# Scalable Cloud-Based Distributed Computing for Efficient Big Data Analytics: A Dask Integration Approach

**ENGR-E516 Engineering Cloud Computing**

Anirudh Penmatcha    Dilip Nikhil Francies    Subhadra Mishra

## 1. Introduction

The exponential growth of data in recent years has transformed the landscape of data science, presenting both unprecedented opportunities and significant challenges. With approximately 2.5 quintillion bytes of data generated daily, the demand for scalable tools and frameworks to analyze and derive insights from this vast volume of information has never been more pressing. In the single year of 2017, the Weather Channel received an astounding 18 million forecast requests per minute, while an overwhelming 103 million spam emails inundated inboxes every minute. Such statistics underscore the magnitude of data generation and the urgent need for efficient data processing methodologies. This has led to the development of a plethora of frameworks for distributed data analysis, including batch [4], streaming [7], and graph [11] processing systems.

Traditionally, data scientists have leveraged the Python Open Data Science Stack, relying on tools like Pandas, SciPy, NumPy, and scikit-learn for data analysis and building predictive models. While effective for smaller datasets that can fit into memory, these tools often falter when confronted with the monumental scale of contemporary data because these were not designed to scale beyond a single machine. The complexity intensifies as many tools, like PySpark, pose steep learning curves. Analysts often resort to rewriting their computations using more scalable tools, sometimes in different languages, leading to frustration and slowed inference. Additionally, PySpark users may need to migrate their codebase to Scala or Java for optimal Spark utilization [6], further complicating matters. Furthermore, the lag in feature updates for PySpark compared to Java and Scala adds to the challenge. Considering these hurdles, what options do Python-native developers have for running analytics on big data leveraging the distributed computing power of the cloud?

In this project, we propose to build a platform designed to be dynamically scalable based on user demand. At its core, it is the integration of Dask[10], a parallel execution framework, with JupyterHub, containerized and deployed on a cloud instance. We intend to benchmark the performance of Dask as a distributed computing framework on our cluster by conducting computationally intensive hyperparameter tuning of tree-based XGBoost algorithm on big data. Through systematic variations in input format, chunk size, task schedulers, worker nodes, clusters, and threading configurations, we seek to quantify the performance and compare it to baseline values obtained from running the program on the instance without distributing the workload. Our evaluation benchmarking serves two purposes: 1) to compare the performance of running computationally intensive ML algorithms with and without parallelizing the workload with Dask on the cloud. 2) To understand in depth the many components of distributed computing that impact its performance.

## 2. Related Work

### 2.1. Dask

Dask is a powerful open-source Python library designed for parallel computing tasks [10]. It seamlessly scales Python code from multi-core local machines to expansive distributed clusters in the cloud. The library offers low-level APIs, empowering programmers to execute custom algorithms in parallel efficiently. Dask employs a sophisticated work-stealing scheduler, meticulously optimized to execute task graphs with utmost efficiency. Interestingly, even a completely random scheduler remains surprisingly competitive with Dask's built-in scheduler [1]. However, the primary bottleneck of Dask lies in its runtime overhead. Notably, Dask demonstrates superior performance compared to Spark with up to two workers. Conversely, Spark exhibits improved performance with an increase in the number of workers [1], particularly noticeable with four and thirty-two workers. Dask's effective data partitioning strategy becomes particularly evident with fewer workers, enabling optimal resource utilization and minimizing execution times.

### 2.2. Sanzu

Sanzu serves as an indispensable tool for assessing systems handling data processing and analytics tasks [12]. Its benchmarking approach encompasses both micro and macro benchmarks. The micro-benchmark isolates and evaluates basic operations, including tasks related to reading and writing data, data manipulation, statistical analysis, machine learning, and time series analysis. Conversely, the macro benchmark assesses analytics applications by simulating real-world scenarios, focusing particularly on sports and smart grid analytics. This evaluation encompasses five prominent data science frameworks and systems: R, Anaconda Python, Dask, PostgreSQL (MADlib), and PySpark.

### 2.3. Spark

Apache Spark is an open-source general-purpose cluster computing framework developed by the AMP lab at the University of California, Berkeley[9]. The core of Spark is written in Scala, runs on the Java virtual machine (JVM), and offers a functional programming API to Scala. This reference study [13] undertook a rigorous benchmarking analysis comparing Dask and Apache Spark, alongside the single-node SciKit Learn framework. Conducted on the EMNIST dataset, consisting of subsets ranging from 50k to 250k samples, assessments were performed across diverse operating systems and levels of parallelization. Notably, findings revealed a slight edge in favor, Dask showcased prowess in data frame manipulation, Apache Spark exhibited superior end-to-end processing performance, particularly evident on larger datasets, with F1 scores comparable to those achieved by Dask. Additionally Dugre, Hayot-Sasson, and Glatard [5] compared Apache Spark and Dask across three neuroimaging applications, finding no significant performance differences. They noted that differences in engine overheads did not affect performance due to compensating lower transfer times when data transfers utilized bandwidth fully. This suggests a need for future research on strategies to reduce the impact of data transfers on application performance.

### 2.4. XGBoost

XGBoost (Extreme Gradient Boosting) is a scalable tree boosting system widely utilized in machine learning, featuring a sparsity-aware algorithm and weighted quantile sketch for improved performance on sparse data.

Leveraging insights on cache access patterns, data compression, and sharding, XGBoost achieves scalability beyond billions of examples with minimal resource consumption [3].

Junda Chen et al. [2] demonstrated that the optimal cluster size for the cost-effectiveness of distributed XGBoost training is primarily determined by the sample size, with datasets containing more entries benefiting from increased parallelism. Conversely, increasing dimensionality did not yield significant gains from parallel processing. The block size emerged as a critical factor affecting both matrix creation performance and training efficiency. Furthermore, Zoya Masih [8] demonstrates Dask-ML's linear scalability in execution time and active data storage with increasing LAZ file (popular vector formats for storing LiDAR data as point clouds) sizes. Consistent CPU utilization at around 25 percent indicates efficient resource usage, while worker bandwidth displays satisfactory communication. Memory-mapped access and deferred loading effectively address challenges in LAZ file processing, resulting in moderate memory scaling and efficient resource utilization.

