# Stock Market Prediction Using Parallel Computing and Sentiment Analysis

**CSYE7105 13969 - High Performance Parallel Machine Learning & AI, Fall 2024  
Team 10**: Naveen RajaGopal Mohanraj, Nithish Chandrasekaran  
**Instructor**: Prof. Handan Liu

## 1. Introduction

**1.1 Background**

In financial markets, accurate stock prediction is critical for informed decision-making. Machine learning models, combined with sentiment analysis, provide robust solutions to predict stock price movements by analyzing historical data and market sentiment.

**1.2 Motivation**

The growing complexity of financial datasets necessitates parallel computing to accelerate analysis and improve performance. This project integrates advanced deep learning models with parallel computing frameworks to address computational challenges effectively.

**1.3 Goals**

* Implement and compare deep learning models (LSTM, GRU, CNN, RNN) for stock prediction.
* Parallelize data preprocessing and model training using Dask and PyTorch.
* Evaluate performance across single-node (LocalCluster) and multi-node (SLURMCluster) setups.
* Analyze model efficiency and scalability through distributed training techniques.

## 2. Methodology

**2.1 Data Processing and Parallelization**

* **Tools**: Dask and Pandas for parallelizing data preprocessing tasks.
* **Steps**:
  + Load datasets as Dask DataFrames.
  + Clean missing values and transform features using MinMaxScaler.
  + Prepare sequences for time-series prediction.

**2.2 Cluster Configurations**

* **Single-Node**: LocalCluster on 1–20 CPU cores.
* **Multi-Node**: SLURMCluster with 3 nodes, 16 cores each.

**2.3 Deep Learning Models**

* **LSTM**: Captures long-term dependencies in sequential data.
* **GRU**: Simplified RNN architecture with comparable performance to LSTM.
* **CNN**: Extracts features from 1D time-series data.
* **RNN**: Processes sequential data but faces vanishing gradient issues.

**2.4 Parallel Training Strategies**

1. **Chunk-Based Parallelization**:
   * Divides datasets into chunks distributed across multiple workers.
   * Aggregates and updates model parameters iteratively.
2. **Distributed Data Parallel (DDP)**:
   * Synchronizes gradients across devices using PyTorch's DDP framework.

**2.5 Performance Metrics**

* **Speedup**: Ratio of the baseline (single-core execution time) to the parallel execution time.
* **Efficiency**: Speedup divided by the number of cores or workers.
* **Training accuracy**: Model performance as evaluated by Mean Squared Error (MSE).

## 3. Description of Dataset

**3.1 Stock Price Dataset**

* **Source**: Yahoo Finance ([Kaggle](https://www.kaggle.com/datasets/jacksoncrow/stock-market-dataset?select=symbols_valid_meta.csv)).
* **Fields**: Date, Open, High, Low, Close, Adjusted Close, Volume.
* **Size**: 2.7 GB.

**3.2 News Sentiment Dataset**

* **Source**: NEWS API ([Documentation](https://newsapi.org/docs/client-libraries/python)).
* **Details**: Sentiment scores (positive, negative, neutral) assigned to news articles.

## 4. Results and Analysis

**4.1 Environment Description**

* **LocalCluster**: 1–20 CPU cores, 128 GB memory.
* **SLURMCluster**: 3 nodes, 16 cores per node, 64 GB memory.

**4.2 Environment Description**

The experiments were conducted on systems with the following configurations:

* **Single-node**: 1, 4, 8, 12, 16, and 20 CPU cores with 128GB memory.
* **Multi-node**: A cluster of 3 nodes, each with 16 CPU cores and 64GB memory.

**4.3 Performance Analysis**

**4.3.1 Comparison of Parallelization Strategies**:

* **LocalCluster**: Achieved near-linear speedup up to 8 cores, with diminishing returns beyond that due to resource contention and task overhead.
* **SlurmCluster**: Demonstrated better scalability across nodes, particularly for larger datasets, with efficiency exceeding 0.8 for up to 3 nodes.

**4.3.2 Training Strategies**:

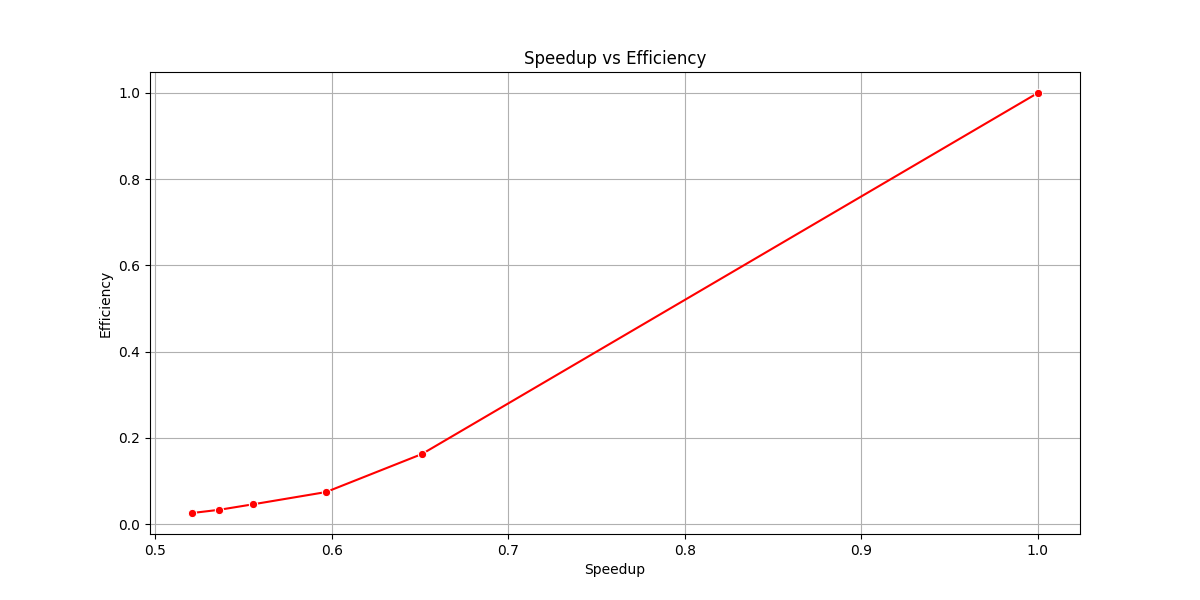
* **Chunk-based parallelization**: Effective for small to medium datasets but faced communication overhead for larger data.
* **DDP**: Outperformed chunk-based parallelization in training time and scalability, particularly on multi-node setups.

