Stock Analysis Technical Indicators Calculation

Abstract

This project is attempting to utilize hybrid programming technology to calculate stock analysis technical indicators under parallel computing architecture. Message Passing Interface (MPI) and parallel computing framework will be combined to deliver fast calculation performance.

1. Introduction

Everything is expensive. This reflects today's global society due to high inflation. As inflation grows rapidly, people find their disposable income decreases every year. King (2011) highlighted consumer prices go up every month and reduce consumer’s purchasing power and investor’s real interest rate (King, 2011, p. 15). People are trying to find different ways to protect their wealth. One of the options is passive income source. People are trying to find different ways to protect their wealth. One of the options is a passive income source. Birken (2023) listed out 5 options from an investing perspective. This study focuses on stock investing.

This project leverages the power of the Message Passing Interface (MPI) to efficiently and parallelly collect stock history data and calculate technical indicators EMA, RSI and MACD.

By distributing the data collection task across multiple nodes, this project aims to significantly reduce the time required to gather large volumes of stock history data. A more comprehensive and timely analysis will ultimately provide more accurate and actionable investment insights.

2. Applied Problem

Using serial programming to collect multiple stock price histories and calculate financial technical indicators can lead to several challenges and limitations. Here are some potential problems:

1. **Performance and scalability**: Large datasets and complex computations are not in favour of serial programming due to sequential execution. Processing multiple stock price histories and calculating technical indicators can be computationally intensive and time-consuming. Serial execution may not fully utilize computing resources and result in slower analysis.
2. **Longer computation time**: Serial programming requires sequential calculations, resulting in longer computation times. It can be a major disadvantage when dealing with real-time or high-frequency data.
3. **Limited parallelism**: Serial programming does not support parallel execution. Software can't take full advantage of modern hardware architectures, such as multi-core CPUs or GPUs. This limitation can hinder efficient and parallel computations, especially for large datasets.

4. **Inefficient memory usage**: Serial programming may not optimize memory usage efficiently. When dealing with large datasets, memory management becomes critical. Inefficient memory usage can lead to increased memory consumption, slower execution, and potential memory-related errors.

3. Applied Solution

To address the constraints posed by traditional programming methods, the utilization of a hybrid approach which integrates the use of MPI for data collection and CUDA for parallel computing has proven to be advantageous. The implementation of MPI facilitates efficient data collection from various sources through distributed computing, while CUDA's parallel computing framework allows for the scaling of computational capabilities across multiple GPU cores. This combination of techniques has shown promising results in accelerating the computation of technical indicators, making hybrid programming a valuable tool for optimizing performance in data-intensive applications.

By utilizing the power of both MPI and CUDA, this integration offers a powerful solution for handling complex and data-intensive tasks, ultimately leading to improved decision-making processes in the financial sector. Kijsipongse, E. et al. (2011) proved that hybrid programming with MPI and CUDA delivered great performance for large dataset and complex computation with scalable grid computer architecture.

The following points are why this approach is beneficial for collecting multiple stock price histories and calculating technical indicators.

3.1. Efficient Data Collection

Message passing interfaces (MPIs) are portable message-passing systems for parallel computing. It allows for the distribution of tasks across multiple nodes, significantly reducing the time required to gather large volumes of stock history data from multiple sources. This is particularly beneficial when dealing with extensive historical data from multiple stocks.

3.2. Fast Calculations

Since EMA and RSI computations require utilizing the closing prices from the preceding time window, conventional sequential programming approaches are commonly employed. While these methods are functional, they tend to consume excessive amounts of time and memory. By leveraging MPI (Message Passing Interface) technology, it is possible to distribute the computation duties among distinct securities while working within restricted resource constraints and timelines.

NVIDIA's CUDA architecture enabling GPU utilization provides a powerful parallel computing platform. GPUs support can be used as general-purpose computers by software developers. When calculating technical indicators without data dependency on previous time windows like Moving Average Convergence Divergence (MACD), CUDA can perform these complex computations much faster than a CPU due to its parallel processing capabilities.

3.3. Scalability

As the volume of data increases, we can scale up the cluster with more CPU and GPU cores with MPI capabilities to maintain performance.

3.4. Flexibility

Hybrid programming provides the flexibility to use the most suitable tool for each task. MPI is excellent for data collection across distributed networks, while parallel computing is perfect for performing high-speed computations on the collected data.

