*A Project Report on*

# **Asteroid Data Classification and Prediction Through Parallel Computing**

*Under the guidance of*

**Dr. Handan Liu**

**Team 21**

Akshit Hasmukh Kumar Jain, 002876444

Uday Kiran Dasari, 002819106

***CSYE 7105 – High Performance Parallel Machine Learning & AI, Spring 2024***

## **INTRODUCTION**

### **Background**

The Asteroid Dataset Classification and Prediction project aims to analyze a comprehensive dataset containing various attributes of known asteroids. By utilizing advanced machine learning algorithms, classification of different types of asteroids based on their properties and predict their potential trajectories and behavior is extremely helpful. Through the application of data-driven models, we can enhance our understanding of the unique characteristics of these space objects and their likelihood of impacting Earth. The insights derived from such research can assist space agencies, researchers, and policymakers in devising effective strategies for planetary defense and risk mitigation.

### **Motivation**

Asteroids, being remnants of the early solar system, carry vital information about its formation and evolution. Moreover, their potential threat to Earth necessitates a comprehensive understanding of their characteristics and behavior. By leveraging advanced data analysis techniques, we can enhance our ability to predict and classify the trajectory and properties of asteroids, thus enabling better preparedness and mitigation strategies. This project aims to contribute to the ongoing efforts to safeguard our planet from potential asteroid impacts and to deepen our knowledge of the cosmos.

### **Goal**

The primary goal of this project is to develop a model for asteroids classification using parallel computing methods for pre-processing and model training for classifying asteroids based on their properties. By employing Dask and other parallel methods, we aim to improve the efficiency of the model enabling faster classification of asteroids. The objective is to compare the speed-up performance achieved through parallel processing on various hardware configurations, aiming to enhance the predictive accuracy and efficiency of asteroid classification models. The project involves data preprocessing, exploratory data analysis, feature engineering, model development, and performance evaluation[1]. By harnessing the power of machine learning, this project endeavors to contribute to the broader field of planetary science and space exploration, reinforcing our ability to protect our planet from potential celestial threats. The successful execution of this project can pave the way for the development of more robust and accurate predictive models, thereby bolstering our capabilities in understanding and mitigating the potential risks associated with asteroids. Additionally, the project's findings may provide valuable insights for future space missions, enabling better planning and decision-making for exploration and resource utilization beyond Earth's orbit.

## **METHODOLOGY**

### **2.1 Data preprocessing and cleaning:**

* Loading the dataset using Dask Dataframe and Pandas[2].
* Extract relevant data focusing on the 45 quantitative parameters for each asteroid.
* Utilize parallel processing with dask dataframe to distribute these tasks across multiple cores or nodes for tasks like data extraction, cleaning, and normalization.

### **2.2 EDA analysis:**

* Identified null values and handled them using Dask SimpleImputer using medium strategy.
* Handled inconsistent data types by converting them into Dask Dataframes or Dask Arrays.
* We employed “ExtraTreesClassifier” using multiprocessing Parallel method for feature extraction.
* We set the threshold as mean of feature importances of columns from ETC.

### **2.3 Model Building:**

* Evaluate ML models on the asteroid dataset. Utilizing different machine learning approaches such as Random Forests, KNN, XGBoost and Feed Forward Neural Network models for performing classification and prediction.
* Employ Dask-ML and Multiprocessing techniques to parallelize the model training and evaluation to expedite the process. Utilized PyTorch framework that supports parallel computation on CPUs/GPUs for faster model training of neural networks.
* Leverage Data Parallelism and Distributed Data Parallelism by distributing the workloads on multiple devices such as multiple GPUs.

### **2.4 Scope for Parallelization:**

* We had 958524 rows, and 45 columns comprises of 435 MB which is computationally intensive for serial processing. We used parallelization for Data preprocessing, features selection and Hyperparameter tuning to get the best parameters in Machine learning models, and we ran the neural networks using two methods data parallel and distributed data parallel.
* Implemented parallelization using Dask and multiprocessing libraries to distribute tasks across multiple cores or nodes efficiently while training the various ML models.

### **2.5 Performance Evaluation:**

* Measure the execution time of critical tasks in both serial and parallel implementations.
* Compare the speed-up performance achieved and efficiency by different parallel methods on different hardware configurations by using multiple CPUs.