## 3. Proposed Design and Objectives

As mentioned in our midterm report, in our study's initial phase, we aimed to deploy Dask gateway on Kubernetes and configure JupyterHub for private Dask clusters, preferably on AWS, Jetstream2, or Saturn Cloud, with a medium-scale 2Xlarge instance.

We have opted to utilize AWS, and at present, the Amazon Elastic Container Service cluster 'dask-cluster-2' is operational, comprising three services: 'dask-scheduler', 'dask-worker', and 'dask-notebook'. Understanding the intricacies and specifics of their setup is crucial at this juncture.

The Architecture of the system behind our cluster consists of four distinct aspects: the Jupyter server, the scheduler, the client, and the workers. The Jupyter server serves as a frontend for the user to submit jobs. The scheduler then receives the jobs and divides them among the workers, the client is the user who submits the jobs.

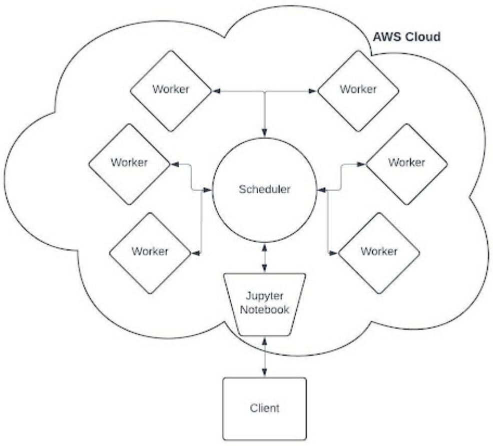
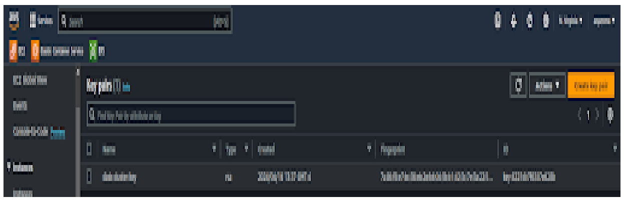


**Figure 1:** Architecture of the Dask Cluster

To set up the cluster it involved five major steps: Create the ECS cluster, configure the network settings for the cluster, establish a shared drive using Elastic File System (EFS), reserve storage, build, and deploy the Docker images within Elastic Container Repository (ECR), and finally connecting to the cluster.

Before we created the cluster, we first needed to create a Key Pair in EC2 that will be needed later on.

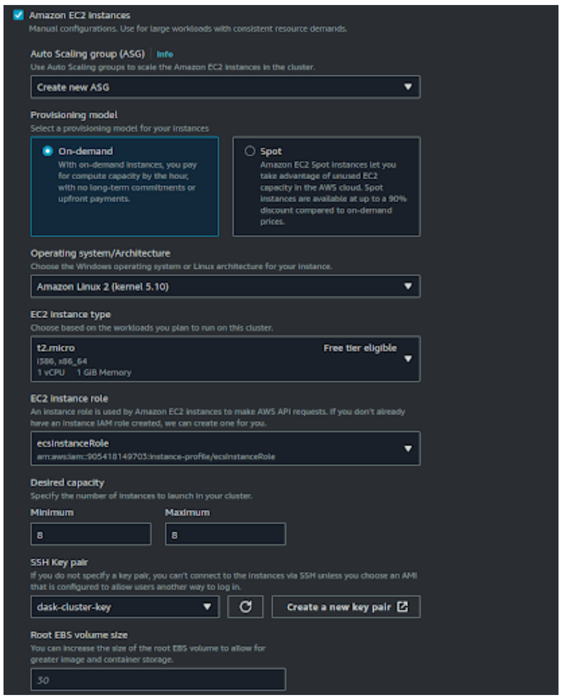


Key pairs (1)

| Name | Type | Created | Fingerprint | ID |
|---|---|---|---|---|
| dask-cluster-key | rsa | [illegible] | [illegible] | [illegible] |

**Figure 2:** Dask Cluster Key

With the Key Pair generated, the first major step is to create an ECS Cluster. This was done using the ECS Create Cluster Wizard. The operating system chosen was Amazon Linux 2. Next, for the purpose of setting up and testing the functionality of our cluster, we opted to use only the t2.micro option, which falls under the free tier category. This choice allows us to minimize costs until everything is ready, at which point we can make use of premium services for a minimum amount of time. Following this, we specified 8 instances as the desired capacity, and the key pair used was the one we created earlier. The remaining options were left at default settings.



[✓] Amazon EC2 instances
Manual configurations. Use for large workloads with consistent resource demands.

Auto Scaling group (ASG) Info
Use Auto Scaling groups to scale the Amazon EC2 instances in the cluster.
Create new ASG

Provisioning model
Select a provisioning model for your instances
[● On-demand] With on-demand instances, you pay for compute capacity by the hour, with no long-term commitments or upfront payments.
○ Spot Amazon EC2 Spot instances let you take advantage of unused EC2 capacity in the AWS cloud. Spot instances are available at up to a 90% discount compared to on-demand prices.