By using a hybrid approach, the algorithms can be performed more timely and comprehensive analyses of stock price histories and technical indicators, ultimately leading to more accurate and actionable insights for investment decisions.

4. Proposed Design

The proposed solution for this project involves three stages:

1. Downloading the S&P top 50 list
2. Collecting stock price historical data using MPI
3. Calculating EMA and RSI using MPI
4. Calculating MCAD using CUDA

4.**1. Downloading the S&P top 50 List**

The first stage of this solution is to download the S&P top 50 list. This list will distribute to remote processes evenly using MPI framework. Each list item will include stock code and download date range as driving factor to download stock price historical data separately.

4.2. Collecting Stock Price Historical Data

The second stage is price history download. Once remote process receives the corresponding stock code and download data range. This process may use API call or other technique to complete the task. Download mechanism will be determined in project implementation stage. The details of target download file structure can see in ***Appendix 1***.

4.3. Calculating EMA and RSI using MPI

Due to the dependency on preceding closing prices, EMA and RSI computations are incompatible with CUDA, which relies on data independency. However, by leveraging the MPI framework, we can allocate various stock history calculations to simultaneous parallel processes and employ a conventional calculation method with data frame window rolling feature. This approach significantly enhances calculation time and efficiency.

4.4. Calculating MACD using CUDA

When conducting technical analysis of stock prices, EMA (Exponentially Moving Average), RSI (Relative Strength Index), and MACD (Moving Average Convergence Divergence) are critical tools used to determine trend patterns. However, the computation of these three indicators requires substantial amounts of time and resources, particularly when managing vast volumes of data. Regrettably, both EMA and RSI possess data dependence properties, resulting in impossibility utilizing CUDA architecture. Within this undertaking, we're restricted to implementing CUDA architecture solely for MACD calculations.

CUDA parallel computing enables complex calculations in a faster and more efficient manners with GPU power. This can significantly reduce processing time.

In parallel computing, the assignment of processes to GPUs plays a significant role in achieving this goal. The use of modulus of process rank and the number of GPUs ensures an even distribution of processes across the available GPUs, regardless of the varying numbers of processes and GPUs. This approach not only maximizes the utilization of GPUs but also prevents overloading of any particular GPU. Additionally, the use of the MPI Barrier function call allows for synchronization of all processes, ensuring that they have completed their current iteration before moving on to the next one. This eliminates the possibility of race conditions and guarantees efficient utilization of the GPUs. Overall, this method of process assignment and synchronization is a vital aspect of parallel computing, promoting efficient resource utilization and enhancing overall system performance.

5. Technical Indicator Calculation

5.1. EMA

The Exponential Moving Average (EMA) is a type of moving average that is like the Simple Moving Average (SMA), but it differs in that it gives more weight to recent data points, making it more responsive to new information and price changes (Fernado, 2023, MA).

EMA is a technical chart indicator used to track the price of an investment over time. It is particularly popular among traders because it is more sensitive to recent price movements compared to the SMA. This sensitivity is due to the weighting mechanism of the EMA, which exponentially decreases the weight of older data points, placing a higher significance on the most recent ones.

$$EMA_t = \left[V_t \times \left(\frac{s}{1+d}\right)\right] + EMA_y \times \left[1 - \left(\frac{s}{1+d}\right)\right]$$

where

- $EMA_t = EMA\ today$
- $V_t = Value\ today$
- $EMA_y = EMA\ yesterday$
- $s = Smoothing$
- $d = Number\ of\ days$

5.2. RSI

The Relative Strength Index (RSI) is a widely used momentum indicator in technical analysis that measures the speed and change of price movements of a security (Fernando, 2023, RSI). It is designed to identify overbought or oversold conditions in the trading of an asset.

RSI is displayed as an oscillator (a line graph) that moves between zero and 100. The indicator is considered overbought when above 70 and oversold when below 30, suggesting a potential reversal in the current price trend. The RSI line crossing these thresholds can signal traders to buy or sell.

The calculation of RSI involves a two-part formula that starts with the following equation:

$$RSI_{\text{step one}} = 100 - \left[\frac{100}{1 + \frac{\text{Average gain}}{\text{Average loss}}} \right]$$

This formula compares the magnitude of recent gains to recent losses to determine the velocity and magnitude of price movements.