### **2.6 Analysis and Visualization:**

* Analyze the time taken for model training on different CPU configurations.
* Visualize the performance gains using graphs or charts to present the efficiency improvements clearly.

### **2.7 Optimization and Iteration:**

* Fine-tune the parallel implementation by optimizing task distribution and resource allocation.
* Iterate on the methodology based on performance insights to further enhance efficiency.

### **2.8 Description of the Dataset**

This dataset comprises of 958524 records, which are observations related to asteroids. Each record contains a unique asteroid with its associated data such as orbital parameters, physical properties etc. It features a collection of 45 quantitative parameters for each asteroid. These parameters include eccentricity e, semi-major axis a, perihelion distance q, inclination i, absolute magnitude H, diameter among many other characteristics. The dataset’s diversity in variables allows a detailed and nuanced modelling approach. The dataset is available in the Jet Propulsion Laboratory of NASA[3].

This dataset sourced from Kaggle, serves an excellent resource in honing our skills in using ML models and analysing the performance of these models when considering parallelism, offering a real-world context for Parallel Processing. It is a classification problem since we need to categorize asteroids based on the features.

|  |  |  |
| --- | --- | --- |
| **Dataset Size** | 435 MB | |
| **Contains** | 958524 rows × 45 columns | |
| **Data Types** | object (8), float64(34), int64(3) | |
| **Target** | “***class***” (0...13) namely [MBA, OMB, MCA, AMO, IMB, TJN, CEN, APO, ATE, AST, TNO, IEO, HYA] | |
| **Features** | n (Mean Motion) | angular speed at unit degree per day to complete 1 orbit |
| Q (Perihelion Distance) | closest distance between the orbiting body (Here eg. Asteroids) and the sun |
| a (Semi-major axis) | Semi-major axis of an elliptical orbit is half of the major axis |
| moid\_ld | Earth Minimum Orbit Intersection Distance au Unit |
| e (Eccentricity) | orbital parameter that describes the structure of orbit |
| i (inclination) | angle with respect to x-y ecliptic plane |
| H (Absolute Magnitude) | measured from where observation was placed at unit heliocentric and geocentric distance at zero phase angle |
| ma (Mean Anamoly) | product of mean motion of orbiting body and past perihelion passage |
| tp | time of perihelion passage TDB Unit |
| diameter | diameter of an Asteroid at unit Kilometer |
| class | describes the class of an asteroid |

**Data Sources:** <https://www.kaggle.com/datasets/sakhawat18/asteroid-dataset/data>

## **Results and Analysis:**

**CLUSTER**

Cluster: Discovery High-Performance Computing Cluster

Reservation: csye7105-gpu

Reservation memory: 128 GB

**GPU:**

Model: Nvidia V100-SXM2

GPU count: 1,2

GPU Memory: 64 GB

**CPU:**

CPU Configuration: x86\_64

CPUs count: 28

**Code Files Description**

There are 11 code files, which are arranged according to the order of execution as follows:

1. SequentialExecution.ipynb
2. RFDask.ipynb
3. RFMP.ipynb
4. KNNDask.ipynb
5. KNNMP.ipynb
6. XGBoostParallel.ipynb
7. NNDataParallel.ipynb
8. DDPNeuralNetworkon1GPU.ipynb
9. DDPNeuralNetworkon2GPU.ipynb
10. DDP Neural Network on CPU – Can be executed with ***python filename.py***
11. DDP Neural Network on GPU – Can be executed with ***python filename.py***

### **3.1 Data preprocessing and cleaning**

In the initial stages of our analysis, we focused on ensuring the quality and suitability of the astronomical dataset for further exploration. This crucial step, known as data preprocessing and cleaning, involved several key processes:

**3.1.1 Data Loading and Type Checking**: We employed Dask Dataframe[4], a powerful library for parallel computing, to efficiently load the large dataset (“./dataset.csv”). We load the data into 9 batches of 50 MB on multiple CPUs. We have used “***Block size”*** as parameter with value as “50e6” that defines loading 50MB of data on multiple CPUs. Dask outperforms traditional libraries like Pandas when dealing with loading extensive data volumes into the dataframe. Additionally, we verified the data types of each column to ensure consistency and proper interpretation during analysis.

