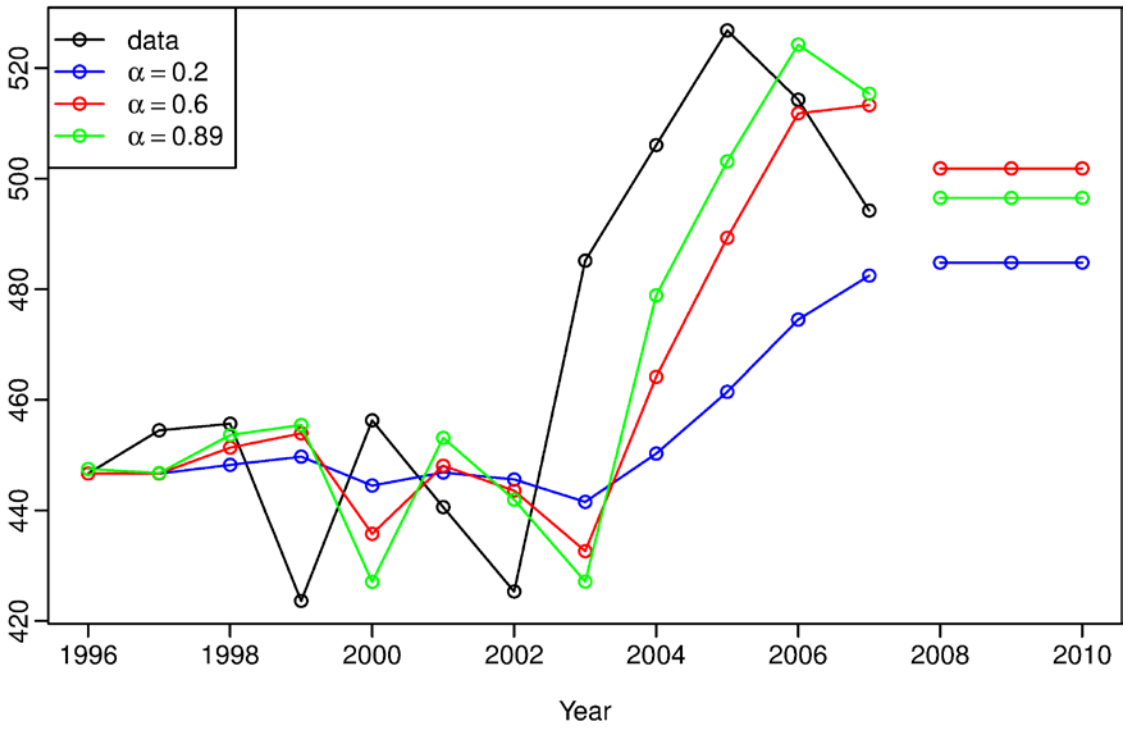
S_i = Value of EMA for i'th time period,

α = degree of weight for the current day, or the smoothing factor between 0 and 1.

Y_i = Value of the stock price at the i'th date

A higher 'α' will reduce the weight of older observations quickly. S_1 might be initialized in many ways, most common way in initializing S_1 is setting it to Y_1.

In the figure below, we show the effect of different values of α. As we can easily observe, higher values of α smoothens the curve further, thus reducing any jarring variations in the stock value.



Our MACD depends on three time parameters, the combination of which will generate a potentially unique signal. Each parameter combination will lead to buys and sells that adapt to different market conditions. No single parameter combination can work for all stocks and at all time periods. These 3 parameters are:

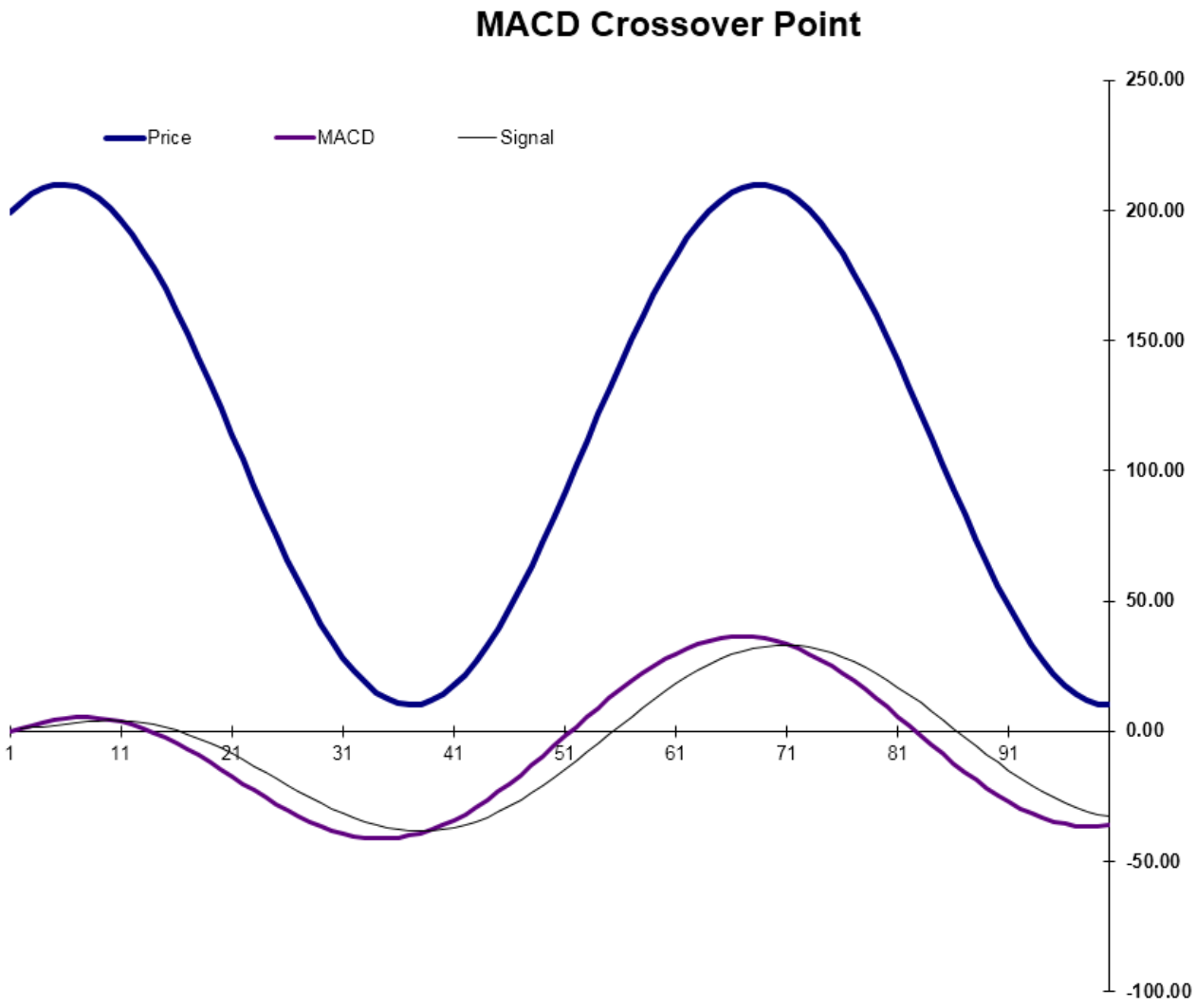
- Short – Signifies the period for the shorter moving average. We term this as "b".
- Long – Signifies the period for the longer moving average. Termed as "a".
- Period – This will be the EMA calculated for the signal generated by the difference of short and long, generated as explained above. This is called "c" in our implementation.

These parameters are usually mentioned in days. The divergence series, which is the difference of slow EMA and fast EMA can reveal subtle shifts in the stock's trend.