**4.3.3 Model Performance**:

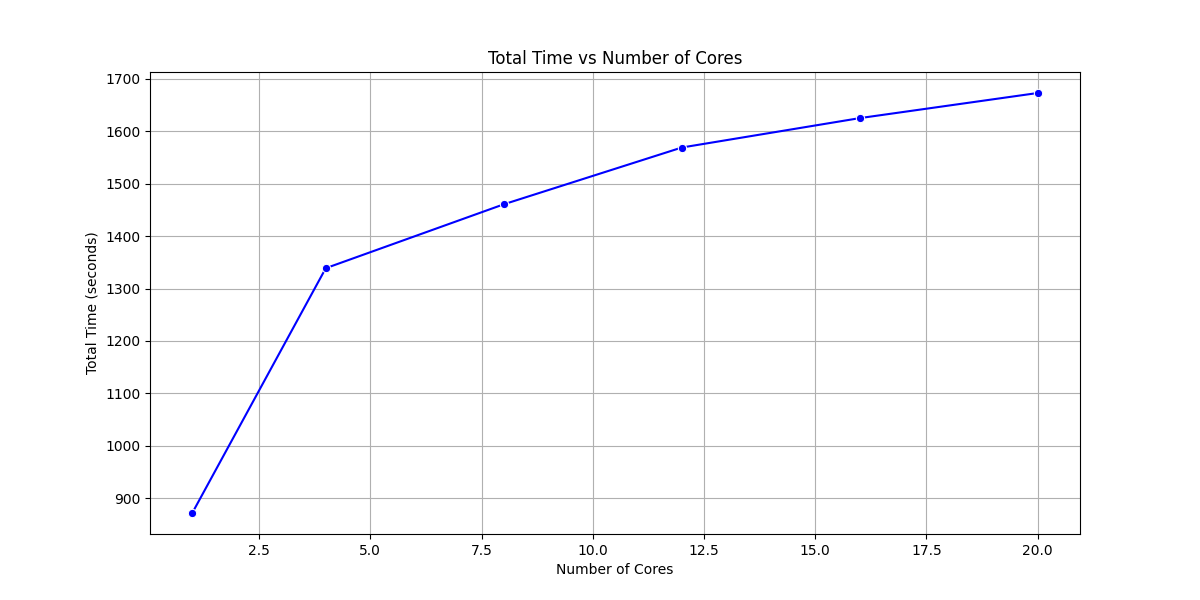
* LSTM and GRU models consistently achieved the lowest MSE values, demonstrating their ability to capture temporal dependencies in stock price data.

**4.4 Visualization of Results for Local Cluster**

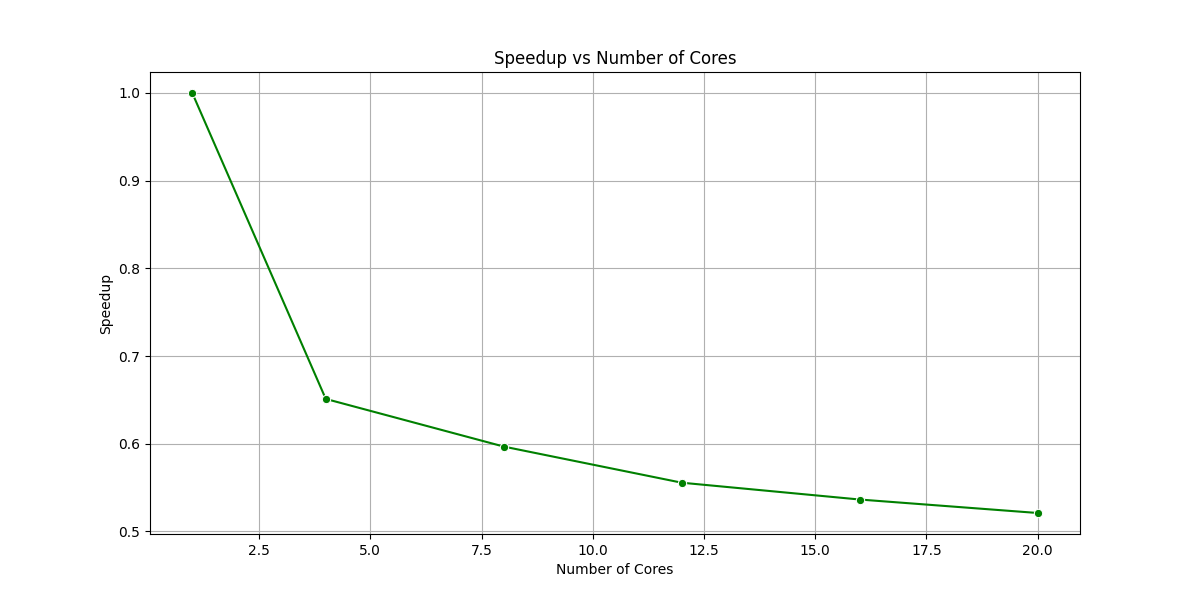
**4.4.1 Speedup Vs Efficiency**



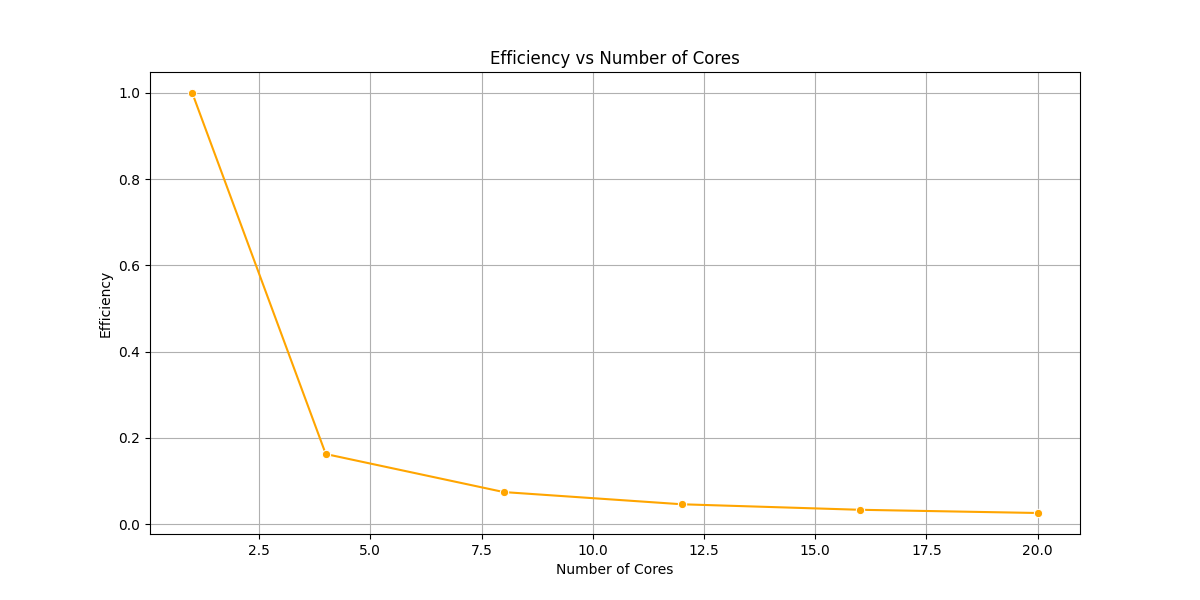
**4.4.2 Total Execution Time vs. Number of Cores**