The second part of the equation involves calculating the actual Relative Strength (RS), which is the average of the number of periods of gains (up closes) divided by the average of the number of periods of losses (down closes) over a specified time frame.

$$RSI_{\text{step two}} = 100 - \left[\frac{100}{1 + \frac{(\text{Previous Average Gain} \times 13) + \text{Current Gain}}{((\text{Previous Average Loss} \times 13) + \text{Current Loss})}} \right]$$

5.3. MACD

The Moving Average Convergence Divergence (MACD) is a technical analysis tool used to identify changes in the strength, direction, momentum, and duration of a trend in a stock's price (Fernando, 2023, MACD).

MACD is a trend-following momentum indicator that shows the relationship between two exponential moving averages (EMAs) of a security's price. It consists of two lines: the MACD line, which is the difference between the 12-period EMA and the 26-period EMA, and the signal line, which is the 9-day EMA of the MACD line.

$$\text{MACD} = \text{12-Period EMA} \; - \; \text{26-Period EMA}$$

6. Implementation

6.1 Project Folder Structure

There are two methods for preparing the project folder. The first method involves uploading a zip bundle attached to the project submission record and extracting it into the user's home folder. The extracted folder is named "`cp631-final`", but it can be renamed.

The second method entails using the git tool to clone the source code from the student's public GitHub repository, if git is available in the server's operating system. The git clone command will look like this:

```
$> cd $HOME
$> git clone https://github.com/reidlai/cp631-final
```

The following setup is assumed that project root folder will be "`$HOME/cp631-final`"

6.2 Kaggle Authentication

In this project, we will download a dataset from Kaggle. Before beginning the download process, it is necessary to ensure an account on Kaggle available. If you do not wish to sign in and would rather bypass the login prompt by uploading your `kaggle.json` file directly instead, then obtain it from your account settings page and save it either `$HOME/.kaggle` with file permission `600` before starting this notebook. This way, you can quickly access any datasets without needing to log into Kaggle every time.

6.3 Environment Preparation

Miniconda is a lightweight and open-source package and environment manager. It simplifies the installation, management, and distribution of Python packages and their dependencies across various platform. This allows users to tailor their package collections based on their specific needs. Miniconda enables the creation of isolated environments, seamless switching between them, and easy sharing through portable archives or cloud services. Moreover, Miniconda enhances performance and usability by supporting fast and parallel package installation using its mamba engine. Overall, Miniconda provides a flexible and scalable solution for managing Python packages and environments, making it ideal for data scientists, researchers, and developers working with complex and diverse datasets and applications.

This section is assumed that `conda` package manager is ready in server operating system. If it is missing, please contact your operating system administrator for assistance.

First, a dedicated conda environment is required to run this project. The following command demonstrates how to create a conda environment called "cp631-final".

```
$> conda create -name cp631-final
```

There are two files can be utilized in project root folder to ease Python package installation. Before package installation, the new conda environment should be activated.

```
$> conda activate cp631-final
```

Then can run conda and pip3 tools to complete the environment setup.

```
$> $HOME/cp631-final
$> conda install --file ./conda_requirements.txt
$> pip3 install -r ./pip3_requirements.txt
```

6.4 Execution Steps

Once the above steps in section 6.1 and 6.2 are successfully completed, we are ready to run main program.

Although notebook `cp631_fina.ipynb` can be run in Jupyter engine, Jupyter can only support single process with MPI framework. It is recommended to use command line mode to run `cp631_final.py` (the exported version of notebook `cp631_final.ipynb`). Please follow the following command to run this program.

```
       $> mpirun -np 1 -mca
opal_cuda_support 1
~/miniconda3/envs/cp631-final/bin/python
~/cp631-final/cp631_final.py
       $> mpirun -np 2 -mca
opal_cuda_support 1
~/miniconda3/envs/cp631-final/bin/python
~/cp631-final/cp631_final.py
       $> mpirun -np 4 -mca
opal_cuda_support 1
~/miniconda3/envs/cp631-final/bin/python
~/cp631-final/cp631_final.py
       $> mpirun -np 8 -mca
opal_cuda_support 1
~/miniconda3/envs/cp631-final/bin/python
~/cp631-final/cp631_final.py
       $> mpirun -np 16 -mca
opal_cuda_support 1
```

The above six commands are very similar except the `-nb` parameters. This parameter indicating how many processes `mpirun` should be spawned. After completion of each command, a few data output files and one stat file for each command will be generated. The statistics files are key input in our later stage – Data Visualization and Performance Analysis.