Parameter exploration is a computationally expensive and time-consuming exercise. Due to the wide array of possibilities in the combination of these parameters, the search for the perfect set of parameters which maximize earnings is exhaustive. Hence, our main concern would be to parallelize parameter exploration. We look at different methods through which we could achieve this goal and describe the most favorable solution.

One point to be noted here is that 'c' is a parameter that evaluates a function that is also dependent on the resulting signals generated by "a" and "b". Also, each point in the EMA calculation is dependent on the previous point. Hence the problem is tricky to solve, when seen from a parallel programming perspective.

Using these parameters, a final signal is generated, the MACD signal, based on the EMA of the difference in the Long and Short trend lines. This signal crosses over the Long-Short EMA signal itself at several points. These points represent buy and sell recommendations for a trader following this strategy. An example graph for this cross-over is given below.



Here, the Price line indicates the stock closing value. The signal line is computed by difference in EMA of the Long and Short signals. The MACD line is computed by taking the "c" days EMA of the Signal line. Wherever there is a positive cross-over (MACD line crosses signal and exceeds its value), a buy occurs. To identify a sell, the process is reversed.

2. **Monte Carlo Simulation for evaluating algorithm's performance**

To evaluate the result of any trading strategy, we would have to prove that this method performs better than any set of random trading strategies taken independently. One technique for such

evaluations is that of the Monte Carlo simulation. In this method, we use a random number generator that is compatible with parallel threads, to decide what action should be taken on a given day. These actions, cumulatively, will decide the reward for that run of the Monte Carlo simulation. This can be explained pictorially in the following manner.

Let us assume that we are running this technique on a stock with following Open (stock price at the beginning of the trading session) and Close (stock price at the end of the trading session) prices for 5 days:

DATE	OPEN (O)	CLOSE (C)
Day 1	x0	y0
Day 2	x1	y1
Day 3	x2	y2
Day 4	x3	y3
Day 5	x4	y4

For any day, the action we take based on the result of the random number generator is tabulated below:

Random value	Case	Reward
1	LONG	O-C
2	NEUTRAL	ZERO
3	SHORT	C-O

Here:

- LONG: Buy at OPEN price and sell at CLOSE price.
- NEUTRAL: No action is taken
- SHORT: Buy at CLOSE price and sell at OPEN price.

Based on a random number generator, we'll be generating a random value that decides our case and reward. When a random value of '1' is generated, LONG will be selected and reward is updated accordingly. When a random value of '2' is generated, NEUTRAL will be selected and reward has nothing to update. Similarly, when a random value of '3' is generated, SHORT will be selected and reward will be updated accordingly.

On a sample run for our hypothetical stock returns:

DATE	OPEN (O)	CLOSE (C)	Random Number	Action	Daily Reward
Day 1	x0	y0	1	Long	y0 - x0
Day 2	x1	y1	3	Short	x1 - y1
Day 3	x2	y2	2	Neutral	0
Day 4	x3	y3	2	Neutral	0
Day 5	x4	y4	1	Long	y4 - x4

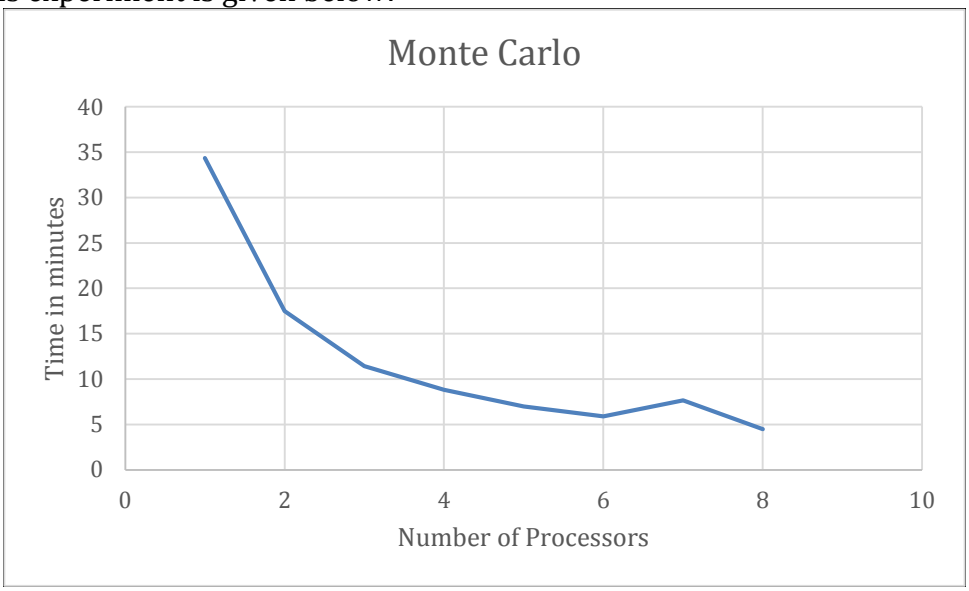
The cumulative reward is the sum of all daily rewards. We then calculate the percentage of gain achieved on this run by comparing the initial buy price with the final reward.

Parallel Approaches

In this section we describe the techniques we used to gain parallel performance, with the results and any associated analysis if relevant.

Parallelizing Monte Carlo

Parallelization of the Monte Carlo simulation is made easy by the fact that there is no dependency of one run on any of the other runs. Hence, our methodology to improve performance in this case was to use OpenMP's parallel "for" pragma for each run. This gave us hefty parallel gains. The results of this experiment is given below:



Parallelizing MACD Parameter Exploration

To improve the performance of the parameter exploration code is not as trivial as that of the Monte Carlo Simulation, due to the dependencies of the parameters and the fact that each value in the EMA is dependent on the previous value. There is also slightly more leeway to experiment with the data to achieve optimal results.

There are many ways to distribute work in parallel fashion since we can manipulate the 3 parameters.

Approach 1 – Parallelize c:

As a first approach, we can compute "a" and "b" but distribute the calculation of "c" among the different processors, which can run in parallel, as each "c" is independent of the other. This will reduce workload on single processor and balance load such that every processor will be having enough work while the others are busy.

To do this, a single processor computes the EMA signal for parameters “a” and “b”. These results become shared as read-only objects to a set of parallel processors that each calculate a result based on a different “c”. An OpenMP call to parallelize the “for” loop for computing “c” will solve this problem. This method is pictorially represented below.