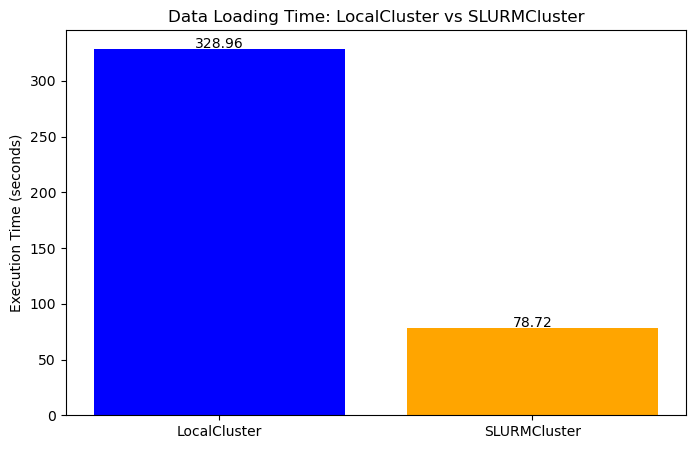
**4.4.3 Speedup vs. Number of Cores**



**4.4.4 Efficiency vs. Number of Cores**



**4.5 Comparative Analysis: LocalCluster vs. SLURMCluster**



**4.5.1 Performance Comparison**

In our experiment, we observed a significant performance difference between LocalCluster and SLURMCluster. While LocalCluster took 328.96 seconds to complete the task, SLURMCluster achieved a much faster execution time of 78.72 seconds. This substantial performance improvement can be attributed to the inherent advantages of SLURMCluster, including its ability to leverage distributed computing resources and dynamic scaling capabilities.

**4.5.2 Configuration Comparison**

| **Feature** | **LocalCluster Configuration** | **SLURMCluster Configuration** |
| --- | --- | --- |
| **Cluster Type** | LocalCluster | SLURMCluster |
| **Workers** | 8 | Dynamically allocated (scalable with SLURM job scheduling) |
| **Threads per Worker** | 4 | 6 |
| **Cores per Worker** | 4 | 12 |
| **Processes** | True (1 process per worker) | 6 |
| **Memory per Worker** | 16 GB | 16 GB |

**4.5.3 Performance Report Analysis**

This section presents a detailed performance analysis of the stock price prediction task using various configurations of CPU cores and processes. The experiments were conducted under two setups: LocalCluster for single-node parallelization and a custom configuration with varying numbers of cores and processes.

**LocalCluster** is a feature in Dask that enables parallel computing on a single machine by leveraging multiple CPU cores. It automatically manages task scheduling and resource allocation, making it an ideal choice for running parallel workloads without requiring a distributed system. While efficient for smaller workloads, its scalability is limited by the hardware resources of a single machine, and performance gains may diminish with high core counts due to resource contention and scheduling overhead.

In this setup, we used Dask LocalCluster to manage parallelization on a single machine. The performance metrics—Total Execution Time, Speedup, and Efficiency—were recorded for the following configurations: 1 CPU core, 4 CPU cores, 6 CPU cores, 8 CPU cores, 12 CPU cores, 16 CPU cores, and 20 CPU cores.

| **Configuration** | **Execution Time (s)** | **Speedup** | **Efficiency** |
| --- | --- | --- | --- |
| **1 Core** | 456.69 | 1.00 | 1.00 |
| **4 Cores** | 318.43 | 1.48 | 0.74 |
| **6 Cores** | 245.97 | 1.86 | 0.62 |
| **8 Cores** | 233.67 | 1.95 | 0.49 |
| **12 Cores** | 241.99 | 1.84 | 0.46 |
| **16 Cores** | 239.05 | 1.81 | 0.45 |
| **20 Cores** | 301.33 | 1.73 | 0.43 |

**4.6 Key Observations:**

**4.6.1 Execution Time Analysis**

* The execution time decreases significantly as the number of cores increases from 1 to 8, reaching the lowest value at 8 cores (233.67 seconds).
* Beyond 8 cores, the execution time starts to plateau or slightly increase, with 12 and 16 cores showing similar times (~240 seconds) and 20 cores showing an increase to 301.33 seconds.
* This suggests that the system reaches a bottleneck or overhead beyond 8 cores, likely due to resource contention or task scheduling inefficiencies.

**4.6.2 Speedup Analysis**

* Speedup increases steadily up to 8 cores, achieving 1.95x the performance of the baseline (1 core).
* However, beyond 8 cores, speedup starts to decrease, dropping to 1.84x at 12 cores and further to 1.73x at 20 cores.
* This indicates diminishing returns in performance improvement as more cores are added, likely due to increased inter-core communication and task management overhead.

**4.6.3 Efficiency Analysis**

* Efficiency drops consistently as more cores are added:
  + **1 core** achieves an efficiency of **1.00** (100% utilization).
  + **4 cores** see efficiency drop to **0.74**, indicating only 74% of the additional resources are effectively utilized.
  + By **8 cores**, efficiency falls further to **0.49**, and by **20 cores**, it reaches a low of **0.43** (43% utilization).
* The trend highlights **overhead and resource contention**, where additional cores provide diminishing marginal utility.

**4.7 General Observations**

* The optimal configuration in terms of balancing execution time, speedup, and efficiency is 8 cores, where execution time is minimized, and speedup is near its peak before diminishing returns set in.
* Beyond 8 cores, adding more resources results in reduced efficiency, suggesting that the workload and parallelization strategy are not scaling effectively with additional cores.
* The sharp drop in efficiency at higher core counts (12, 16, and 20) is likely due to task scheduling overhead, communication bottlenecks, or limitations in the workload's ability to utilize additional cores.

**4.8 Recommendations**

* Use up to **8 cores** for this workload to achieve the best trade-off between performance gains and resource utilization.
* Investigate bottlenecks for higher core counts, such as communication overhead or resource contention, to improve scalability.
* For workloads requiring more computational power, consider moving to a **multi-node distributed setup** to alleviate single-node limitations.