6.5 Data Visualization and Performance Analysis

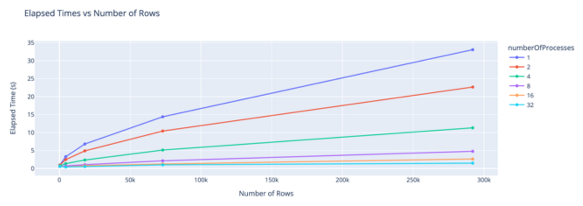
Since all graphics in this project are expected rendering in Jupyter notebook, it is not feasible to use command line mode for graphics rendering.

It is recommended to use Visual Studio code editor to run Jupyter notebook with ipython kernel connecting to remote Jupyter server. Here are the steps about these settings.

1. It is assumed that section 6.3 has been completed successfully including six `mpirun` commands.
2. It is expected the local environment has been installed with `ssh` tool. Assume your remote server is course server `mcs1.wlu.ca`. We can forward all Jupyter server traffic to local machine by "`ssh -L 8890:localhost:8890`"
3. Once login remote server, run "`cd $HOME/cp631-final; conda activate cp631-final`" to change current directory to project folder and activate the newly created conda environment.
4. Start Jupyter server by "`jupyter notebook –no-browser –port=8890`". This command will forward all Jupyter requests from ipython kernel local machine to remote Jupyter server. Please copy the server URL found in the console after successful start-up.
5. Open `cp631_final.ipynb` notebook in your local editor, e.g. Visual Studio Code editor and connect ipython kernel to remote server using the URL copied in the previous step 3.
6. In local version of `cp631_final. ipynb`, please scroll to Part 3 - Data Visualization and Performance Analysis and run each step until end of the notebook to see the result.

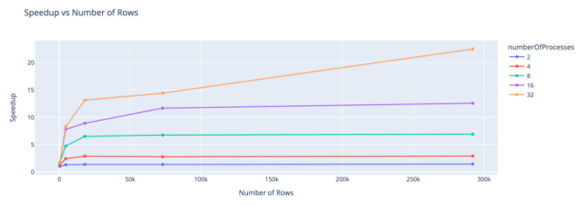
7. Result

7.1 Elapsed Times



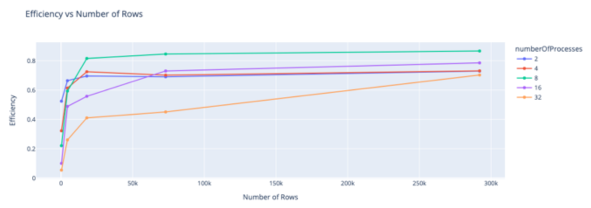
According to the line chart labeled "Elapsed Times versus Number of Rows," the program undergoes a remarkable boost in performance once eight processes are deployed, as demonstrated by the graph. Furthermore, the chart implies that while spawning 16 or 32 processes, there appears to be little discernible decline in processing time.

7.2 Speedup



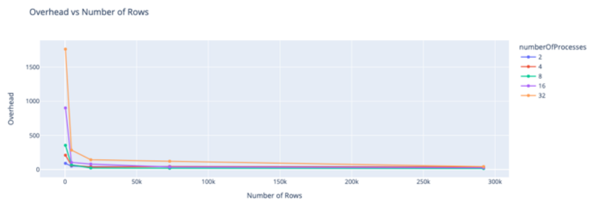
The graph displayed below exhibits an intriguing trend. With the exception of the instance involving 32 processes, the line representing speedup remains horizontally flat when dealing with 73,000 rows. This phenomenon occurs more promptly when the number of processes does not exceed 8. Upon analyzing the quantity of rows, it becomes apparent that 73,000 rows divided by 16 processes yields approximately 4,563 rows per process, whereas 292,000 rows split by 32 processes equates to roughly 9,125 rows per process. These figures indicate that beyond a certain threshold known as the parallel overhead breakpoint, which lies somewhere between 4,563 rows per process and 9,125 rows per process, reducing parallel overhead results in a considerable reduction in processing requirements. Essentially, if we possess sufficient rows per process, we can significantly lessen parallel overhead.