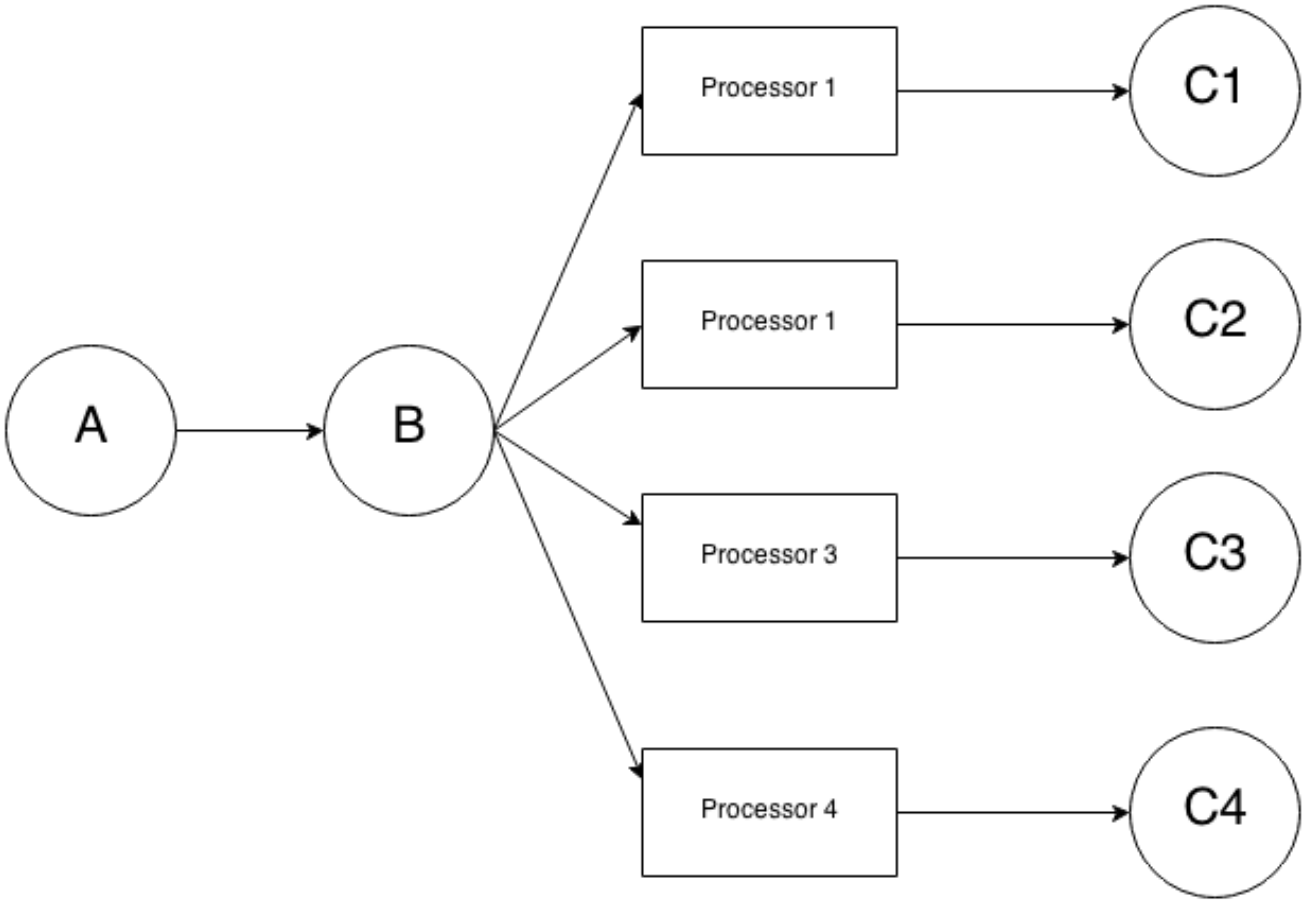
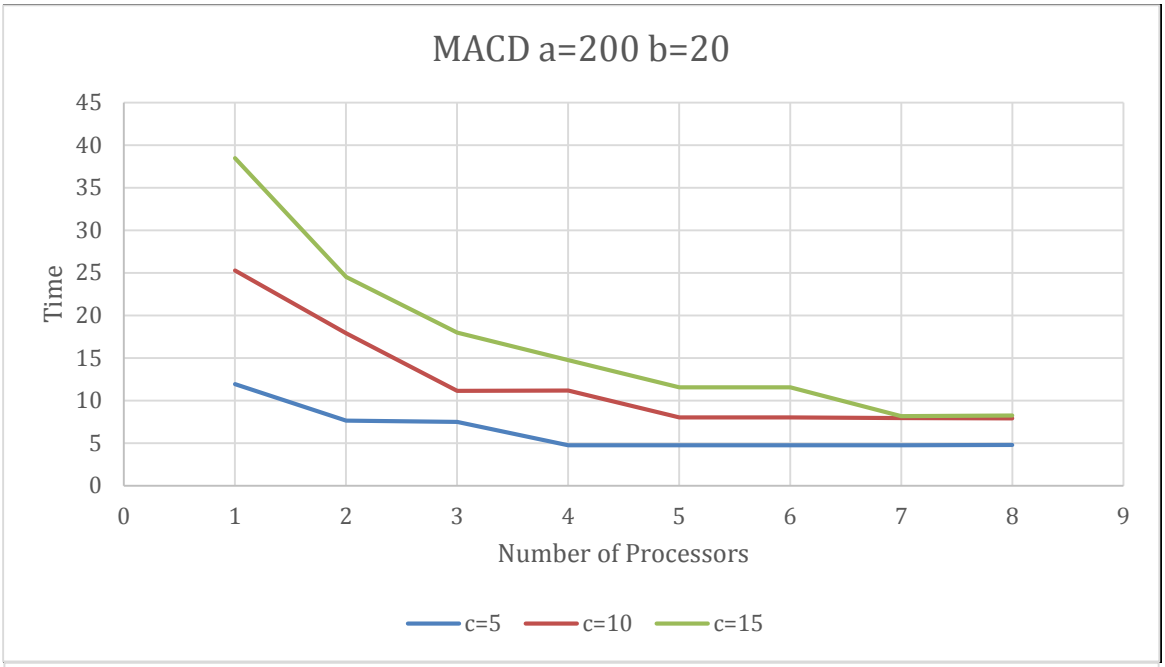


Figure 1 Parallelizing calculation of parameter "c"

Results:

Result of the run for exploring parameters from a = 0 to 200, b = 0 to 20 and c = 0 to 5, 10 and 15 is given below. As expected, increasing values of “c” takes increased amount of time since more combinations have to be tried. Parallel performance also increases with increasing values of “c” because the parallel component of the task increases.



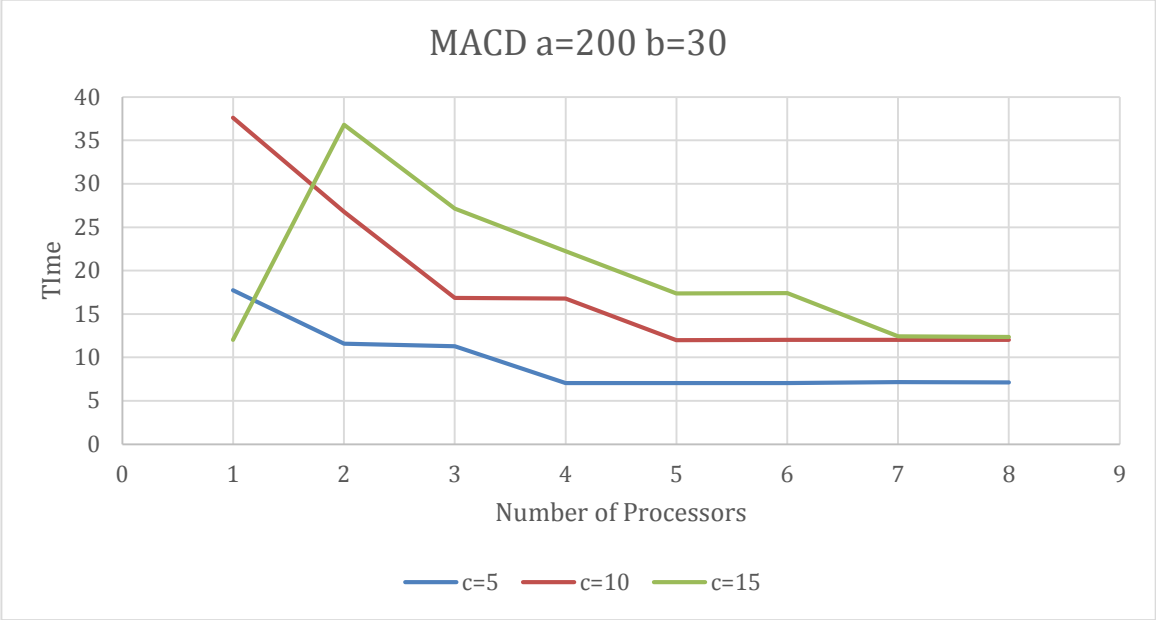


Figure 14Parallelizing C, Results of Two Runs

The results clearly prove that this method is effective. This is because, no data is shared as a writeable, common resource. Locks are not necessary to read values. Results are stored in independent memory locations. The EMA signal for various "c" values (c1..c4 in Figure 1) are used to compute cross-overs and generate buys and sells. Reward calculation also occurs in parallel.