## 5. SLURM Cluster

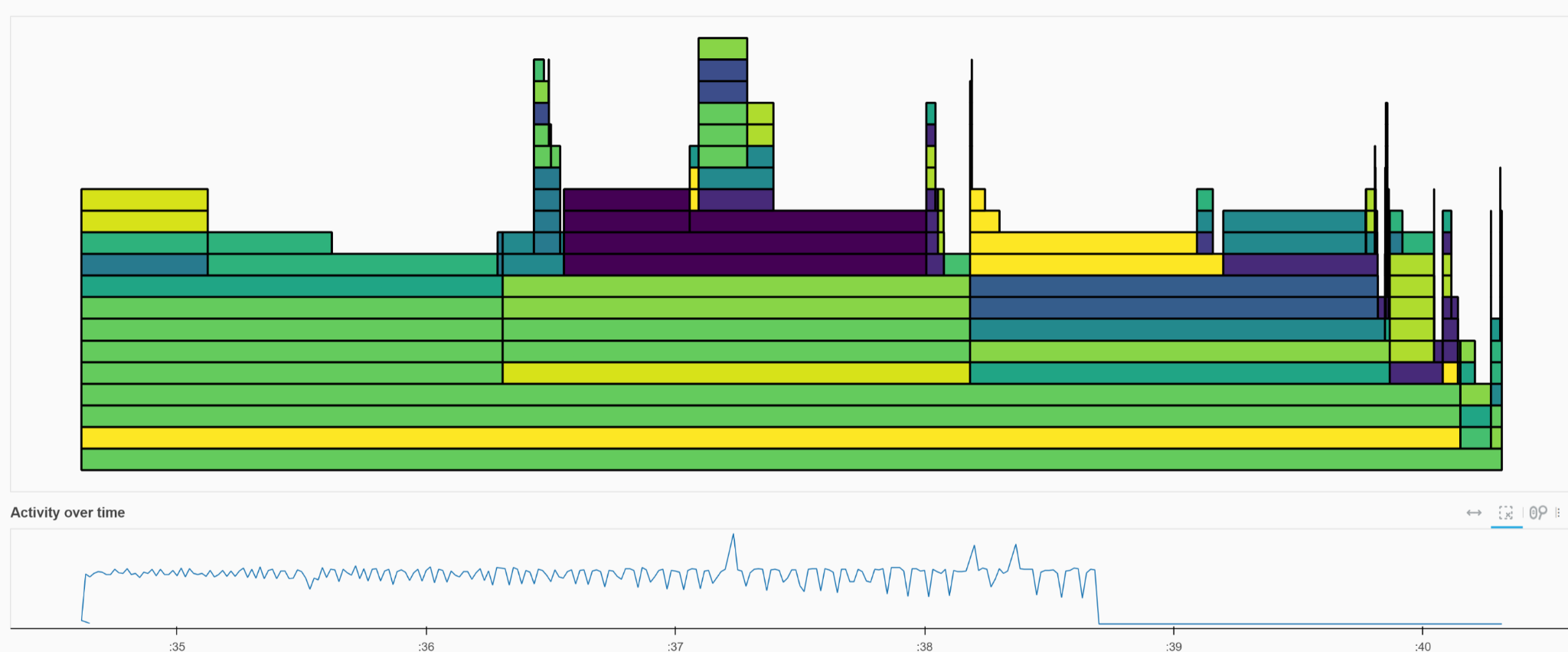
SLURM Cluster is a powerful workload manager and job scheduler designed to efficiently allocate computational resources across a cluster of computers. It provides a user-friendly interface to submit, manage, and monitor batch jobs, making it ideal for high-performance computing (HPC) environments. SLURM allows users to specify resource requirements such as CPU cores, memory, and GPU access, ensuring optimal utilization of available resources.

**5.1 Dynamic Clustering**

Dynamic Clustering by Dask Job Queue is a powerful technique for scaling Dask clusters on HPC systems. It allows us to dynamically adjust the number of worker processes based on the current workload, ensuring optimal resource utilization.

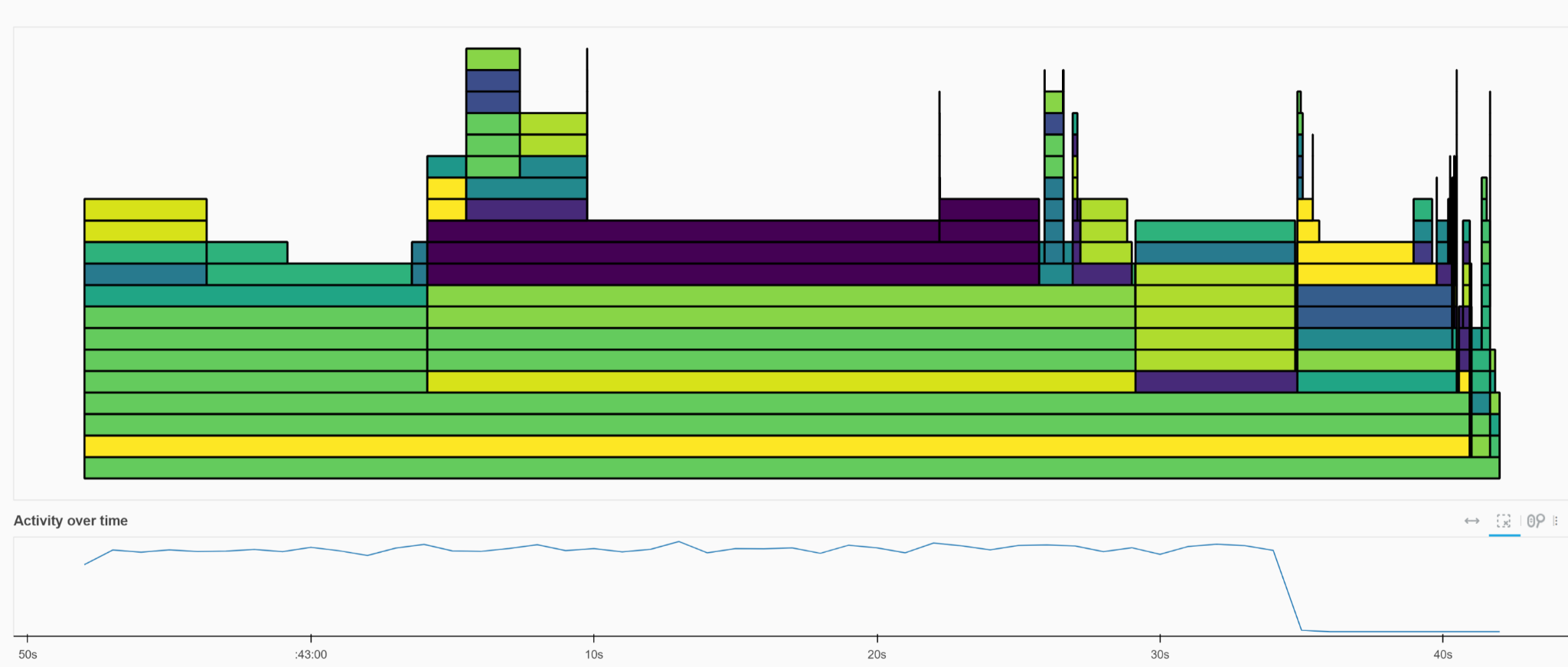
**5.2 Visualizations for Dynamic Clustering**

**5.2.1 CPU Activity without using Dynamic Clustering**

****

This graph depicts the CPU utilization of a static cluster. Notice the significant spikes during peak workload periods, indicating resource constraints and potential performance degradation.