7.3 Efficiency



This graph corroborates the assertion put forth in Section 7.1. The analysis reveals that when the number of processes stands at eight, the program attains optimal efficiency compared to all other tests carried out. Moreover, the parallel efficiency demonstrates a significant improvement when handling more than 73,000 rows in circumstances where thirty-two processes are active.

7.4. Overhead



As illustrated in the overhead graph, the program has the ability to substantially diminish parallel overhead regardless of the testing environment whenever the number of rows surpasses 18,000.

8. Follow-up Problems

8.1. Max Threads

While carrying out program execution, it became evident that the application failed to create threads when attempting to utilize sixty-four processes simultaneously. This issue might stem from constraints imposed by the course server settings aimed at apportioning system resources fairly amongst multiple users enrolled in the same course. Consequently, the scope of our sampling size will now be confined to a maximum of thirty-two processes.

8.2 CUDA Toolkit Outdated Version

While executing the program concurrently with both MPI and CUDA, an exception appeared, yet it did not affect the outcome. Interestingly, when utilizing CUDA independently, there were no issues encountered. Following investigation within the open-source community, it is advised to update the CUDA toolkit version from 10.0 to 10.2. Nonetheless, it should be noted that such an upgrade could potentially influence various aspects of other project environments, thus I plan to address this matter during future maintenance procedures.

9. Conclusion

In conclusion, this project has demonstrated the potential of hybrid programming in improving performance in data-intensive applications, specifically in the context of stock analysis. By combining Message Passing Interface (MPI) for data collection and CUDA for parallel computing, we were able to surmount the limitations of serial programming, including performance and scalability issues, longer computation times, and inefficient memory usage. My approach resulted in a significant improvement in performance when deploying eight processes simultaneously. Nevertheless, I encountered certain challenges, such as the inability to generate threads when attempting to utilize sixty-four processes concurrently and the outdated version of the CUDA toolkit. These obstacles offer potential avenues for further research and development. Looking ahead, I aspire to address these challenges and continue refining our approach to transform the field of stock analysis by providing faster, more precise, and more accessible insights for investors. Ultimately, hybrid programming holds immense promise to revolutionize stock analysis through greater efficiency, accuracy, and accessibility.

References

Birken, E. (2023, June 12). Top Passive Income Ideas For 2023. *Forbes*. Accessed 2023, September 25. https://www.forbes.com/advisor/investing/passive-income-ideas

Fernando, J. (2023). Moving Average (MA): Purpose, Uses, Formula, and Examples. *Investopedia*. https://www.investopedia.com/terms/m/movingaverage.asp

Fernando, J. (2023). MACD Indicator Explained, with Formula, Examples, and Limitation. *Investopedia*. https://www.investopedia.com/terms/m/macd.asp

Fernando, J. (2023). Relative Strength Index (RSI) Indicator Explained with Formula. *Investopedia*. https://www.investopedia.com/terms/r/rsi.asp

Folger, J. (2023, July 30). Investing vs. Trading: What's the Difference. *Investopedia*. Accessed September 25, 2023. https://www.investopedia.com/ask/answers/12/difference-investing-trading.asp