Approach 2 – Compute "a" and "b" in parallel:

Since 'a' and 'b' are independent parameters, both can be calculated on different cores simultaneously. Results can be used in calculating 'c' (for different time periods of the difference signal EMA) on different cores similar to the one that was previously achieved.

Here, only the address of the locations of “a” and “b” are communicated after Processors 1 and 2 complete their tasks. Processors responsible for computing various values of “c” remain idle until this information is received.

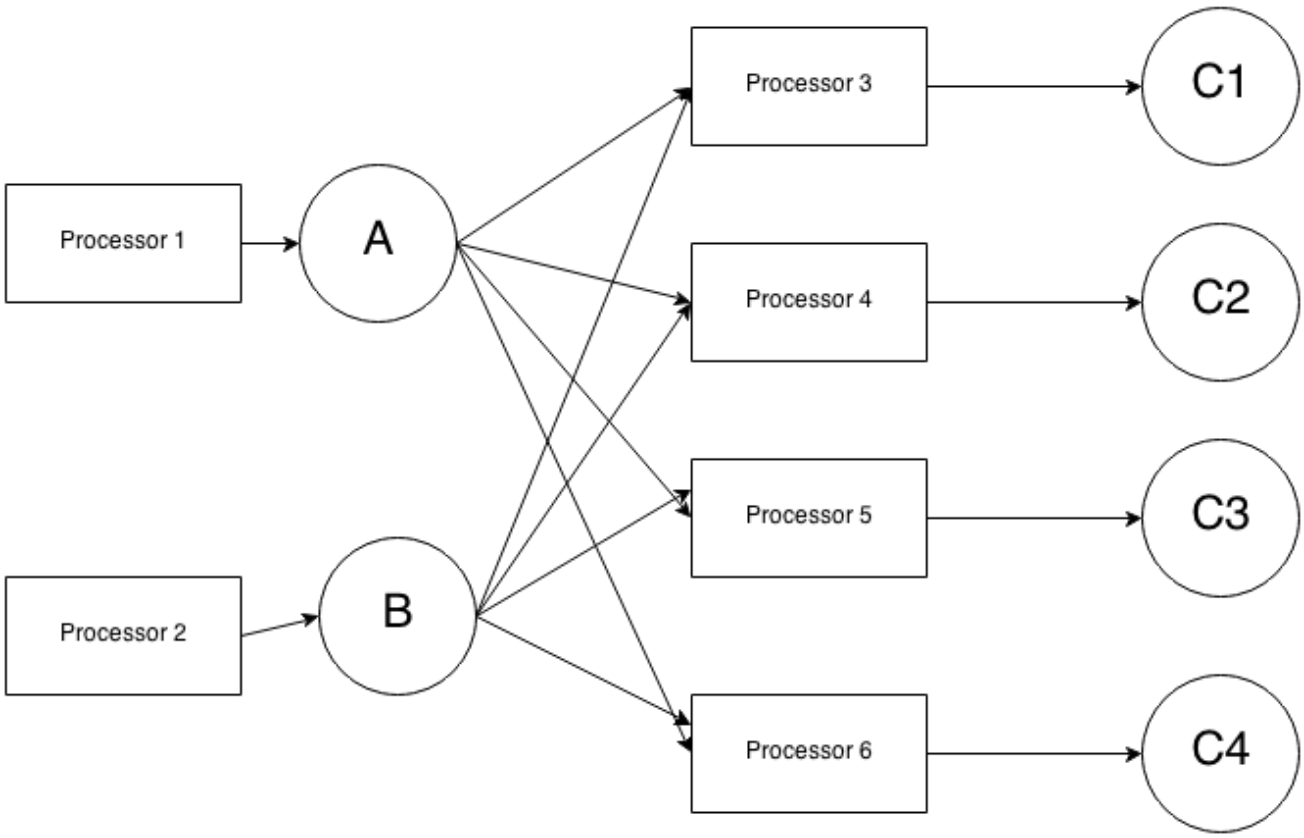


Figure 2 Parallelizing computation of "a" and "b", along with "c"

Results:

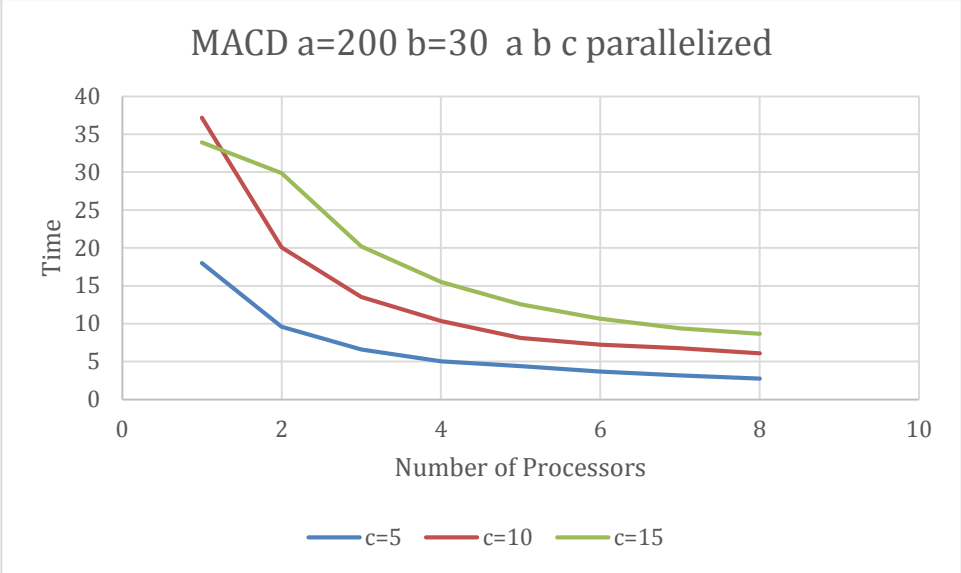


Figure 3 Parallel Performance of A, B and C parallelized

This did not perform as well as the previous technique due to the communication overhead of having to communicate the location of signals A and B to each processor. The earlier method did

not have to receive a signal to start processing and thus proves slightly more efficient. It can be inferred that parallelizing the computation of A and B together is not going to achieve the necessary performance increase.