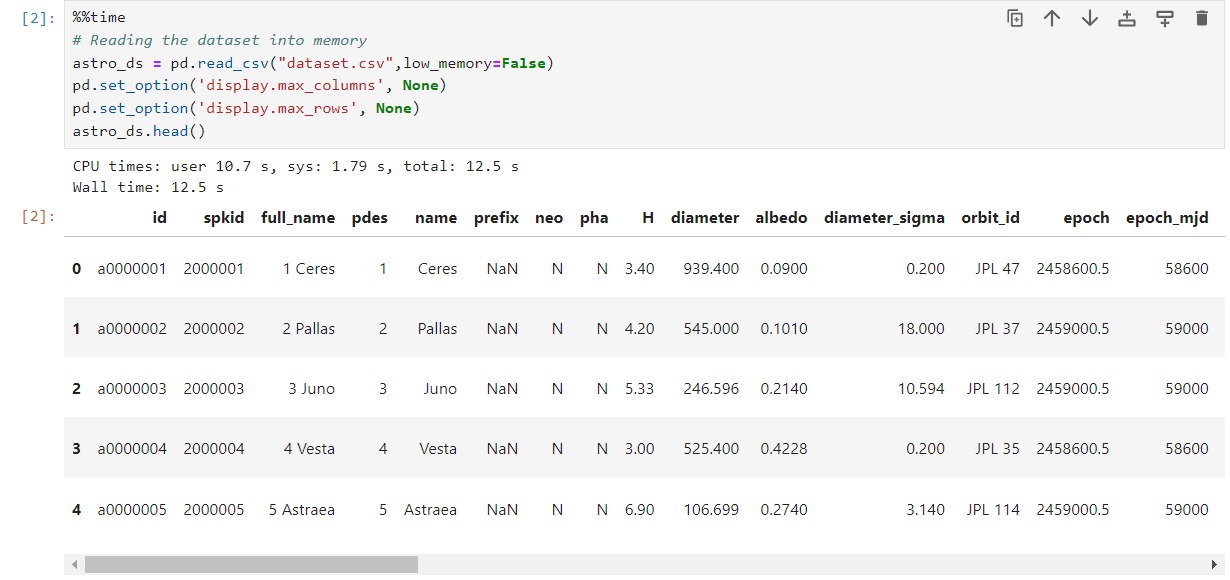


Figure 1: Data loading using Pandas (Serial Execution)

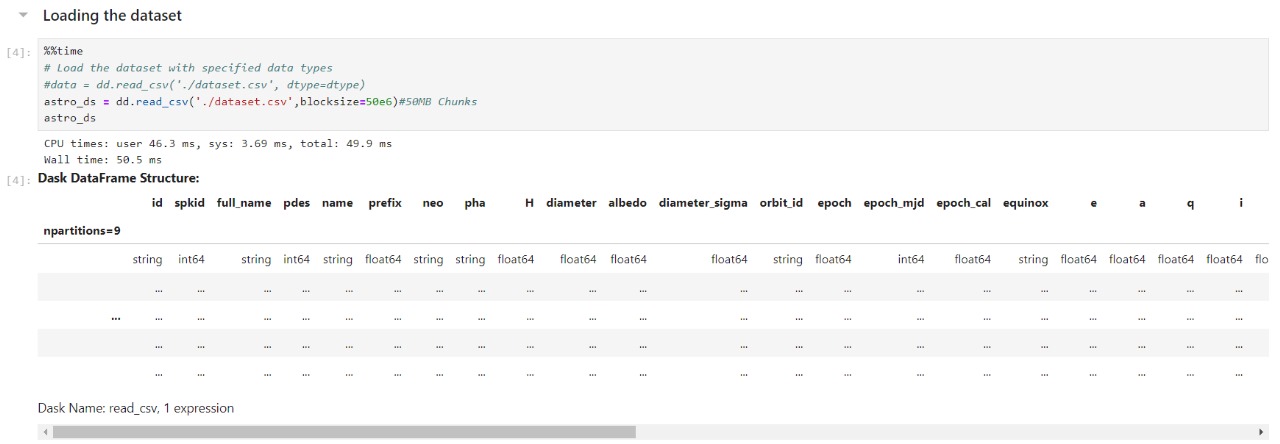


Figure 2: Data loading using Dask Dataframe (Parallel Execution)

The results as shown in figure 1 and figure 2, when using Pandas, it takes 12.5 seconds and when we use parallel method Dask Dataframe to load the dataset it takes 50.5 ms, improving computation time.