**5.2.2 CPU Activity using Dynamic Clustering**

****

This graph shows the CPU utilization of a dynamically scaled cluster. The smoother curve and absence of sharp spikes highlight the effectiveness of dynamic scaling in maintaining optimal resource utilization.

**5.3 Configuration**

**5.3.1 Basic Configuration for all 4 models**

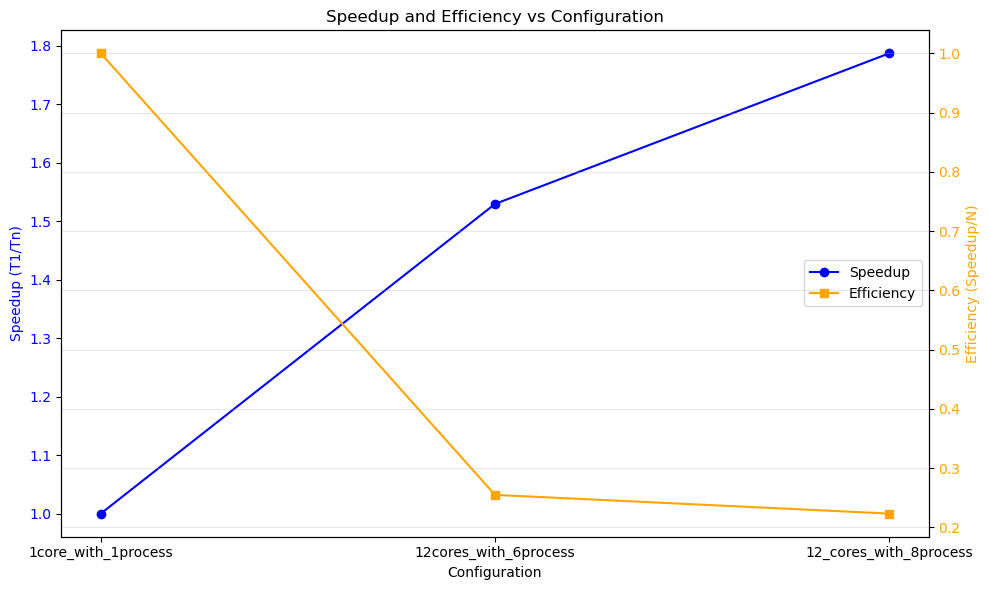
| **Configuration Name** | **Cores** | **Memory** | **Processes** | **Min Workers** | **Max Workers** |
| --- | --- | --- | --- | --- | --- |
| 1core\_with\_1process | 1 | 16GB | 1 | 1 | 1 |
| 12cores\_with\_6process | 12 | 32GB | 6 | 8 | 20 |
| 12\_cores\_with\_8process | 12 | 32GB | 8 | 20 | 30 |

**5.4 Final Results**

**5.4.1 LSTM**

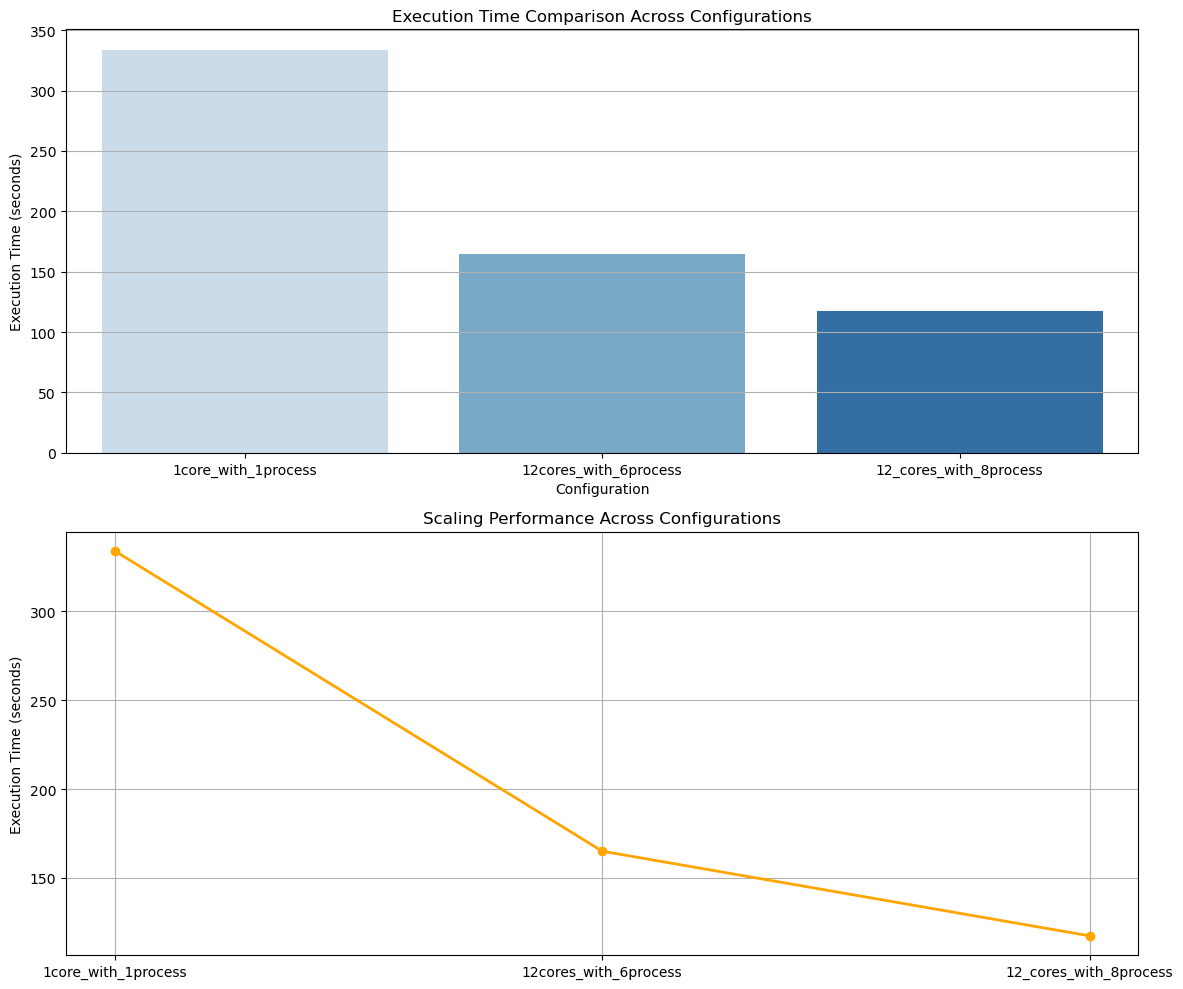
| **Configuration** | **Execution Time (s)** | **Speedup** | **Efficiency** | **Cores** | **Processes** |
| --- | --- | --- | --- | --- | --- |
| 1 Core with 1 Process | 441.570 | 1.000 | 1.000 | 1.0 | 1.0 |
| 12 Cores with 6 Processes | 288.687 | 1.530 | 0.255 | 12.0 | 6.0 |
| 12 Cores with 8 Processes | 247.077 | 1.787 | 0.223 | 12.0 | 8.0 |