Approach 3 – Parallelize "b" and "c":

Computation of 'a' can be done on one core and b and 'c_x' can be computed on other different cores. This is represented in the diagram below:

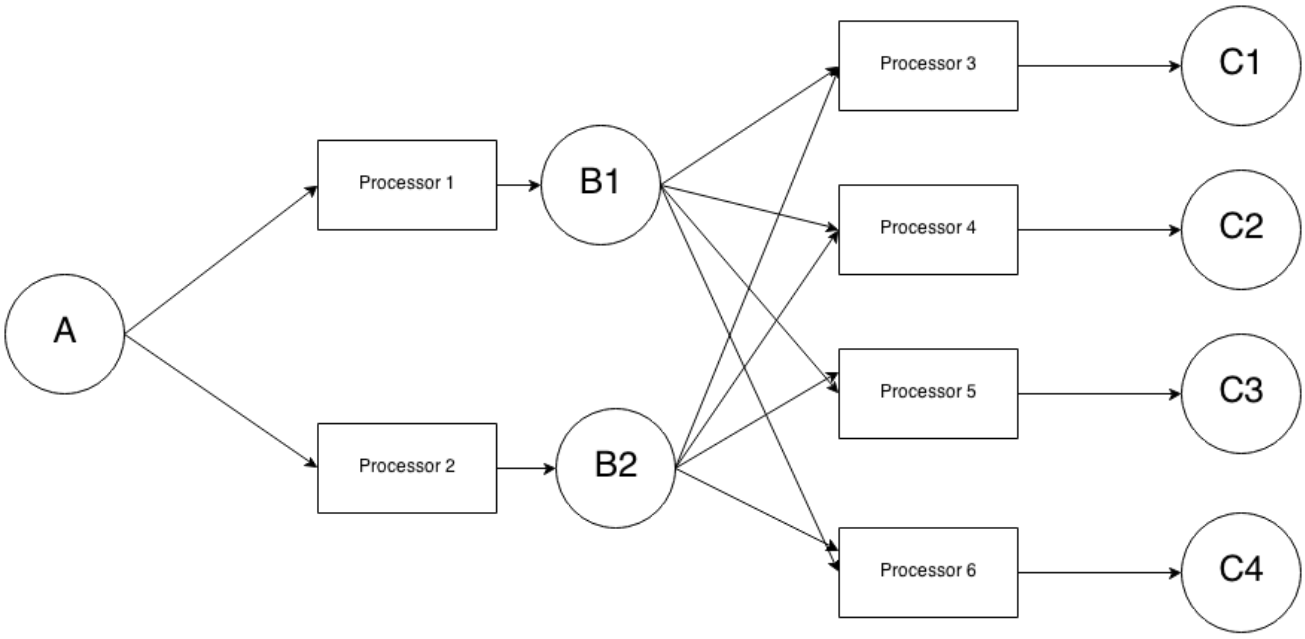
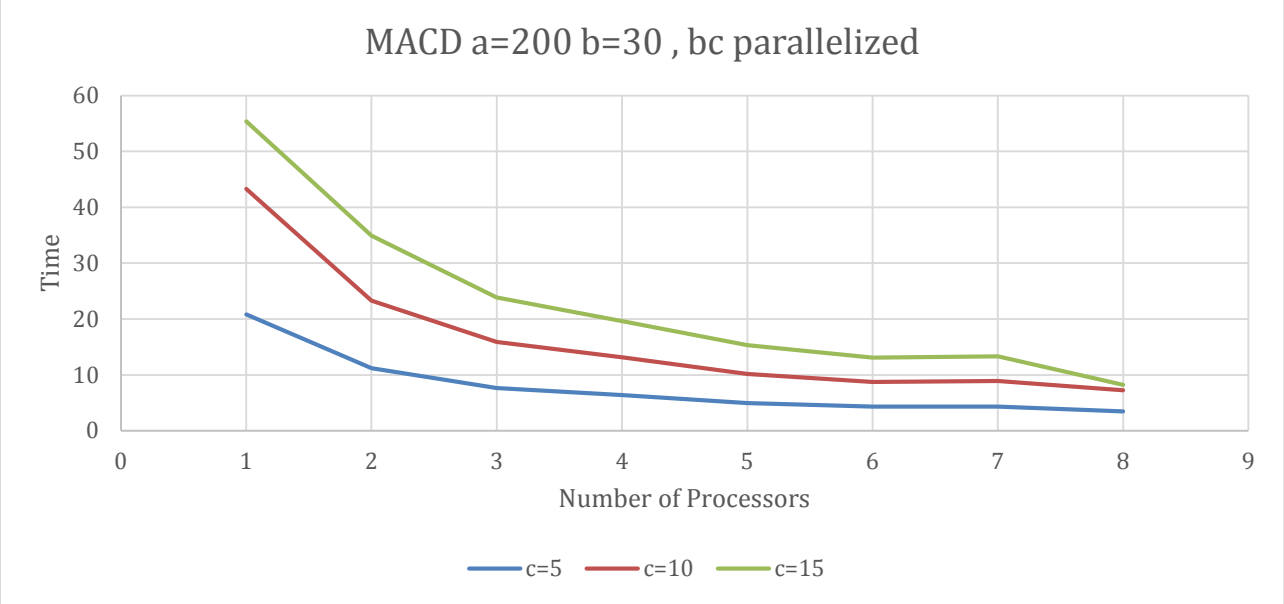


Figure 4 Parallelizing computation of "b", along with "c"

Results:



This result has been most favorable because of the fact that "b" is a parameter that is of much higher value than "c" and there is much scope for parallelization. The serial component of this code is minimal, since only A is computed at one processor. The MPI version of communication is not apt in this case, and the OpenMP "for" loop parallelization has proven to give much more favorable results because several threads are created, each of independent nature and can utilize the CPU most efficiently.

The MPI version of this parallelization lacked load balancing, since B1 and B2 computation is relatively inexpensive, compared to the steps that a processor must take, after computing the difference signal, as is the case when it has to compute the MACD signal using parameter "c".

Approach 4 – Memory Reorganization:

To achieve better ratio of cache hits, we decided to reorganize the memory to eliminate data that is of no consequence to the calculation of the MACD signal, like the date of the stock value. This approach can be best explained with the help of a figure that describes the current organization of the structure that is both the result of, and is used to, compute EMA.

The current moving average structure is composed of the fields given below. The main area of focus here is that of the Moving Average structure array. This structure's internal representation is given in Figure 5. The date and value portion is contiguous and leads to injudicious usage of cache when more values could be stored for computation of EMA.

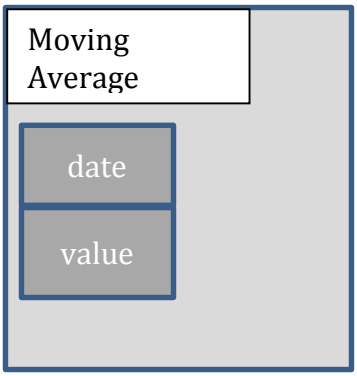


Figure 13 Moving Average Internal Representation

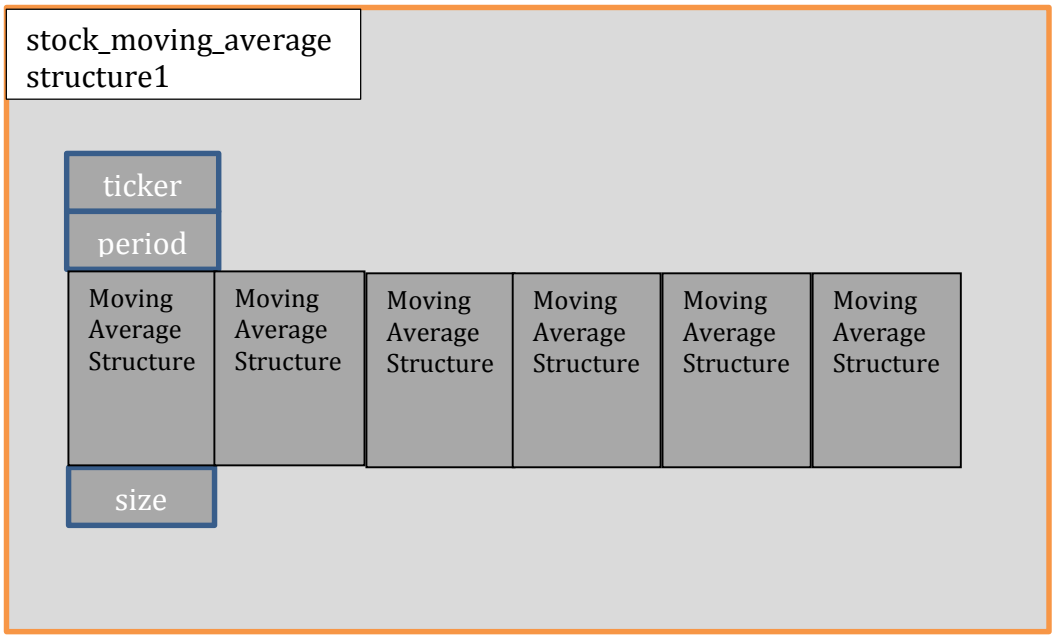


Figure 5 Structure of Stock Moving Average

Temporal locality is the same elements being accessed again and again. Since the calculations involving 'c' are based on 'a' and 'b'. Once 'a' and 'b' are computed, they are stored in the cache and repeatedly accessed in order to compute 'c'. Spatial locality is the fact that elements are

accessed sequentially. In our case, after the memory reorganization, all values are stored contiguously and in neighboring locations.