**3.1.2 Dropping Less Informative Features**: We identified and removed specific columns (e.g., “id,” “full\_name,” “equinox”) that were deemed less relevant to our analysis goals. These columns might not contribute significantly to understanding the classification of celestial objects within the dataset.

A screenshot of a computer

Description automatically generated

Figure 3: Showing first 5 columns after removing less informative columns

**3.1.3 Data Exploration and Assessment**: We utilized Dask's describe().compute() function to obtain a comprehensive overview of the dataset's statistical properties, including measures like mean, standard deviation, minimum, and maximum values for numerical features. This provided valuable insights into the distribution and potential skewness of the data.

A screenshot of a computer

Description automatically generated

Figure 4: Statistical properties of the data

**3.1.4 Identifying Missing Values**: We employed Dask's isnull().sum().compute() function to identify and quantify missing values present within each column.

A white rectangular object with a white border

Description automatically generated

Figure 5: Null columns using Dask compute()

We can see that there are 22 columns which had Null values. Columns like diameter, albedo, diameter\_sigma, moid, moid\_ld, sigma\_e, sigma\_a, sigma\_q, sigma\_i, sigma\_w, sigma\_om, sigma\_ma, sigma\_ad, sigma\_n, sigma\_tp, sigma\_per and few other columns.

**3.1.5 Imputation of Missing Values**: To address missing data points, we implemented serial and parallel execution. SimpleImputer from sklearn.impute for serial and Dask-ML's SimpleImputer with the “median” strategy for parallel. This strategy fills missing values in each column with the median value of that column. This approach helps mitigate the potential biases introduced by missing data and ensures a more robust analysis.

A screenshot of a computer program

Description automatically generated

Figure 6: Filling Null values in the Column using SimpleImputer (Serial Execution)

A screenshot of a computer

Description automatically generated

Figure 7: Filling Null values in the Column using Dask-ML SimpleImputer (Parallel Execution)

We explored two approaches for handling missing values with Pandas and Dask Dataframe where Dask Dataframe outperformed Pandas in handling Null values.

### **3.2 Feature Importance**

This step aims to identify and isolate the most informative features within the dataset that best contribute to the classification task. In our case, the classification task likely involves predicting the class (type) of a celestial object based on the remaining features.

**3.2.1** **Target Variable Separation**: We separated the target variable (“class”), which represents the type of class asteroid belongs to, from the remaining features in the dataset. This distinction is essential as the model aims to predict the class based on the values of the other features.

**3.2.2 Model Selection and Training**: We have used “ExtraTreesClassifier” model to learn the relationships between the features and the target variable to select the important features. This model inherently performs feature extraction during the training process, identifying the features that best discriminate between different classes of Asteroid.

A screenshot of a computer code

Description automatically generated

Figure 8: Feature extraction using ExtraTreesClassifier (Serial Execution)

A screenshot of a computer program

Description automatically generated

Figure 9: Feature extraction using ExtraTreesClassifier with Joblib and Multiprocessing

(Parallel Execution)

**3.2.3 Feature Importance Analysis**: After training the model, we leveraged the feature\_importances\_ attribute of the “ExtraTreesClassifier” to understand the relative importance of each feature in predicting the class. This analysis helps us prioritize the most informative features that contribute most significantly to the classification task.

Finally, features with higher relative importance scores were selected as they deemed more influential in predicting the class labels.

A graph with different colored bars

Description automatically generated

Figure 10: Feature Importance during Serial Execution

A blue and white graph

Description automatically generated

Figure 11: Feature Importance using joblib and multiprocessing during Parallel Execution

Our analysis of feature extraction times revealed interesting results. While both serial and parallel implementation using joblib and multiprocessing produced identical feature sets, the parallel approach achieved significantly faster execution speeds. It demonstrates that parallel execution offers a significant advantage in terms of feature extraction speed. By utilizing multiple processors or cores, we achieved a speedup of 8.24x compared to serial execution.

### **3.3 Model Building**

**3.3.1 Dask and Multiprocessing for Scalable Model Training**: We leveraged Dask, a parallel computing framework, and multiprocessing library to efficiently train machine learning models on our large dataset. Both the dask and multiprocessing libraries distribute the computational load across multiple CPU cores, enabling significantly faster training times compared to traditional single-core execution.