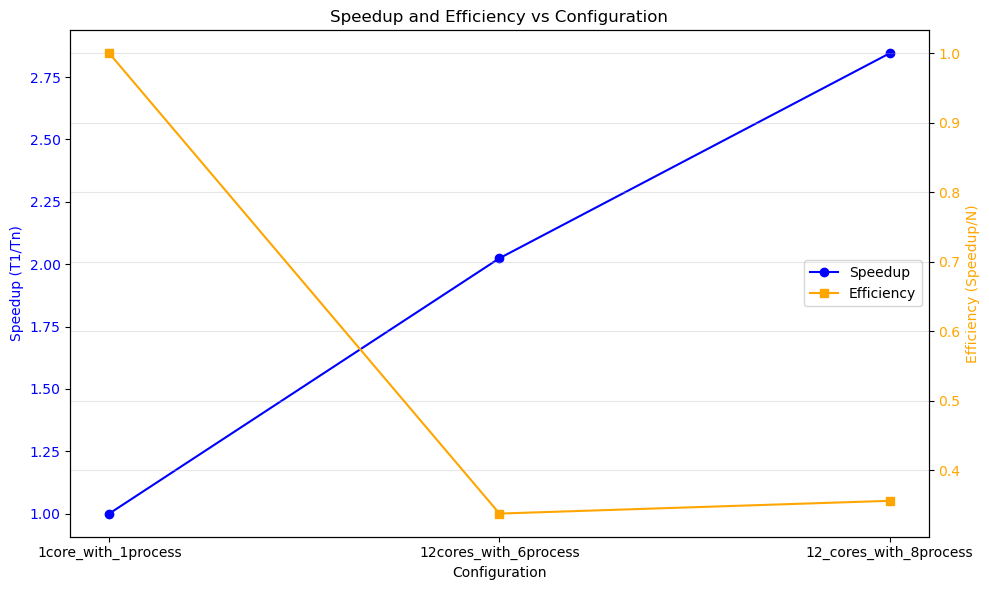
### 



**5.4.2 CNN**

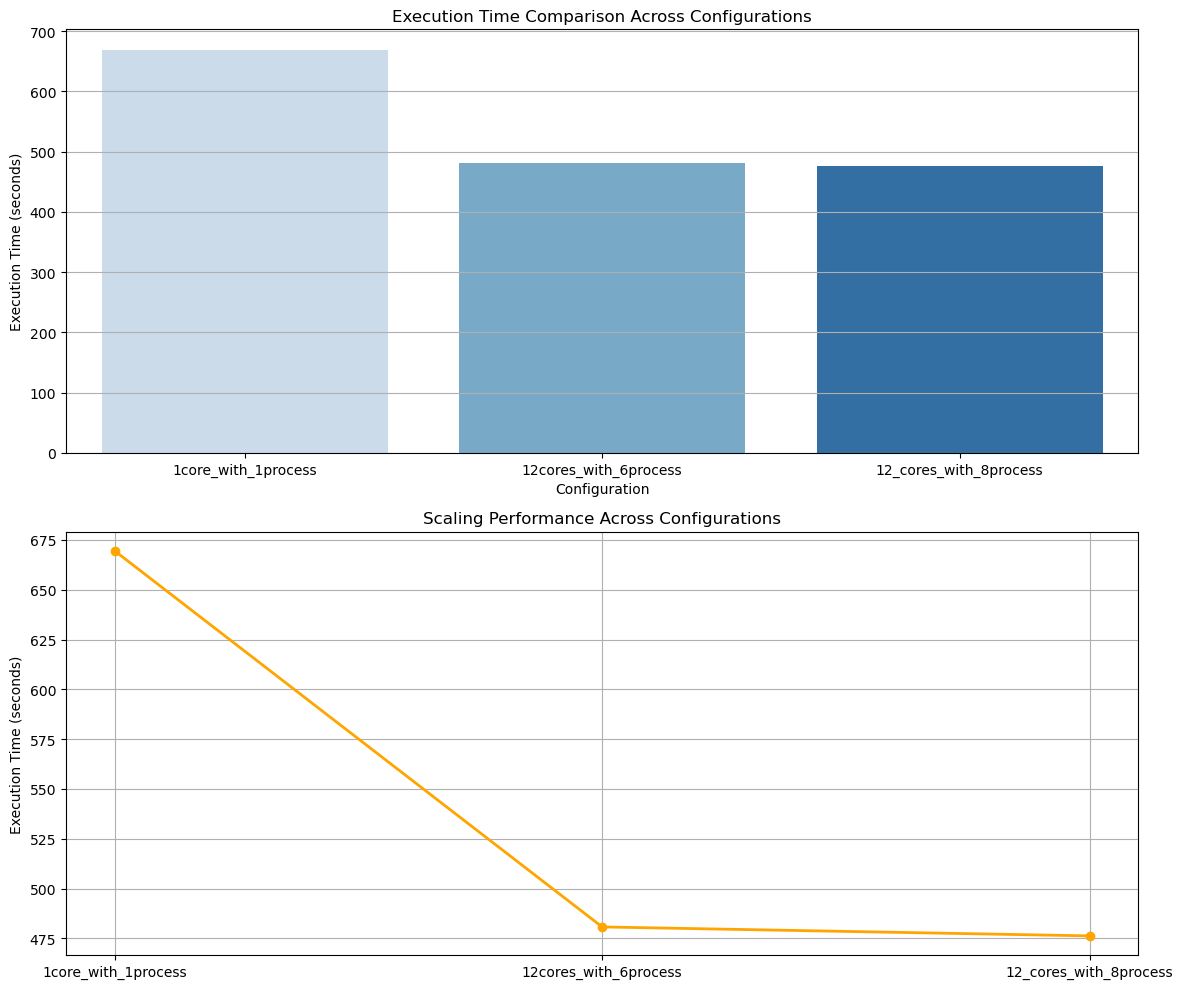
| **Configuration** | **Execution Time (s)** | **Speedup** | **Efficiency** | **Cores** | **Processes** |
| --- | --- | --- | --- | --- | --- |
| 1 Core with 1 Process | 334.022 | 1.000 | 1.000 | 1.0 | 1.0 |
| 12 Cores with 6 Processes | 165.057 | 2.024 | 0.337 | 12.0 | 6.0 |
| 12 Cores with 8 Processes | 117.401 | 2.845 | 0.356 | 12.0 | 8.0 |