Results:

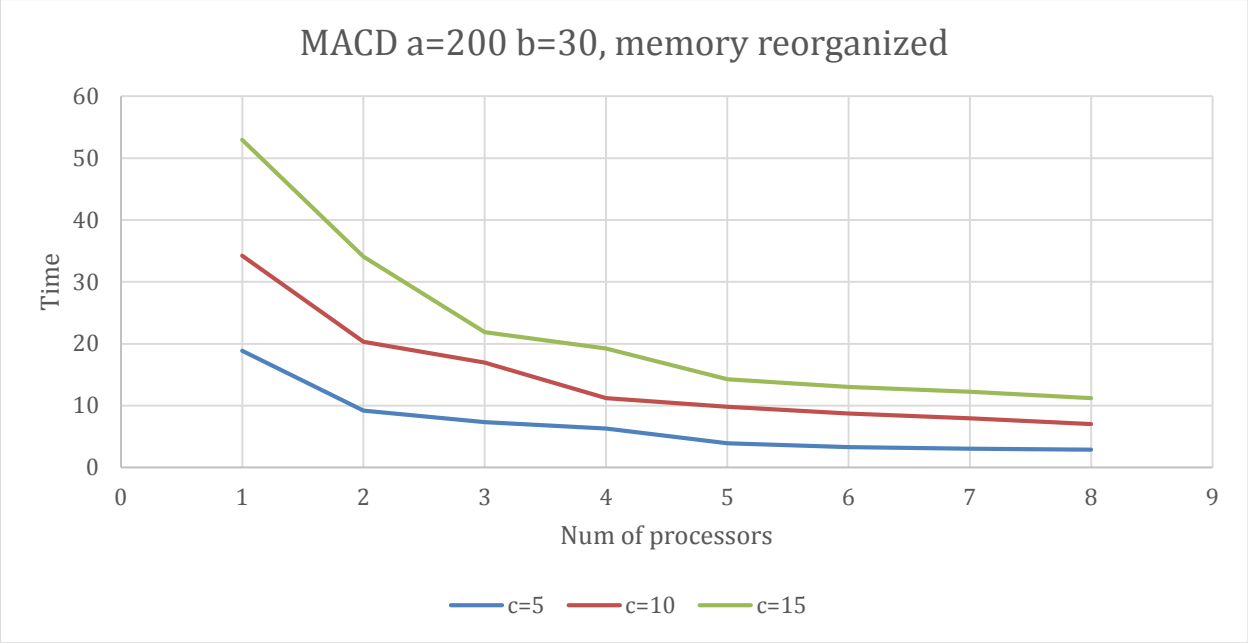


Figure 13 Memory Reorganization Performance

Approach 5 – Parallelize across stocks:

This approach, as can be predicted due to the obvious cache eviction issues, performed badly when parallelized. There are also issues with load balancing, since the stocks are of varying durations, if run without specifying a common start and end date. Some processor might be heavily loaded while another processor remains idle after completing its share, since each missing date could add a lot of time saved over several iterations.

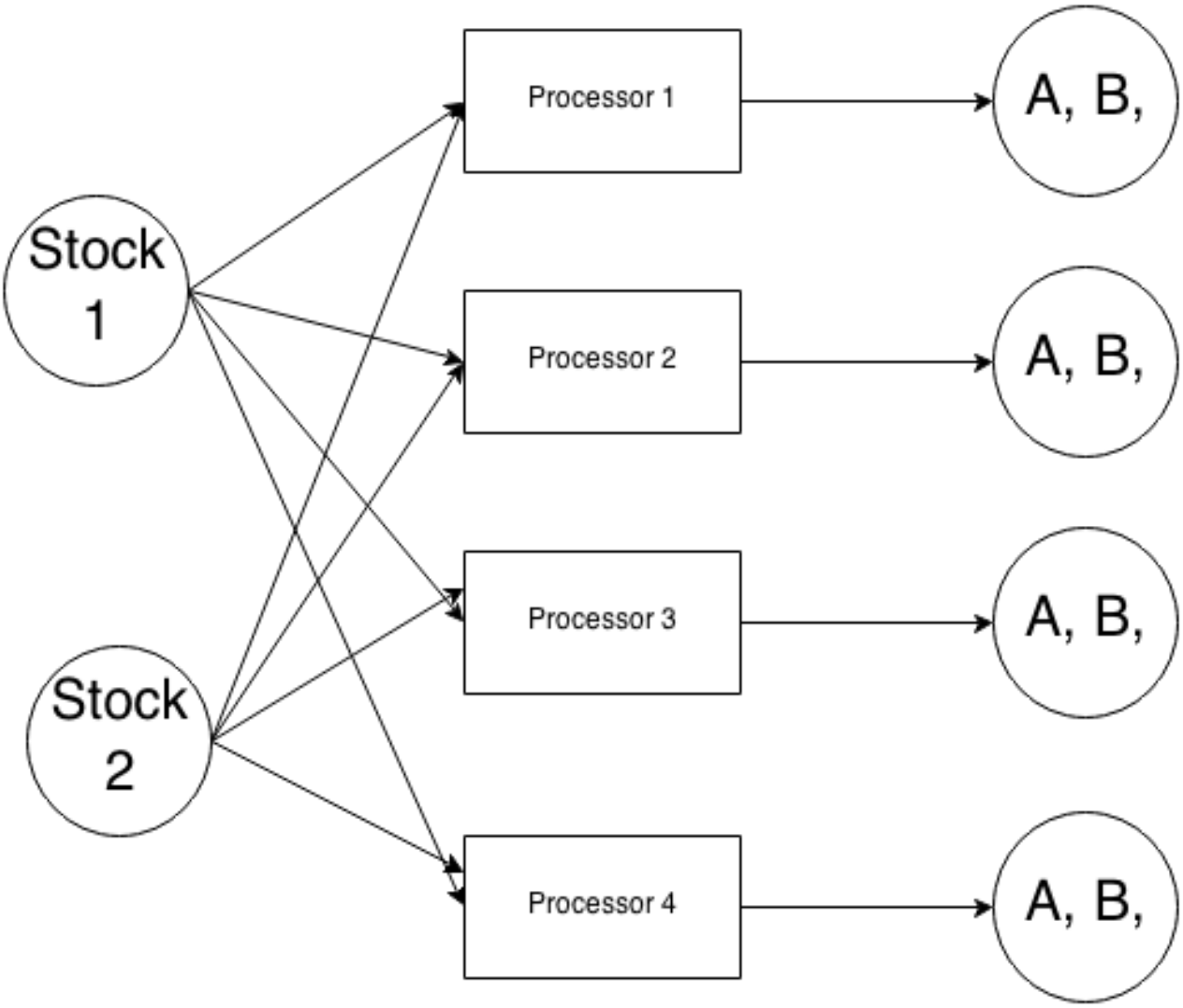


Figure 6 Parallelize across stocks

Approach 6 – Parallelize by reusing Average values:

In this approach, we try to reuse average values by computing EMA in parallel for various values of "b". As "b" increases sequentially, communicating the first average computed by the previous processor will allow the current processor to avoid re-computing the average for the first point, when calculating EMA. The rest of the computation is similar.