**3.3.1.1 Serial Implementation for Random Forest**

**A screenshot of a computer program

Description automatically generated**

Figure 12: Code snippet of Random Forest implementation (Serial Execution)

**3.3.1.2 Random Forest Using Dask**

A screenshot of a computer program

Description automatically generated

Figure 13: Code snippet of Random Forest implementation using Dask

**3.3.1.3 Random Forest Using Multiprocessing**

**A screenshot of a computer program

Description automatically generated**

Figure 14: Code snippet of Random Forest implementation using Multiprocessing

**3.3.1.4 Serial Execution for KNN**

**A screenshot of a computer program

Description automatically generated**

Figure 15: Code snippet of KNN implementation (Serial Execution)

**3.3.1.5 KNN Using Dask**

A screenshot of a computer program

Description automatically generated

Figure 16: Code snippet of KNN implementation using Dask

**3.3.1.6 KNN Using Multiprocessing**

A screenshot of a computer code

Description automatically generated

Figure 17: Code snippet of KNN implementation using Multiprocessing

**3.3.1.7 Serial Execution for XGBoost**

**A screenshot of a computer program

Description automatically generated**

Figure 18: Code snippet of XGBoost implementation (Serial Execution)

**3.3.1.8 Dask XGBoost**

**A screenshot of a computer program

Description automatically generated**

Figure 19: Code snippet of XGBoost implementation using Dask XGBoost

**3.3.1.9 Serial Execution for Neural Network**

A screenshot of a computer

Description automatically generated

Figure 20: Serial Execution of Neural Network using pytorch

**3.3.1.6 Neural Network with Data Parallelism**

**A screenshot of a computer program

Description automatically generated**

Figure 21: Code snippet of Neural Network with Data Parallelism

Data parallelism is a technique in parallel computing where a large task is divided into smaller ones that can be executed simultaneously on multiple processors. We employed data parallelism across a system with 24 CPU cores to expedite the neural network training process. By distributing the training data among these CPUs, we enabled parallel computations during training. This technique significantly reduces training time compared to using a single CPU core.

The nn.DataParallel module in PyTorch[5] automatically distributes the input data and aggregates the gradients, making it easy to leverage multiple GPUs for faster training.

**3.3.1.7 Neural Network with Distributed Data Parallelism (DDP)**

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

Figure 22: Code snippet of Neural Network with Distributed Data Parallelism

To accelerate the training process for our complex neural network model, we leveraged Distributed Data Parallelism (DDP)[6]. DDP is a distributed training technique that partitions a large training dataset across multiple processing units, such as GPUs. In our experiment, we were fortunate to have access to two GPUs, which we utilized for parallel training with DDP. By distributing the workload across these GPUs, DDP enabled significant reductions in training time compared to using a single GPU. This approach has been particularly beneficial in training large datasets.

**3.3.2 Hyperparameter Tuning with Grid Search**: To identify the optimal configuration for each model (e.g., Random Forest, KNN), we employed GridSearchCV with Dask as well as multiprocessing library. This approach systematically evaluates different hyperparameter combinations in parallel, accelerating the search process.

**3.3.3 Performance Analysis Across CPU Configurations**: We trained the models on various CPU configurations (e.g., 4, 8, 16, 20) and measured the execution time. This analysis allowed us to assess the scalability and speedup achieved through parallel processing with Dask-ML. The results are visualized using plots that depict the relationship between CPU count and training time. Also, this gave us a chance to compare the serial execution time with parallel execution times.

**3.3.3.1 Random Forest Model**

A screen shot of a computer

Description automatically generated

Figure 23: Serial Execution of Grid Search CV for Random Forest

A graph with a line going up

Description automatically generated

Figure 24: Parallel Execution of Random Forest using Dask

A graph with a line

Description automatically generated

Figure 25: Parallel Execution of Random Forest using Multiprocessing

The baseline serial execution time for Random Forest is 1269.75 seconds. Both Dask and Multiprocessing significantly reduces the execution time when compared with serial execution on all CPU configurations (CPU = 4, 8, 16, and 20). Multiprocessing shows better performance than Dask on all CPU configurations. The speedup advantage of Multiprocessing over Dask increases with the number of CPUs. For instance, on 4 CPUs, Multiprocessing is 2.4 times faster than Dask, whereas on 20 CPUs, Multiprocessing is nearly 3 times faster.