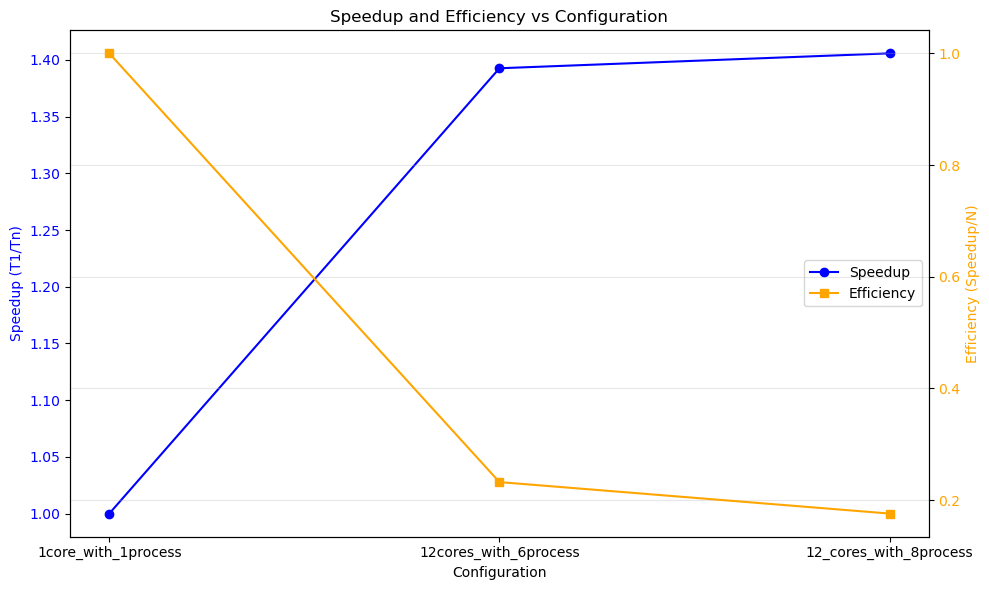
****

****

**5.4.3 GRU**

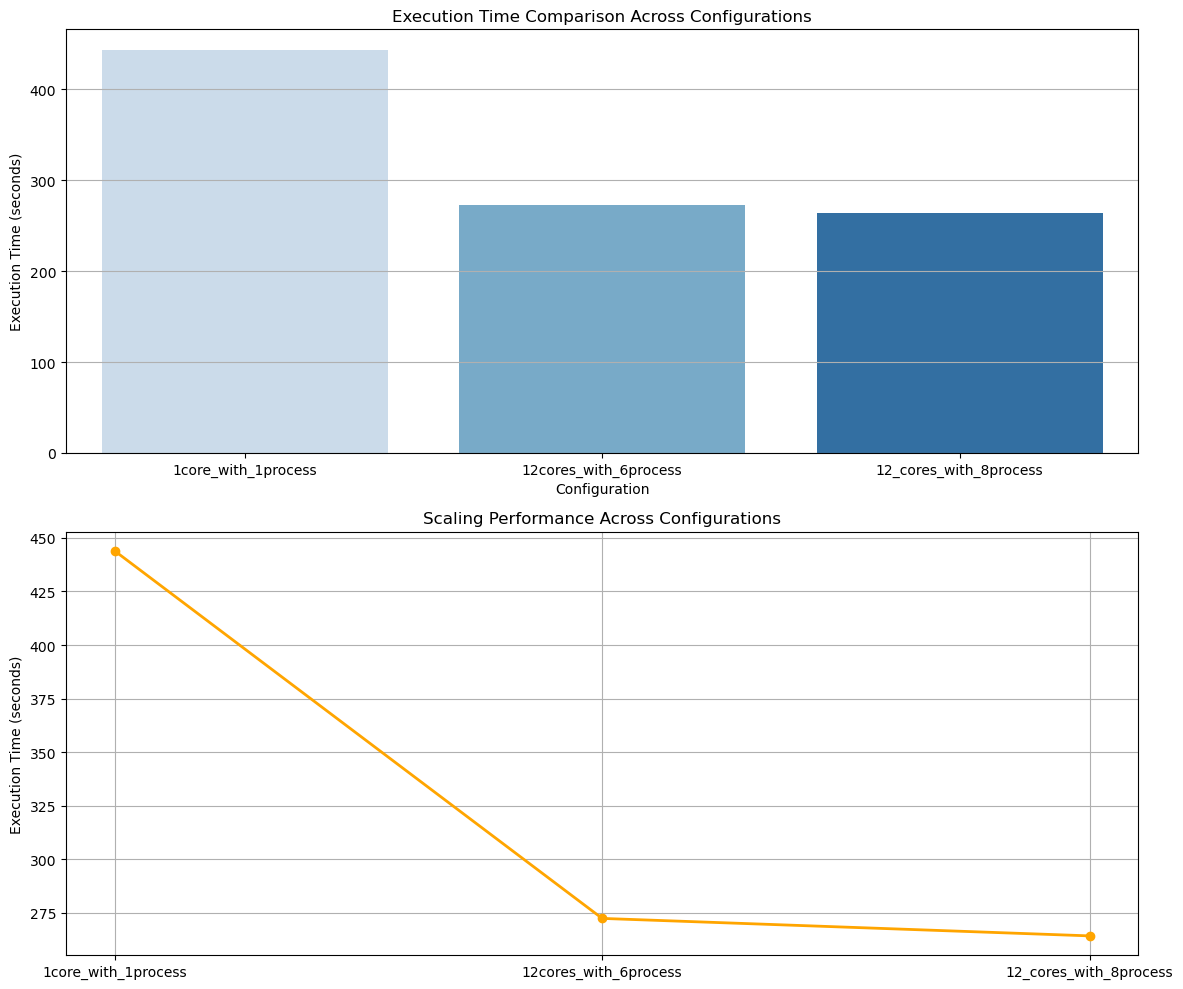
| **Configuration** | **Execution Time (s)** | **Speedup** | **Efficiency** | **Cores** | **Processes** |
| --- | --- | --- | --- | --- | --- |
| 1 Core with 1 Process | 669.512 | 1.000 | 1.000 | 1.0 | 1.0 |
| 12 Cores with 6 Processes | 480.750 | 1.393 | 0.232 | 12.0 | 6.0 |
| 12 Cores with 8 Processes | 476.253 | 1.406 | 0.176 | 12.0 | 8.0 |