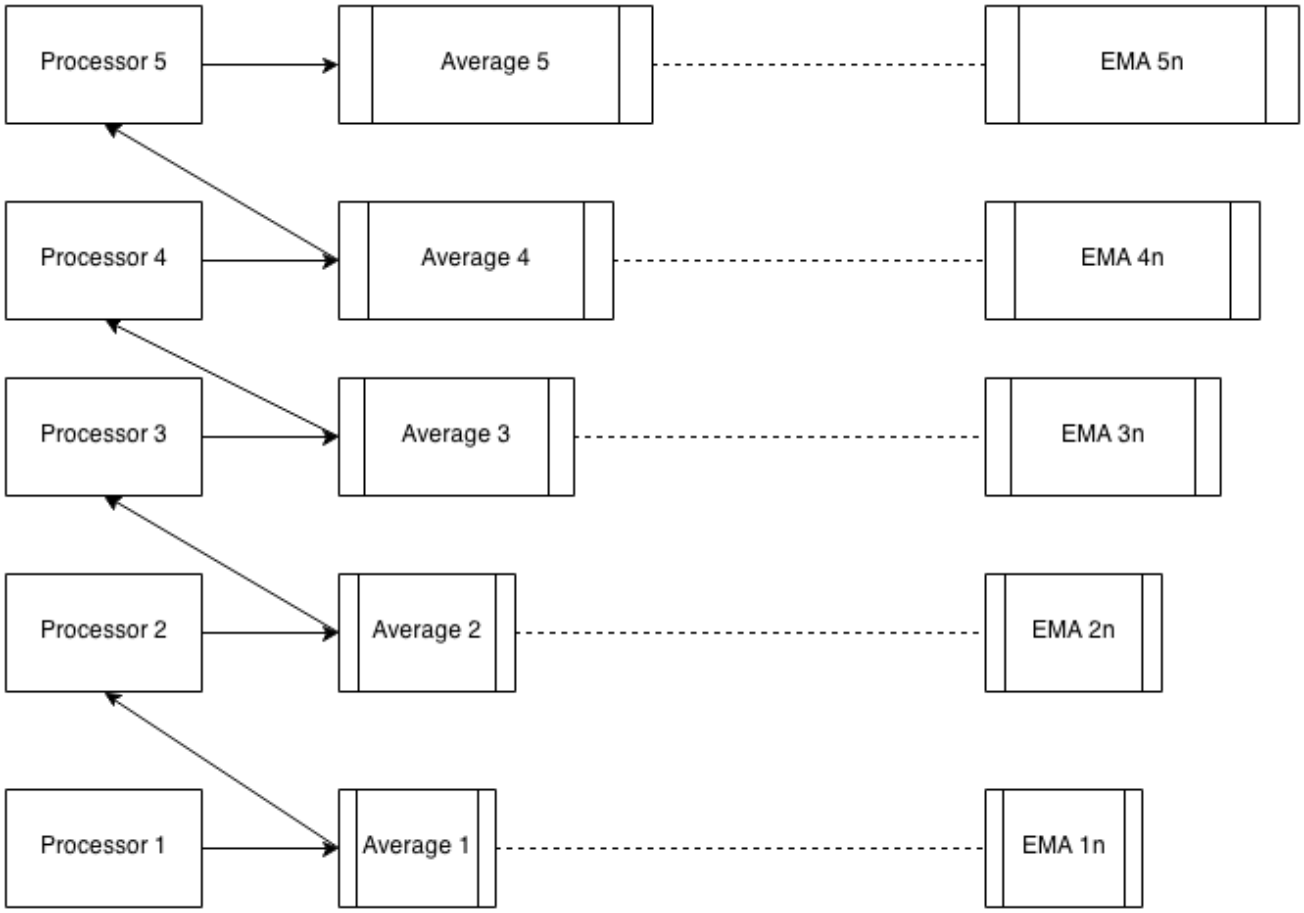


Figure 7 Parallelization by reusing average

Results:

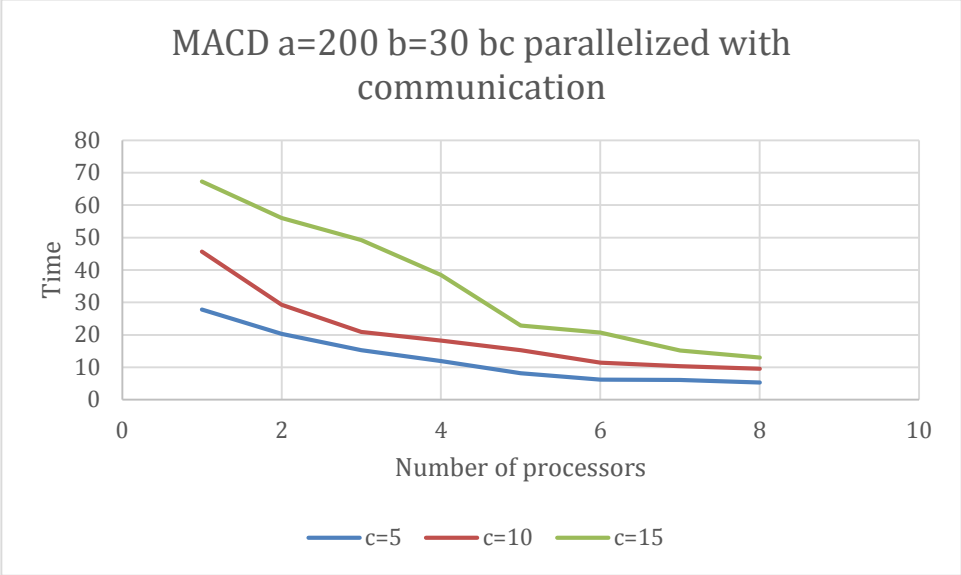


Figure 8 Parallel Average Communication

As per the graph above, we can clearly see that this method is not efficient at low values of "b". Its efficiency increases, however for higher values of "b" as the cost of communication becomes negligible while the cost of computation of the first average becomes costly.

Profiling Results

1. Monte Carlo simulation:

1.1 Monte Carlo Serial: Monte Carlo simulation is based on the random analysis over a period of time. However, we'll be carrying out this analysis for multiple iterations to find out the best possible average. If we run code serially on a single processor, it takes lot of time as shown by the profiler since we are running multiple iterations on a single core as shown below