**3.3.3.2 KNN Model**

A screenshot of a computer

Description automatically generated

Figure 26: Serial Execution of Grid Search CV for KNN

A graph with a line

Description automatically generated

Figure 27: Parallel execution of KNN using Dask

A graph with a line going up

Description automatically generated

Figure 28: Parallel Execution of KNN using Multiprocessing

The serial execution time for the KNN model is 260.91 seconds. Interestingly, for the KNN model, Dask shows better performance than Multiprocessing on all CPU configurations. The speedup advantage of Dask over Multiprocessing increases with the number of CPUs. For instance, on 4 CPUs, Dask is 1.48 times faster than Multiprocessing, whereas on 20 CPUs, Dask is nearly 2.1 times faster. However, we observed an increase in execution time when moving from 16 to 20 CPUs. This can likely be attributed to a combination of factors related to the overhead of managing a larger number of cores, such as increased synchronization, communication, and software overheads.

**3.3.3.3 XGBoost Model**

A screenshot of a computer

Description automatically generated

Figure 29: Serial Execution of Grid Search CV for XGBoost

A graph with a line

Description automatically generated

Figure 30: XGBoost Dask in Parallel

The baseline serial execution time for XGBoost observed was 180.13 seconds. DaskXGBoost successfully reduced XGBoost training time by utilizing parallel processing across multiple CPU cores. The execution time generally decreased with an increasing number of CPUs (4, 8, and 16), achieving a speedup. However, we observed an increase in execution time when moving from 16 to 20 CPUs. This can likely be attributed to a combination of factors related to the overhead of managing a larger number of cores, such as increased synchronization, communication, and software overheads. These factors can limit the effectiveness of adding more cores beyond a certain point.

**3.3.3.4 Neural Network on 24 CPUs**

A graph of a graph

Description automatically generated with medium confidence

Figure 31: Neural Network Execution of Data Parallel

**3.3.3.5 Distributed Data Parallel**

**3.3.3.5.1 Distributed Data Parallel on 24 CPUs**

**A screenshot of a computer

Description automatically generated**

Figure 32: Neural Network Execution of Distributed Data Parallel

**3.3.3.5.2 Distributed Data Parallel on 1 GPU**

A screenshot of a computer

Description automatically generated

Figure 33: Execution time of DDP on 1 GPU

**3.3.3.5 Distributed Data Parallel on 2 GPUs**

**A screenshot of a computer

Description automatically generated**

Figure 34: Execution time of DDP on 2 GPU

We used a Distributed Data Parallel (DDP) model to train an asteroid classification model on two GPUs. The DDP model achieved a significant improvement in execution time compared to using a single GPU. The execution time was reduced from 909.73 seconds to 483.1 seconds, which is a reduction of almost half (46.6%). These results suggest that DDP can be an effective way to parallelize the training of asteroid classification models.

**3.3.4 Model Selection and Evaluation**: Based on the obtained results, we selected the best performing parameters for further analysis. We evaluated the model's performance on the unseen test set using metrics like accuracy, classification report, and confusion matrix.

### **3.4 Scope of Parallelization**

**3.4.1 Parallel Data Loading:** Utilized Dask Dataframe to load the large dataset. Dask enables loading large datasets in parallel across multiple cores.

**3.4.2 Multiprocessing:** We used joblib.parallel\_backend('multiprocessing') which sets the backend for parallel execution within scikit-learn to leverage the multiprocessing module in Python.

**3.4.3 Joblib Parallel Backend**: We used joblib.parallel\_backend('dask') context manager that configures scikit-learn to leverage Dask's parallel capabilities during model fitting. This enables parallel execution of the model training process on the data.

**3.4.4 Dask Client and Workers**: We used a Dask client to connect to a cluster of worker processes. These workers handle the parallel execution of tasks distributed by the client, such as hyperparameter tuning with GridSearchCV.

**3.4.5 GridSearchCV with Dask**: The GridSearchCV function from Dask-ML is employed to perform hyperparameter tuning in parallel. This allows for the evaluation of multiple hyperparameter combinations simultaneously across different workers in the Dask cluster.