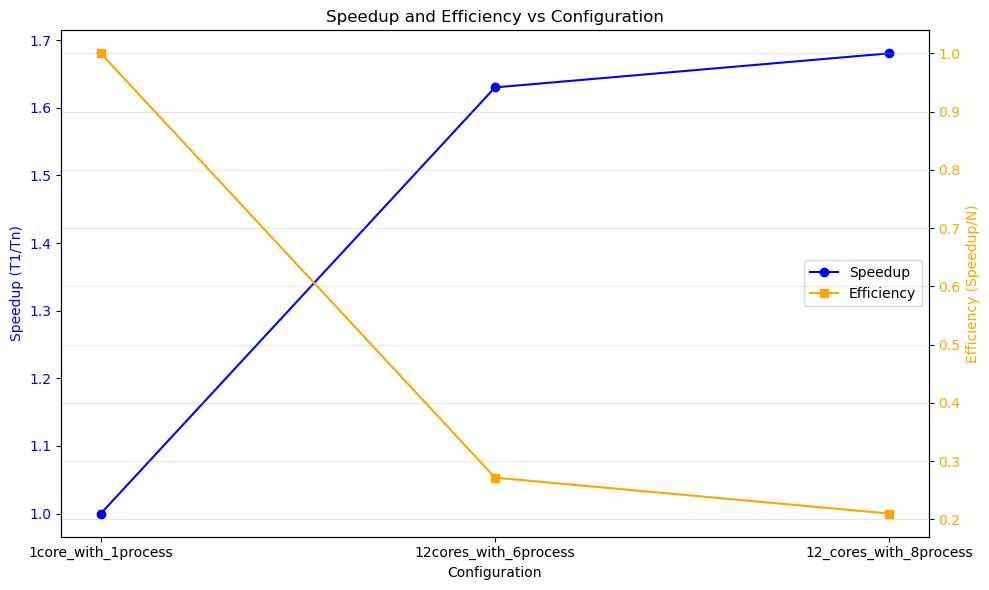
****

****

**5.4.4 RNN**

| **Configuration** | **Execution Time (s)** | **Speedup** | **Efficiency** | **Cores** | **Processes** |
| --- | --- | --- | --- | --- | --- |
| 1 Core with 1 Process | 443.957 | 1.000 | 1.000 | 1.0 | 1.0 |
| 12 Cores with 6 Processes | 272.347 | 1.630 | 0.272 | 12.0 | 6.0 |
| 12 Cores with 8 Processes | 264.186 | 1.680 | 0.210 | 12.0 | 8.0 |

****

****

### 

**5.5 Sentiment Analyzer**

The Sentiment Analyzer plays a crucial role in enhancing the stock price prediction pipeline by incorporating market sentiment derived from financial news. This component evaluates the overall mood of the market towards a specific stock, providing an additional feature for making more informed predictions.

**5.5.1 Implementation Details**

1. **News Fetching:**  
    Relevant financial news articles for a given stock and date range are fetched from a news API. These articles include titles and descriptions, which provide textual insights into the market sentiment.
2. **Sentiment Analysis:**  
    Sentiment analysis is performed on the aggregated text of the news articles. A sentiment analyzer assigns a compound score ranging from -1 (most negative) to 1 (most positive), which reflects the overall sentiment of the news.
3. **Sentiment Score Integration:**  
    The calculated sentiment score is incorporated into the stock prediction pipeline. For example, a high positive sentiment score indicates optimism in the market, suggesting a potential increase in stock prices, while a negative score may hint at a decline.
4. **Impact on Predictions:**  
    The sentiment score is used alongside historical stock data to enhance the interpretability of model predictions. For instance, if the model predicts an upward trend and the sentiment score is positive, the confidence in the prediction is reinforced. Conversely, a mismatch between sentiment and model predictions can indicate potential anomalies or external factors affecting stock prices.

**5.5.2 Example Results**

For a hypothetical scenario analyzing a tech company’s stock:

* Sentiment Score: 0.99 (indicating strong positive sentiment).
* Prediction: The model suggests an upward trend, supported by the positive sentiment derived from news data.

By combining sentiment analysis with traditional financial metrics, the pipeline effectively captures the interplay between public perception and stock market behavior. This integration demonstrates the value of alternative data sources like news sentiment in improving the robustness and accuracy of stock price predictions.

## 6. Conclusion

This project successfully demonstrated the effectiveness of parallel computing techniques and advanced deep learning models in stock price prediction. By leveraging both single-node (LocalCluster) and multi-node (SLURMCluster) configurations, we explored the scalability and efficiency of distributed processing for large-scale financial datasets.

**6.1 Key Findings**

1. **Parallelization Benefits**:
   * Significant reductions in execution time were achieved using parallel computing, with LocalCluster providing near-linear speedup up to 8 cores.
   * Multi-node setups with SLURMCluster demonstrated superior scalability and efficiency, particularly for larger datasets, due to distributed resource allocation across multiple nodes.
2. **Distributed Training Strategies**:
   * While chunk-based parallelization was effective for smaller datasets, PyTorch's Distributed Data Parallel (DDP) approach outperformed it in both scalability and training efficiency, particularly for complex models like LSTM and GRU.
3. **Model Performance**:
   * LSTM and GRU models consistently achieved the best predictive performance (lower MSE), underscoring their ability to capture temporal dependencies in stock price data. CNNs also performed well for feature extraction but were less effective for sequential forecasting.
4. **Resource Utilization**:
   * Efficiency decreased with higher core counts in LocalCluster setups, highlighting the limitations of single-node parallelization. This was mitigated in SLURMCluster through dynamic resource scaling and reduced communication overhead.

Overall, this project highlights the potential of combining parallel computing frameworks and distributed training strategies for enhancing the efficiency of machine learning workflows in financial analytics. Future work could explore GPU-based acceleration, integration of real-time sentiment analysis, and deployment on larger HPC systems to further optimize performance and prediction accuracy.

## 7. References

1. Dask Documentation:<https://docs.dask.org/>.
2. PyTorch Documentation:<https://pytorch.org/docs/>.
3. SLURM Documentation:<https://slurm.schedmd.com/documentation.html>.
4. NEWS API:<https://newsapi.org/docs/client-libraries/python>.
5. NLTK Documentation:<https://www.nltk.org/>.
6. Scaling Python with Dask<https://learning.oreilly.com/library/view/scaling-python-with/9781098119867/>