Operating system/Architecture
Choose the Windows operating system or Linux architecture for your instance.
Amazon Linux 2 (kernel 5.10)

EC2 instance type
Choose based on the workloads you plan to run on this cluster.
t2.micro Free tier eligible
i386, x86_64
1 vCPU 1 GiB Memory

EC2 instance role
An instance role is used by Amazon EC2 instances to make AWS API requests. If you don't already have an instance IAM role created, we can create one for you.
ecsInstanceRole
arn:aws:iam::905418149703:instance-profile/ecsInstanceRole

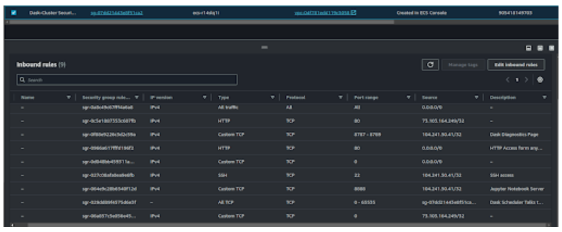
Desired capacity
Specify the number of instances to launch in your cluster.
Minimum: 8
Maximum: 8

SSH Key pair
If you do not specify a key pair, you can't connect to the instances via SSH unless you choose an AMI that is configured to allow users another way to log in.
dask-cluster-key
Create a new key pair

Root EBS volume size
You can increase the size of the root EBS volume to allow for greater image and container storage.
30

**Figure 3:** ECS Create Cluster Wizard

With this, we had 8 instances running in our cluster, and we added the cluster's firewall inbound rules to ensure communication between all the nodes.



Dask-Cluster-Securi... sg-07dd21443e0f51ca2 ecs-r14dq1l vpc-0d[illegible] Created in ECS Console 905418149703

Inbound rules (9)

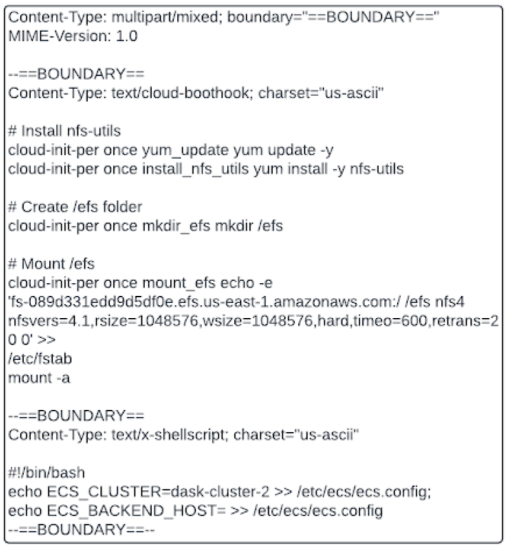
| Name | Security group rule... | IP version | Type | Protocol | Port range | Source | Description |
|---|---|---|---|---|---|---|---|
| – | sgr-0a0e49e07ff4a6a6 | IPv4 | All traffic | All | All | 0.0.0.0/0 | – |
| – | sgr-0c5e1807353c087fb | IPv4 | HTTP | TCP | 80 | 73.105.164.249/32 | – |
| – | sgr-0f80e9226c5d2c59a | IPv4 | Custom TCP | TCP | 8787 - 8789 | 104.241.50.41/32 | Dask Diagnostics Page |
| – | sgr-0986a617fffd186f2 | IPv4 | HTTP | TCP | 80 | 0.0.0.0/0 | HTTP Access form any... |
| – | sgr-0d048b6e459511a... | IPv4 | Custom TCP | TCP | 0 | 0.0.0.0/0 | – |
| – | sgr-027c08afabeadebfb | IPv4 | SSH | TCP | 22 | 104.241.50.41/32 | SSH access |
| – | sgr-064e9c28b6548f12d | IPv4 | Custom TCP | TCP | 8888 | 104.241.50.41/32 | Jupyter Notebook Server |
| – | sgr-028d8894d575d6a3f | – | All TCP | TCP | 0 - 65535 | sg-07dd21443e0f51ca... | Dask Scheduler Talks t... |
| – | sgr-06a057c5e058e45... | IPv4 | Custom TCP | TCP | 0 | 73.105.164.249/32 | – |

**Figure 4:** Inbound Rules for the Cluster

Following this, we needed to create persistent storage using EFS. Once the EFS storage was set up, we had to create a launch instruction for the instances to mount the EFS on startup. To accomplish this, we made use of the Launch Template feature with the following commands:

Now, the next step involves providing the launch template to a service that will use it for launching the instances in our cluster. For this, the Auto Scaling Group is utilized. We simply supply the launch template and specify the desired number of instances to launch.

With the instances running, the network setup completed, and storage mounted, we then needed to deploy our Docker images of the scheduler, notebook, and workers. First, the images had to be built and deployed on the AWS repository. Then, the image URI was assigned to a task definition with instructions on port mapping and the



```
Content-Type: multipart/mixed; boundary="==BOUNDARY=="
MIME-Version: 1.0

--==BOUNDARY==
Content-Type: text/cloud-boothook; charset="us-ascii"

# Install nfs-utils
cloud-init-per once yum_update yum update -y
cloud-init-per once install_nfs_utils yum install -y nfs-utils

# Create /efs folder
cloud-init-per once mkdir_efs mkdir /efs

# Mount /efs
cloud-init-per once mount_efs echo -e
'fs-089d331edd9d5df0e.efs.us-east-1.amazonaws.com:/ /efs nfs4
nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2
0 0' >>
/etc/fstab
mount -a

--==BOUNDARY==
Content-Type: text/x-shellscript; charset="us-ascii"

#!/bin/bash
echo ECS_CLUSTER=dask-cluster-2 >> /etc/ecs/ecs.config;
echo ECS_BACKEND_HOST= >> /etc/ecs/ecs.config
--==BOUNDARY==--
```

**Figure 5:** Launch Template-User Data

number of instances. Following this, a service for all the tasks was initiated. Once the services had been successfully initiated, we were finally able to connect to our clusters. We simply needed to obtain the IP addresses of the instances for the scheduler and notebook and paste them into the browser, along with their respective port numbers (8787 for the scheduler and 8888 for the notebook). In our code, for the Dask Client, we had to manually specify the location of the scheduler by providing the IP address of the scheduler and port number as 8786. By running some test code, we were able to observe that the cluster was running successfully.