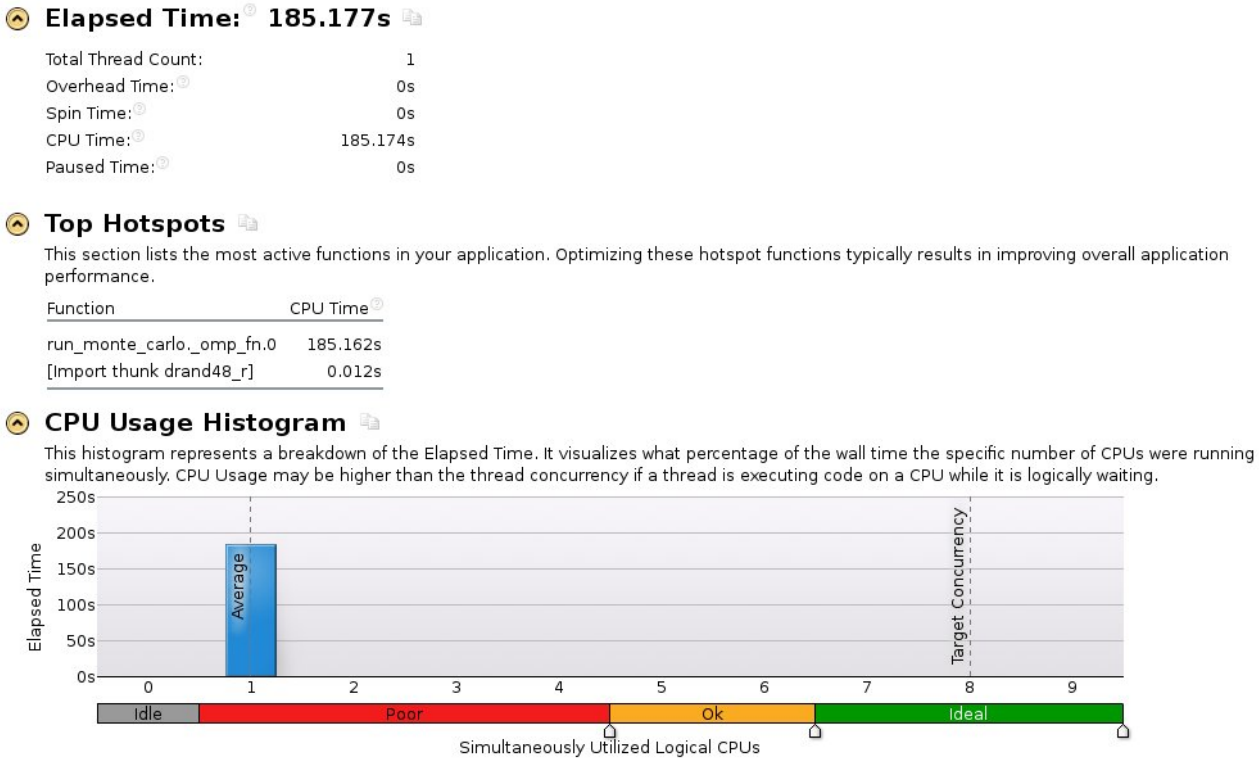


Figure 9 Monte Carlo Serial Profile

1.2 Monte Carlo Parallel: Monte Carlo simulation over a period of time is calculated for multiple iterations. In our parallel implementation, we are running various these multiple iterations on different cores. This way, we are able to reduce computation time as shown as by the profiler. Different cores are working in parallel and reduce amount of time required for computation. Some of the cores take more amount of time than the other because we are having many initializations before the actual parallel processing starts.

Elapsed Time: 36.599s

Total Thread Count:	8
Overhead Time:	0s
Spin Time:	0s
CPU Time:	278.850s
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
run_monte_carlo._omp_fn.0	277.028s
[Import thunk drand48_r]	1.814s
func@0x9c40	0.008s

CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically waiting.

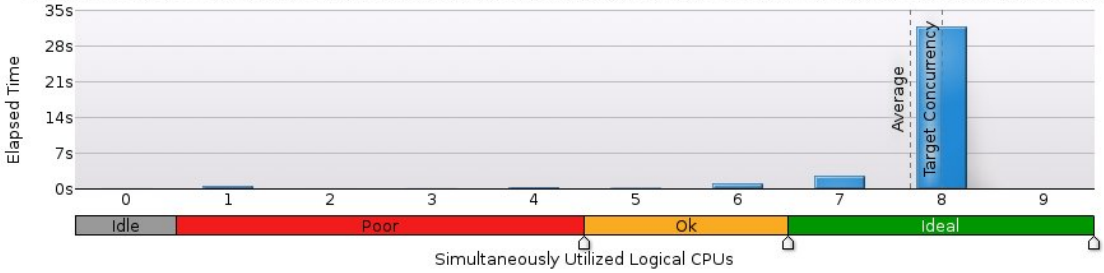


Figure 10 Monte Carlo Parallel Profile

2. MACD:

2.1 Serial: Since 'c' is a function which is evaluated by taking difference between 'a' and 'b'. As discussed earlier, c_1 c_2 etc can be done simultaneously on different cores. However, in serial implementation we are calculating c_1 c_2 on a single core.

Elapsed Time: 79.705s

Total Thread Count:	1
Overhead Time:	0s
Spin Time:	0s
CPU Time:	79.694s
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
subtract_ ma	28.658s
calculate_moving_avg_from_ ma	22.244s
find_price_at_date	13.672s
get_reward	12.256s
[Import thunk strcmp]	1.024s
[Others]	1.840s

CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically waiting.

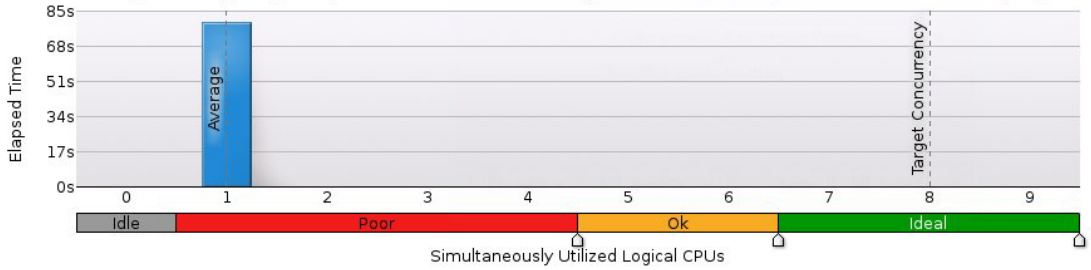


Figure 11 MACD Serial Profile

2.2 Parallel: Once single 'a' and 'b' is calculated, these are accessed from the cache for calculating c_1 c_2 etc. Following figures show the effect of parallelizing these calculations.

Elapsed Time: 23.654s

Total Thread Count:	8
Overhead Time:	0s
Spin Time:	0s
CPU Time:	166.100s
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
subtract_ ma	65.111s
calculate_moving_avg_from_ ma	43.834s
find_price_at_date	24.192s
get_reward	22.130s
destroy_ ma	3.138s
[Others]	7.695s

CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically waiting.

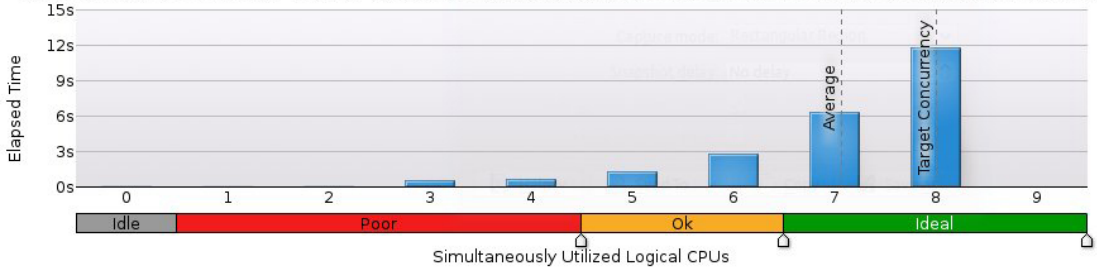


Figure 12 MACD Parallel Profile

References

1. Jeffrey Owen Katz, Donna L. McCormick, Developing systems with a rule-based approach, Stocks & Commodities, Vol 15, 1997, pp. 26-34.
2. Jeffrey Owen Katz, Donna L. McCormick, Sunspots and market activity, Stocks & Commodities, Vol 15, 1997, pp. 401 - 406.
3. Perry J. Kaufman, Trading and System and Methods, John Wiley & Sons, 1998.
4. Michael Harris, Stock Trading Techniques, Trades Press Inc, USA, 2000.