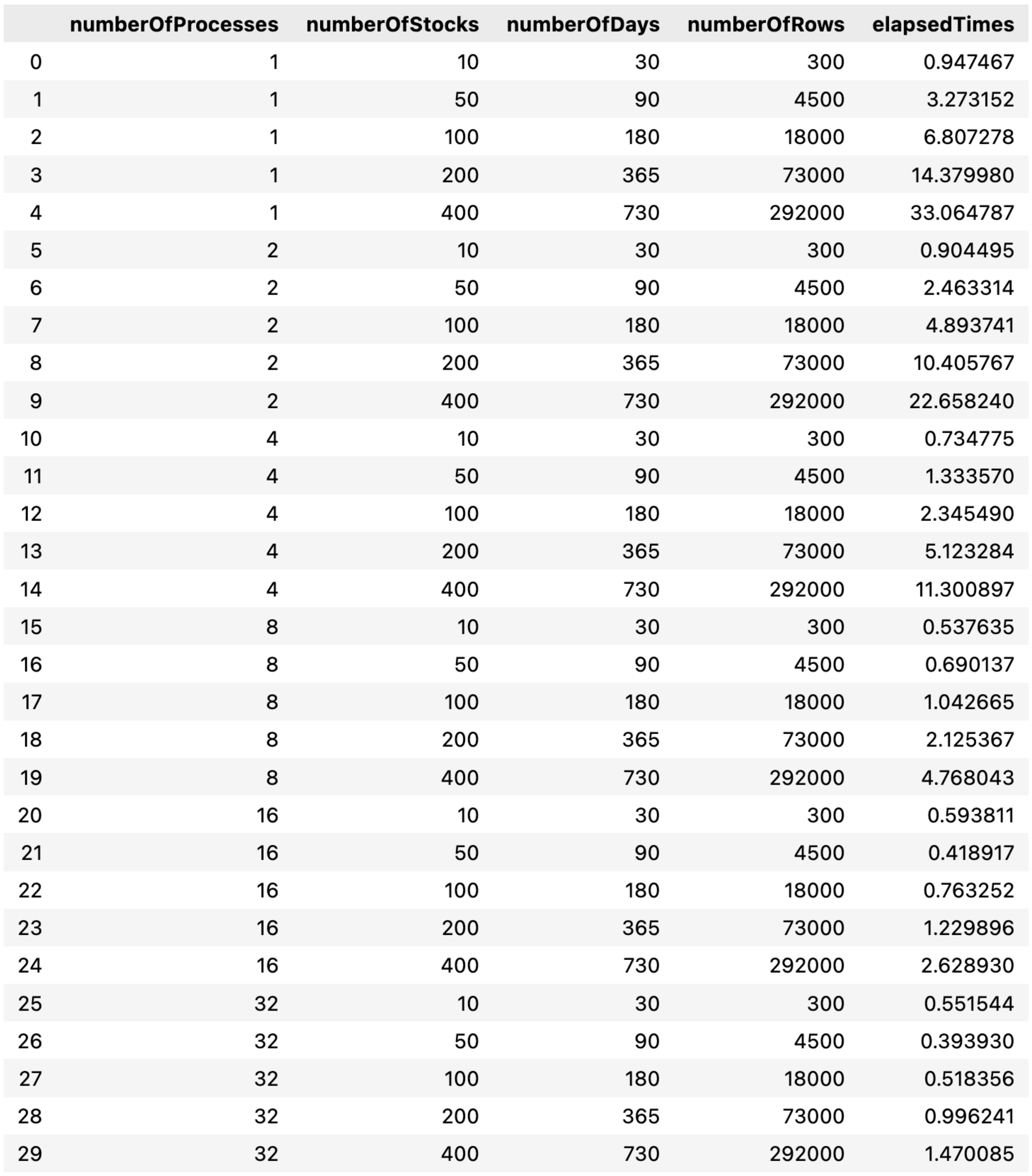
Kijsipongse, E. et al. (2011). Efficient large Pearson correlation matrix computing using hybrid MPI/CUDA. 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE). https://doi.org/10.1109/JCSSE.2011.5930127

King, P. (2011). Protect your Wealth from the Ravages of Inflation. *Apress*. https://books-scholarsportal-info.libproxy.wlu.ca/en/read?id=/ebooks/ebooks2/springer/2012-05-22/1/9781430238232

Appendix 1 – Data Set Schema

VARIABLE	DESCRIPTION	DATA TYPE	FEATURE/LABEL/ STAGING DATA	REMARKS
tick	Stock symbol	text	feature	Nominal data
date	Price date	datetime64	feature	
open	Open price	float64	staging data	
high	High price	float64	staging data	
low	Low price	float64	staging data	
close	Close price	float64	staging data	
adjclose	Adjusted Close price	float64	staging data	
volume	Daily transaction volume	float64	feature	

Appendix 2 – Detail Statistics



	numberOfProcesses	numberOfStocks	numberOfDays	numberOfRows	elapsedTimes
0	1	10	30	300	0.947467
1	1	50	90	4500	3.273152
2	1	100	180	18000	6.807278
3	1	200	365	73000	14.379980
4	1	400	730	292000	33.064787
5	2	10	30	300	0.904495
6	2	50	90	4500	2.463314
7	2	100	180	18000	4.893741
8	2	200	365	73000	10.405767
9	2	400	730	292000	22.658240
10	4	10	30	300	0.734775
11	4	50	90	4500	1.333570
12	4	100	180	18000	2.345490
13	4	200	365	73000	5.123284
14	4	400	730	292000	11.300897
15	8	10	30	300	0.537635
16	8	50	90	4500	0.690137
17	8	100	180	18000	1.042665
18	8	200	365	73000	2.125367
19	8	400	730	292000	4.768043
20	16	10	30	300	0.593811
21	16	50	90	4500	0.418917
22	16	100	180	18000	0.763252
23	16	200	365	73000	1.229896
24	16	400	730	292000	2.628930
25	32	10	30	300	0.551544
26	32	50	90	4500	0.393930
27	32	100	180	18000	0.518356
28	32	200	365	73000	0.996241
29	32	400	730	292000	1.470085