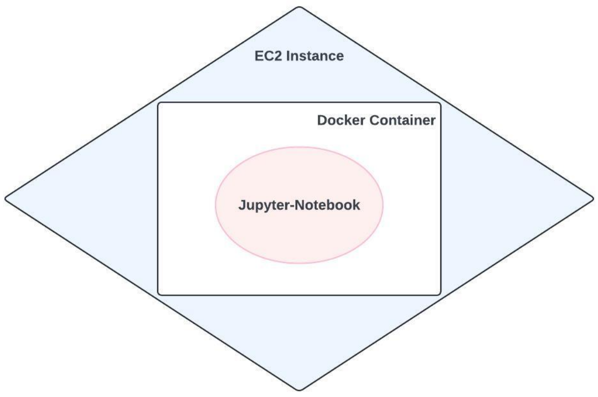
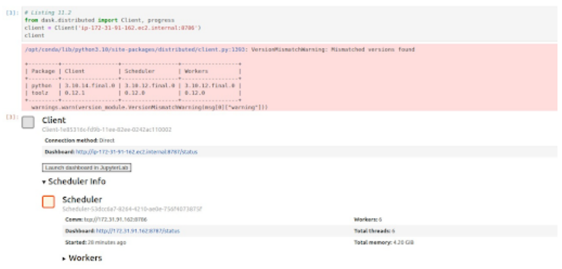


**Figure 6:** A visualization of the notebook instance

After verifying the status of the services, we used the diagnostic dashboard to observe the cluster. From there, we uploaded a notebook to the Jupyter Notebook server and ran code through the Distributed Client interface. The Dask diagnostics page allowed us to monitor job execution and cluster health by enabling us to use the DAG view to observe task dependencies and performance in real-time.



```
[3]: # Listing 11.2
     from dask.distributed import Client, progress
     client = Client('ip-172-31-91-162.ec2.internal:8786')
     client

/opt/conda/lib/python3.10/site-packages/distributed/client.py:1393: VersionMismatchWarning: Mismatched versions found

+---------+-----------------+-----------------+-----------------+
| Package | Client          | Scheduler       | Workers         |
+---------+-----------------+-----------------+-----------------+
| python  | 3.10.14.final.0 | 3.10.12.final.0 | 3.10.12.final.0 |
| toolz   | 0.12.1          | 0.12.0          | 0.12.0          |
+---------+-----------------+-----------------+-----------------+
  warnings.warn(version_module.VersionMismatchWarning(msg[0]["warning"]))
```

[3]: Client
Client-1e85316c-fd9b-11ee-82ee-0242ac110002
Connection method: Direct
Dashboard: http://ip-172-31-91-162.ec2.internal:8787/status

Scheduler
Scheduler-53dcc6a7-8264-4210-ae0e-756f4073875f
Comm: tcp://172.31.91.162:8786
Workers: 6
Dashboard: http://172.31.91.162:8787/status
Total threads: 6
Started: 28 minutes ago
Total memory: 4.20 GiB

**Figure 7:** Sample Dask Code Showing the connection to the Cluster

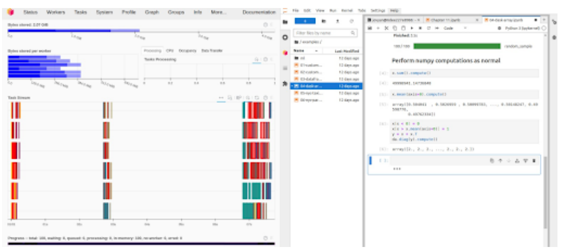


**Figure 8:** Dask's Diagnostic Page

Subsequently, we had mentioned developing a specialized Dask program for benchmarking hyperparameter tuning on an XGBoost algorithm, targeting a dataset of 1.4 GB with over 1.5 million records and 119 features.

To perform the bench marking study, we aim to integrate Dask and XGBoost for hyperparameter optimization and clssification tasks.

Denoting our dataset as $D = \{(x_i, y_i)\}_{i=1}^{n}$, where $x_i$ represents the $i$-th feature vector and $y_i$ represents the corresponding label. XGBoost minimizes a loss function $L(y_i, \hat{y}_i)$, where $\hat{y}_i$ is the predicted label. The objective function can be written as:

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} L(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k) \qquad (1)$$

where $\phi = \{f_k\}_{k=1}^{K}$ represents the model parameters, $K$ is the number of trees, and $\Omega(f_k)$ is the regularization term.

Optuna is a handy tool that tunes the hyperparameters of machine learning models automatically, using a method called Bayesian optimization. It explores different settings to find the ones that make the model perform its best. Denoting the hyperparameters of the XGBoost model as $\theta$. Optuna aims to find the optimal $\theta^*$ that minimizes the validation loss:

$$\theta^* = \arg\min_{\theta} \mathcal{L}_{\text{val}}(\theta) \tag{2}$$

where $\mathcal{L}_{\text{val}}(\theta)$ is the validation loss function.

Combining Dask and Optuna accelerates the hyperparameter optimization process for model development. Dask distributes the evaluation of various hyperparameter settings across multiple workers, enhancing the exploration of the parameter space and speeding up the process. This approach enables the discovery of optimal configurations that maximize Dask's potential, making it a powerful tool for research on distributed computing frameworks.

In our study, we are relying on CPU and wall time as essential benchmarks to assess Dask's computational efficiency and scalability. CPU time measures the duration dedicated to CPU-bound computations, while wall time encompasses all elapsed time, including I/O operations and waiting periods. These metrics provide detailed insights into Dask's performance, highlighting areas for improvement. Additionally, they establish standardized benchmarks for comparing different experiments and configurations, facilitating informed decisions to optimize Dask's parameters for enhanced real-world usability.

## 4. Observations and Analysis

The results demonstrate a significant improvement in speed when using Dask, with Dask completing the optimization tasks in approximately 5.44 seconds, whereas the vanilla XGBoost implementation took 18.9 seconds, converging to similar accuracy in both cases.