### **3.5 Performance Evaluation**

**3.5.1 Loading the dataset**:

|  |  |  |
| --- | --- | --- |
| **Execution Methods** | **Library** | **Elapsed Time(seconds)** |
| Serial | Pandas | 1.76 |
| Parallel | Dask Dataframe | 0.0505 |

Parallel Execution has significantly reduced the time taken to load the dataset. Dask Dataframe significantly reduces the loading time by 34.8 times due to its ability to distribute tasks across multiple cores.

**3.5.2 Feature Extraction:**

|  |  |
| --- | --- |
| **Execution Methods** | **Elapsed Time(seconds)** |
| Serial | 103 |
| Parallel | 12.5 |

Parallel execution significantly reduces the feature extraction time by 8.24 times.

**3.5.3 Results of parallelizing the model training on multi-CPUs/GPUs**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Execution Method** | **Model** | **Elapsed Time(seconds)** | | |
| Serial | Random Forest | 1269.75 | | |
| KNN | 260.91 | | |
| XG Boost | 180.13 | | |
| Neural Network | 175.59 | | |
|  |  | No of CPUs/GPUs | Dask | Multiprocessing |
| Parallel | Random Forest | CPU =4 | 408.51 | 167.68 |
| CPU =8 | 292.67 | 167.36 |
| CPU =16 | 225.94 | 165.38 |
| CPU =20 | 212.05 | 163.33 |
| KNN | CPU = 4 | 176.17 | 190.3 |
| CPU = 8 | 140.52 | 189.04 |
| CPU = 16 | 89.16 | 188.5 |
| CPU = 20 | 89.92 | 189.5 |
| XG Boost | CPU = 4 | 38.21 | |
| CPU = 8 | 29.64 | |
| CPU = 16 | 24.64 | |
| CPU = 20 | 27.17 | |
| Neural Network | CPU = 28 | 180.68 | |
| Distributed Data Parallel | GPU = 1 | 909.73 | |
| GPU = 2 | 483.1 | |

We observed that parallel processing leads to significant speedups compared to serial processing for all the models (Random Forest, KNN, XGBoost, and Neural Network). For instance, using 4 CPUs for KNN in parallel processing resulted in an execution time of 408.51 seconds, whereas serially it took 260.91 seconds. This translates to a speedup of around 1.5x. As the number of CPUs increases, the execution time in parallel processing tends to decrease for all the models. For instance, for Random Forest model, the execution time reduced from 408.51 seconds (4 CPUs) to 163.33 seconds (20 CPUs), which is a speedup of around 2.5x. There seems to be limited benefit of using more than 16 CPUs for some of the models (KNN, XGBoost). For instance, the execution time for KNN with 16 CPUs is 89.16 seconds, whereas with 20 CPUs it’s 88.92 seconds - a negligible difference. Even a single GPU can significantly outperform CPUs for some models (Neural Network). For instance, using a single GPU resulted in an execution time of 909.73 seconds for Neural Network, whereas 20 CPUs took 180.68 seconds - resulting in a speedup of over 5x.

**3.5.4 Speedup comparison of the models on multi-CPUs/GPUs**

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Speedup (times)** | | |
|  | No of CPUs | Dask | Multiprocessing |
| Random Forest | CPU = 4 | 3.11 | 7.57 |
| CPU = 8 | 4.34 | 7.59 |
| CPU = 16 | 5.62 | 7.68 |
| CPU = 20 | 5.99 | 7.77 |
| KNN | CPU = 4 | 1.48 | 1.37 |
| CPU = 8 | 1.86 | 1.38 |
| CPU = 16 | 2.93 | 1.38 |
| CPU = 20 | 2.93 | 1.38 |
| XGBoost | CPU = 4 | 4.71 | |
| CPU = 8 | 6.08 | |
| CPU = 16 | 7.31 | |
| CPU = 20 | 6.63 | |
| Neural Network | CPU = 28 | 0.97 | |

**3.5.5 Efficiency comparison of the models on multi-CPUs/GPUs**

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Efficiency** | | |
|  | No of CPUs | Dask | Multiprocessing |
| Random Forest | CPU = 4 | 0.78 | 1.89 |
| CPU = 8 | 0.54 | 0.95 |
| CPU = 16 | 0.35 | 0.48 |
| CPU = 20 | 0.3 | 0.39 |
| KNN | CPU = 4 | 0.37 | 0.34 |
| CPU = 8 | 0.23 | 0.17 |
| CPU = 16 | 0.18 | 0.09 |
| CPU = 20 | 0.15 | 0.07 |
| XGBoost | CPU = 4 | 1.18 | |
| CPU = 8 | 0.76 | |
| CPU = 16 | 0.46 | |
| CPU = 20 | 0.33 | |
| Neural Network | CPU = 28 | 0.03 | |