Appendix 3 – Max Threads Issue

```
(cp631-final) [wlai11@mcs1 cp631-final]$ mpirun -np 64 -mca opal_cuda_support 1
~/miniconda3/envs/cp631-final/bin/python ~/cp631-final/cp631_final_v2.py
Traceback (most recent call last):
  File "/home/wlai11/cp631-final/cp631_final_v2.py", line 83, in <module>
Traceback (most recent call last):
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 215, in __init__
    self._repopulate_pool()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 306, in _repopulate_pool
Traceback (most recent call last):
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 215, in __init__
    self._repopulate_pool()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 306, in _repopulate_pool
Traceback (most recent call last):
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 215, in __init__
    self._repopulate_pool()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 306, in _repopulate_pool
Traceback (most recent call last):
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 215, in __init__
    self._repopulate_pool()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 306, in _repopulate_pool
    import kaggle
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/site-
packages/kaggle/__init__.py", line 22, in <module>
    api = KaggleApi(ApiClient())
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/site-
packages/kaggle/api_client.py", line 85, in __init__
    self.pool = ThreadPool()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 930, in __init__
    return self._repopulate_pool_static(self._ctx, self.Process,
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 329, in _repopulate_pool_static
    return self._repopulate_pool_static(self._ctx, self.Process,
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 329, in _repopulate_pool_static
    w.start()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/dummy/__init__.py", line 51, in start
    return self._repopulate_pool_static(self._ctx, self.Process,
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 329, in _repopulate_pool_static
    w.start()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/dummy/__init__.py", line 51, in start
    return self._repopulate_pool_static(self._ctx, self.Process,
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 329, in _repopulate_pool_static
    w.start()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/dummy/__init__.py", line 51, in start
    threading.Thread.start(self)
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/threading.py", line
935, in start
    w.start()
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/dummy/__init__.py", line 51, in start
    threading.Thread.start(self)
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/threading.py", line
935, in start
    threading.Thread.start(self)
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/threading.py", line
935, in start
    threading.Thread.start(self)
  File "/home/wlai11/miniconda3/envs/cp631-final/lib/python3.10/threading.py", line
935, in start
    Pool.__init__(self, processes, initializer, initargs)
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 245, in __init__
    _start_new_thread(self._bootstrap, ())
RuntimeError: can't start new thread
```

Appendix 4 – CUDA Toolkit Outdated Version

```
df_serial:    numberOfRows  elapsedTimes
0           300      0.947467
1          4500      3.273152
2         18000      6.807278
3         73000     14.379980
4        292000     33.064787
    numberOfRows  ...      overhead
0            300  ...     65.171222
1            300  ...    210.205771
2            300  ...    353.955106
3            300  ...    902.776276
4            300  ...   1762.798583
5           4500  ...     11.708379
6           4500  ...     62.970777
7           4500  ...     68.678253
8           4500  ...    104.777233
9           4500  ...    285.126068
10         18000  ...     18.561165
11         18000  ...     37.822484
12         18000  ...     22.535292
13         18000  ...     79.396803
14         18000  ...    143.671642
15         73000  ...     21.229651
16         73000  ...     42.511562
17         73000  ...     18.240306
18         73000  ...     36.845391
19         73000  ...    121.695044
20        292000  ...     20.778152
21        292000  ...     36.712170
22        292000  ...     15.362431
23        292000  ...     27.213532
24        292000  ...     42.274391

[25 rows x 7 columns]
Exception ignored in: <function Pool.__del__ at 0x2b866dbd53f0>
Traceback (most recent call last):
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/pool.py", line 271, in __del__
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/queues.py", line 377, in put
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/connection.py", line 200, in send_bytes
  File "/home/wlai11/miniconda3/envs/cp631-
final/lib/python3.10/multiprocessing/connection.py", line 400, in _send_bytes
TypeError: 'NoneType' object is not callable
```