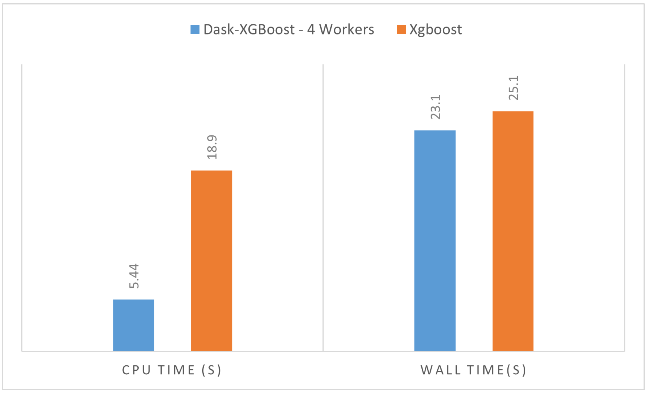


**Figure 9:** CPU and Wall time comparison of DASK-Xgboost vs Vanilla Xgboost implementation

When experimenting with varying the number of worker nodes on the Dask cluster, we observed an average accuracy of 0.967, indicating that the number of workers does not significantly impact the final model's accuracy. Hence, efficient allocation of resources is essential for maximizing resource utilization.

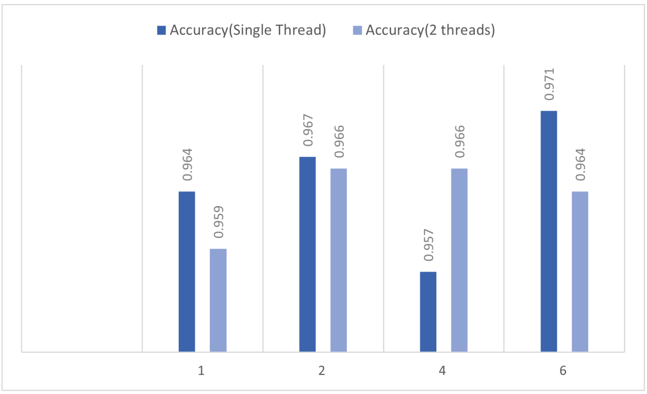


**Figure 10:** Accuracies obtained across different cluster configuration

Subsequently, we bench-marked CPU time and Wall time for single-threaded and dual-threaded worker nodes. Increasing the number of threads decreased both Wall time, and CPU time as it enables parallel execution of tasks, leading to more efficient utilization of computational resources.

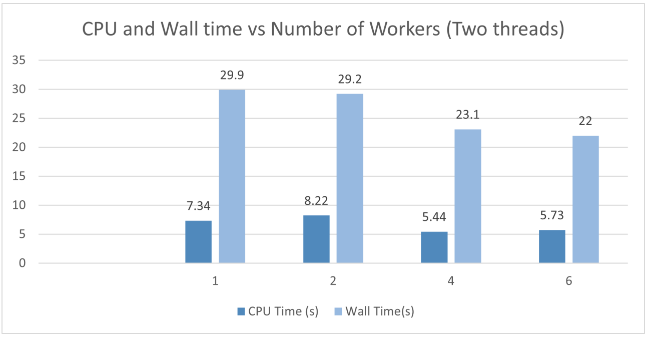


**Figure 11:** CPU and Wall for dual threaded machines

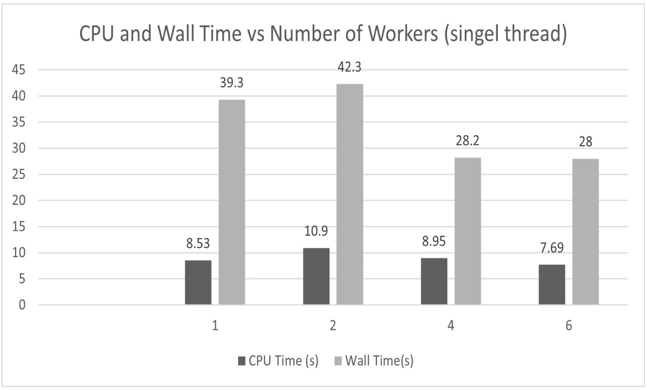


**Figure 12:** CPU and Wall time for single threaded machines

Furthermore, we compared the total time

taken to read, write, and perform basic computations on CSV and Parquet files between Pandas and Dask which can be seen in the figure 13. The results indicate that Pandas takes five times longer compared to Dask. This can be attributed to Dask's efficient handling of large datasets by partitioning them into smaller chunks and distributing computations across multiple cores or machines. This parallelization minimizes disk swapping and maximizes CPU utilization, resulting in significantly shorter processing times compared to Pandas' in-memory approach.

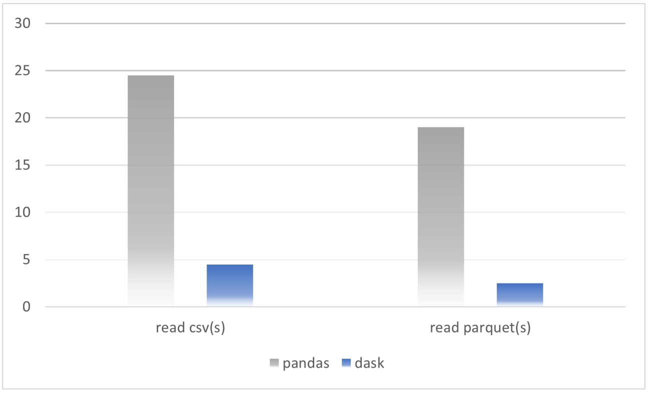


**Figure 13:** DASK vs Pandas

Additionally, we observed in figure 14 that the objective values converge to higher values more quickly on single-threaded machines compared to dual-threaded machines through focused resource allocation and rapid iteration through the parameter space. This difference underscores the trade-off between computational efficiency and exploration of the parameter space in optimization tasks.