**3.5.6 Accuracy**

A table with numbers and letters

Description automatically generatedA table with numbers and letters

Description automatically generated

Figure 35: Accuracy of KNN and XGBoost from Reference Paper[1]

Our models achieved high accuracy (above 99%) for asteroid classification using Random Forest, KNN, and XGBoost algorithms. While this is promising, it's important to consider potential overfitting, which can occur when models memorize training data specifics and perform poorly on unseen data. Our focus prioritized parallelization for faster training over extensive hyperparameter tuning to mitigate overfitting. This approach ensured efficient model development. The reference paper[1] also achieved similar high accuracy (99.99%) using KNN, XGBoost for asteroid classification. This suggests high accuracy might be achievable for this specific task, particularly with datasets well-suited to the chosen algorithms. Future iterations can explore techniques like grid search for hyperparameter optimization and early stopping to prevent memorization of noise in the data. These approaches can help achieve a balance between accuracy and generalizability on unseen data.

**3.5.6 Challenges Involved:**

* Having a greater number of CPUs doesn't boost performance. As we add more CPUs, the overhead of managing communication and task scheduling between these CPUs also increases.
* Based on our observations, using 16 CPUs seems to be the optimal configuration for our hardware and workload. Picking the right configuration is crucial to get efficient performance and utilization of resources.
* There were some issues while configuring the Dask Dasboard in OOD cluster as we (users) don’t have access to local host. Although we used local cluster to connect to the dask dashboard and we were able to view the dask graphs, and all the other features in the dashboard. But this limited our ability to dynamically configure the dask client according to the number of CPUs.

A screenshot of a computer

Description automatically generated

Figure 36: Dask Dashboard for Random Forest Execution

## **Conclusion**

The data indicates that parallelization markedly decreases execution times for machine learning models as compared to serial execution, with more pronounced reductions as CPU cores increase. However, the efficiency gains tend to plateau beyond approximately 16 cores. While Random Forest and XG Boost models benefit significantly from parallel processing, the performance improvements for KNN and Neural Networks vary based on the chosen parallelization method. It is essential to effectively manage the number of cores and select the appropriate parallelization strategy to optimize performance without excessive resource utilization.

Additionally, when employing multiprocessing, it's important to account for overhead costs, which can negate the benefits if not managed properly. Using Dask Dataframe can notably enhance the processing speed of operations typically handled by Pandas, offering a viable solution for speeding up computations. These advancements in parallel processing techniques could also enhance the efficiency of modeling scenarios such as predicting asteroid threats, providing quicker and more reliable estimates.

While data parallelism could have been effective on multiple CPUs, but it wasn’t as you could see the time elapsed on serial execution was less than executing on parallel this is because it is generally less efficient on CPUs due to their architecture compared to GPUs. Similarly, while executing Distributed data parallel on GPUs, effective utilization of resources and computation time also relies on the configuration we choose to execute the model.

## **References:**

[1] Hossain, Mir Sakhawat & Zabed, Md. (2023). Machine Learning Approaches for Classification and Diameter Prediction of Asteroids dx.doi.org/10.1007/978-981-19-7528-8\_4 Accessed 12 March 2024.

[2] Dhariwal, Kunal. “Pandas with Dask, for an Ultra-Fast Notebook.” Medium, Towards Data Science, 31 Oct. 2021, https://towardsdatascience.com/pandas-with-dask-for-an-ultra-fast-notebook-e2621c3769f.

[3] NASA. (2020, April). JPL Small-Body Database Search Engine. https://ssd.jpl.nasa.gov/tools/sbdb\_lookup.html#/

[4] Dask Development Team. (n.d.). Dask DataFrame. Dask Documentation. https://docs.dask.org/en/stable/dataframe-create.html

[5] PyTorch. (n.d.). Documentation. <https://pytorch.org/docs/stable/index.html>

[6] Hu, L. (2020). Distributed parallel training: Data parallelism and model parallelism. Towards Data Science. https://towardsdatascience.com/distributed-parallel-training-data-parallelism-and-model-parallelism-ec2d234e3214