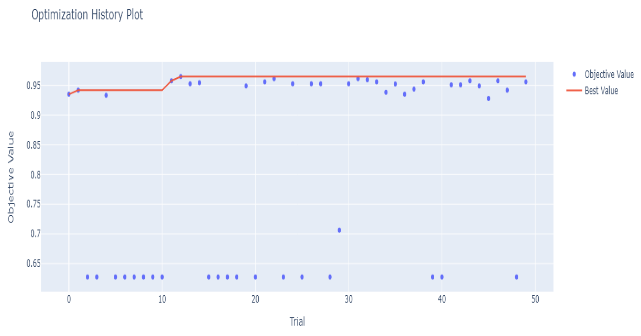


**Figure 14:** Objective value vs trials - Single threaded

In Dask-parallelized computations, single-threaded machines lean towards fine-tuning the learning rate to ensure convergence, while dual-threaded setups allocate resources across a broader spectrum of hyperparameter dimensions, ensuring a more comprehensive optimization process. This can be observed from the figure 15. Similarly, in the parallel coordinate plot 17, darker lines indicate hyperparameters striving towards higher objective values, while lighter lines indicate convergence towards lower objective values. This phenomenon occurs because, in single-threaded setups, resources are concentrated on a narrower range of hyperparameters, resulting in more pronounced movements towards higher objective values. In contrast, dual-threaded setups distribute resources across multiple hyperparameter dimensions, leading to a smoother optimization process with lighter lines indicating convergence towards lower objective values across various dimensions.

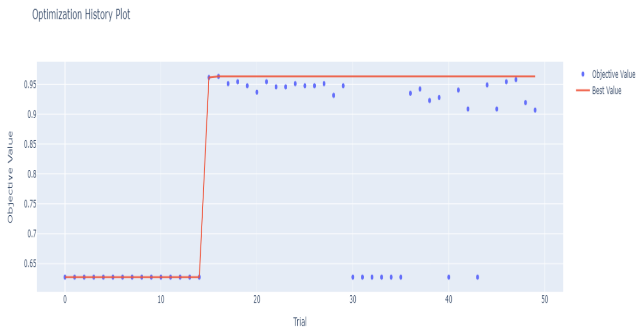


**Figure 15:** Objective value vs trials - Dual threaded

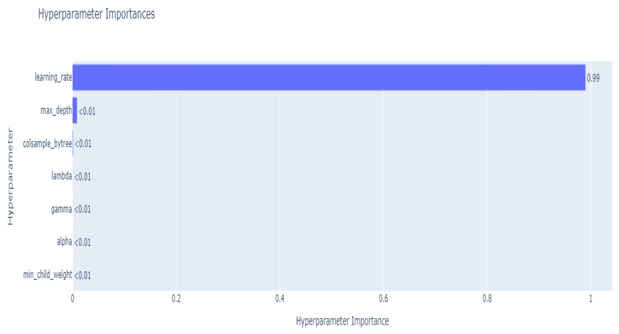


**Figure 16:** Hyper parameter importance - Single threaded worker nodes

The research study also included computations on a 10-dimensional NumPy matrix with dimensions (10000, 10000, 10), where various performance metrics were compared. The results of these comparisons are illustrated in the table and plot below.

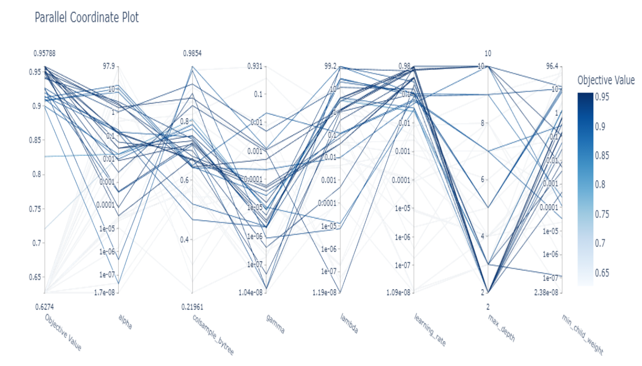


**Figure 17:** Parallel coordinate plot - Single threaded worker nodes

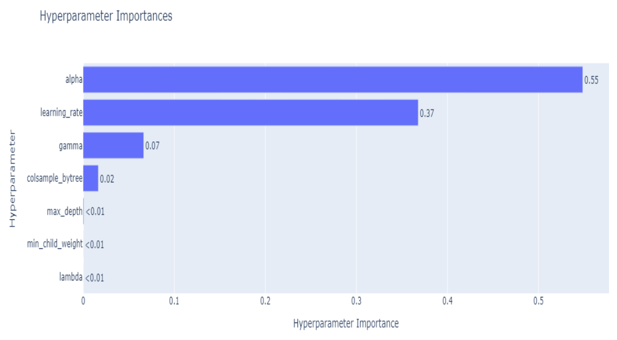
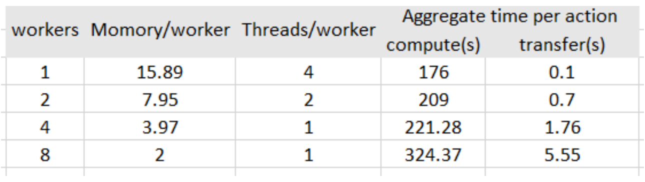


**Figure 18:** Hyper parameter importance - Dual threaded worker nodes



| workers | Momory/worker | Threads/worker | Aggregate time per action | |
|---|---|---|---|---|
| | | | compute(s) | transfer(s) |
| 1 | 15.89 | 4 | 176 | 0.1 |
| 2 | 7.95 | 2 | 209 | 0.7 |
| 4 | 3.97 | 1 | 221.28 | 1.76 |
| 8 | 2 | 1 | 324.37 | 5.55 |

**Figure 21:** Aggregate compute times for varied number of workers

The findings indicate that as the number of workers increases, there is a corresponding increase in computation times for reading and writing. A consistent chunk size of (1000, 1000, 5) was maintained to analyze the read performance. This trend becomes more pronounced when scaling workers from 4 to 8, suggesting diminishing returns in computational efficiency with higher worker nodes.

The increase in computation times as the number of workers increases can be attributed to factors such as increased overhead, resource contention, scaling limitations, and bottlenecks. These challenges highlight the importance of optimizing resource allocation and addressing scalability limitations for optimal performance in distributed computing environments.

Analysis of the task graphs below (figure 24 25 26) indicates that the white gaps between

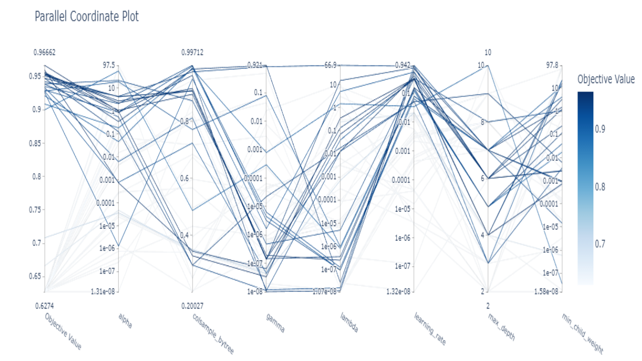
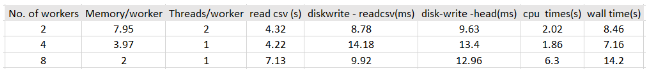


**Figure 19:** Parallel coordinate plot - Dual threaded worker nodes



| No. of workers | Memory/worker | Threads/worker | read csv (s) | diskwrite - readcsv(ms) | disk-write -head(ms) | cpu times(s) | wall time(s) |
|---|---|---|---|---|---|---|---|
| 2 | 7.95 | 2 | 4.32 | 8.78 | 9.63 | 2.02 | 8.46 |
| 4 | 3.97 | 1 | 4.22 | 14.18 | 13.4 | 1.86 | 7.16 |
| 8 | 2 | 1 | 7.13 | 9.92 | 12.96 | 6.3 | 14.2 |

**Figure 20:** Performance metrics for varied number of workers in dask cluster

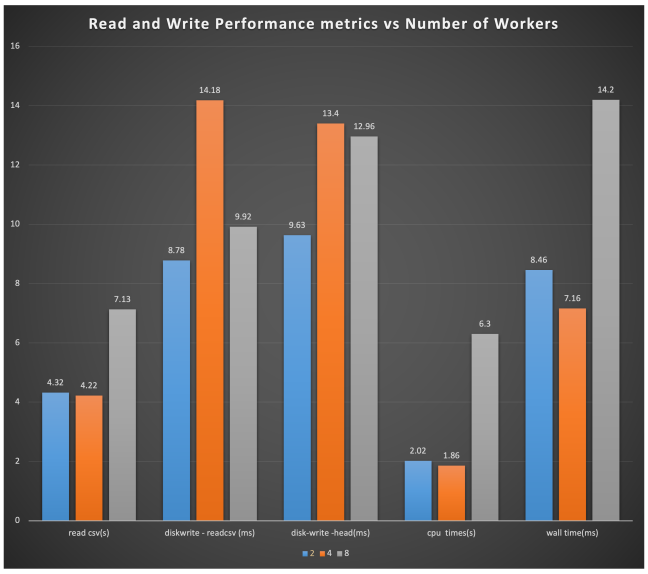


**Figure 22:** Read and Write performance comparisons

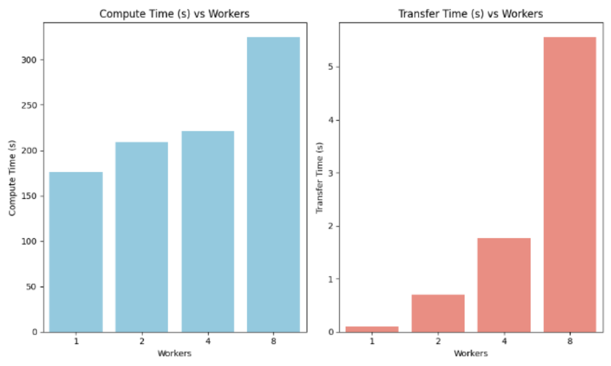


**Figure 23:** Compute and Transfer times vs Number of workers

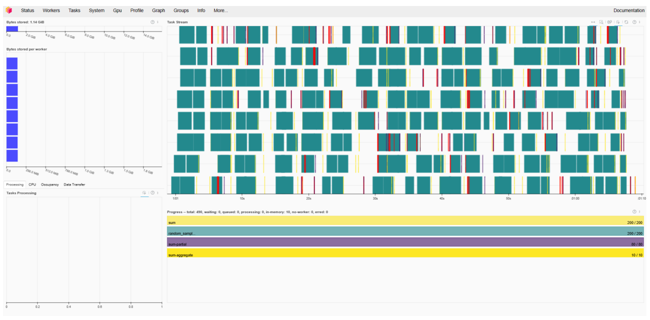


**Figure 24:** Task Stream plot - 8 workers

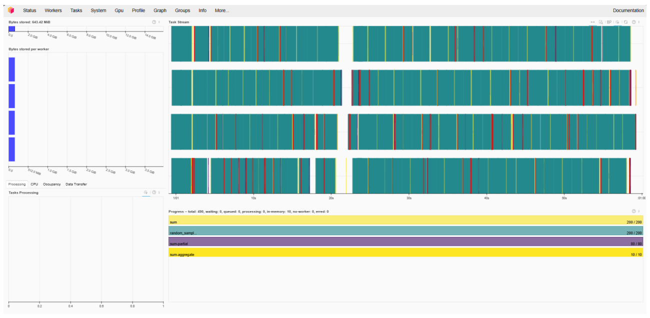


**Figure 25:** Task Steam plot - 4 workers

tasks represent idle periods without computations, thereby increasing overall computational time. Conversely, when the worker count was 2 or 4, fewer white spaces were observed, suggesting more efficient task computation and scheduling within Dask.

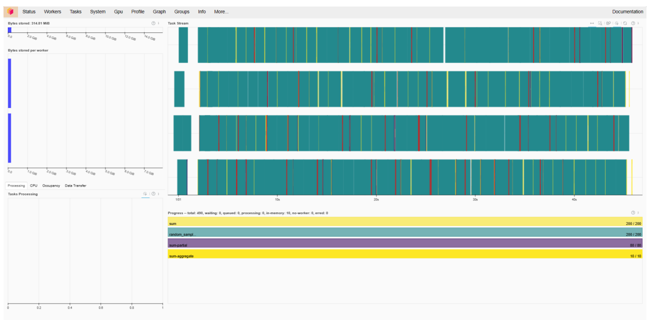


**Figure 26:** Task steam plot - 2 workers

## 5. Conclusion

In conclusion, our exploration into the Cloud Platforms and distributed computing capabilities of Dask has revealed its remarkable efficiency and scalability in handling large datasets and complex computations. Leveraging AWS, we experienced the ease and cost-effectiveness of spinning up clusters, providing the necessary computing power at a fraction of the cost. Through various experiments and analyses, we observed that Dask outperforms traditional tools such as Pandas in terms of processing speed, particularly with sizable datasets exceeding available memory capacity. The parallelization capabilities of Dask enable efficient distribution of tasks across multiple cores or machines, resulting in significantly shorter processing times and improved system throughput.

Moreover, our investigation into different worker node configurations highlighted the importance of resource allocation, concurrency, and fault tolerance in maximizing performance and scalability in distributed computing environments. The observed trends in computation times, convergence rates, and task distribution provide valuable insights into the factors influencing the efficiency and scalability. Looking ahead, continued research and experimentation in this field will be essential for advancing our understanding of distributed computing frameworks like Dask and leveraging their full potential to address complex data analysis and processing tasks across various domains.

Please find the project GitHub repository at this link.

## 6. Future Work

Several avenues can be explored to further enhance this research experiment:

1. **Benchmarking with Larger Datasets**: Conducting benchmarking experiments with datasets exceeding 50 GB will provide insights into Dask's performance and scalability with extremely large datasets.

2. **Comparative Analysis with Other Frameworks**: Extending the comparison to include other distributed computing frameworks such as Pyspark and RAY will offer a comprehensive understanding of their respective strengths and weaknesses. This comparative analysis will enable researchers to make informed decisions when selecting the most suitable framework for specific use cases.

3. **Programmatic Deployment on Cloud Platforms**: Implementing programmatic deployment of Dask clusters on AWS and scaling up to build larger clusters will facilitate the evaluation of Dask's performance in more complex and demanding environments. This will involve automating the setup and configuration of Dask clusters, optimizing resource allocation, and monitoring cluster performance.

4. **Exploration of Advanced Features**: Investigating advanced features and functionalities of Dask, such as dynamic task scheduling, adaptive scaling, and fault tolerance mechanisms, will further enhance our understanding

of its capabilities and potential applications. This will involve experimenting with different configuration settings and parameters to optimize performance and scalability.

By addressing these future research directions, we can deepen our understanding of Dask's capabilities and contribute to the advancement of distributed computing technologies for handling big data and complex computational tasks effectively. .

## References

[1] Baglioni, Michele, Fabrizio Montecchiani, Mario Rosati et al. 2023. Large-scale computing frameworks: Experiments and guidelines. En *CEUR WORKSHOP PROCEEDINGS*, vol. 3606, CEUR-WS

[2] Chen, Junda, Aditya Kumar Akash & Yukiko Suzuki. 2021. Explore optimal degree of parallelism for distributed xgboost training. Relatório técnico.

[3] Chen, Tianqi & Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. En *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794

[4] Dean, Jeffrey & Sanjay Ghemawat. 2008. Mapreduce: simplified data processing on large clusters. *Communications of the ACM* 51(1). 107–113

[5] Dugré, Mathieu, Valérie Hayot-Sasson & Tristan Glatard. 2019. A performance comparison of dask and apache spark for data-intensive neuroimaging pipelines. En *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 40–49. IEEE

[6] Dünner, Celestine, Thomas Parnell, Kubilay Atasu, Manolis Sifalakis & Haralampos Pozidis. 2017. Understanding and optimizing the performance of distributed machine learning applications on apache spark. En *2017 IEEE international conference on big data (big data)*, 331–338. IEEE

[7] Gonzalez, Joseph E., Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin & Ion Stoica. 2014. Graphx: graph processing in a distributed dataflow framework. En *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* OSDI'14, 599–613. USA: USENIX Association

[8] Masih, Zoya. 2023. Analyzing io performance when using dask-ml

[9] Meng, Xiangrui, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen et al. 2016. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research* 17(34). 1–7

[10] Rocklin, Matthew et al. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. En *SciPy*, 126–132

[11] Vatter, Jana, Ruben Mayer & Hans-Arno Jacobsen. 2023. The evolution of distributed systems for graph neural networks and their origin in graph processing and deep learning: A survey. *ACM Computing Surveys* 56(1). 1–37

[12] Watson, Alex, Deepigha Shree Vittal Babu & Suprio Ray. 2017. Sanzu: A data science benchmark. En *2017 IEEE International Conference on Big Data (Big Data)*, 263–272. IEEE

[13] Zevnik, Filip, Din Music, Carolina Fortuna & Gregor Cerar. ???? Scikit learn vs dask vs apache spark benchmarking on the eminst dataset