Integrating Cerebro with Dask

Vignesh Nanda Kumar[*]
UC San Diego
La Jolla, CA, USA
vnandakumar@ucsd.edu

Pratik Ratadiya[*]
UC San Diego
La Jolla, CA, USA
pratadiya@ucsd.edu

ABSTRACT

Cerebro is a distributed model selection system that supports a novel parallel SGD execution strategy called Model Hopper Parallelism (MOP). Cerebro has been implemented in a modular way combining model and data parallelism such that the support can be extended for different backend execution layers. This project extends the support for Cerebro to Dask users. Dask is a parallel computing framework built in Python that natively supports embarrassingly task-parallel workloads. Our work manages to support scalable model selection using MOP in Dask without modifying any internal Dask implementations. We use Dask for the initial Extract Transform Load (ETL), data partitioning, and orchestration of the MOP strategy. The system is evaluated by training multiple model configurations on a standard dataset using MOP. We observe that the Dask backend can reproduce the same results as the original implementation in terms of both system performance and accuracy.

KEYWORDS

Distributed Computing, Dask, Scalable Model Selection, Model Hopper Parallelism

1 INTRODUCTION

Scalable deep learning has always been a challenging task, especially when maximizing system utilization and performance. Implementing model selection strategies at such a large scale becomes even more difficult. Previous solutions to this problem-focused either on data-parallelism or task-parallelism, leading to a trade-off. To this end, Cerebro[27] was proposed for model selection in large-scale deep learning by combining the advantages of data-based and task-based parallelism approaches. It employs a novel Model Hopper Parallelism (MOP) approach for performing parallel training of multiple model configurations.

Data-parallelism focuses on sharding data across different machines. However, it only trains one model at a time and does not look into model selection workloads. On the other hand, task-parallel approaches train multiple models parallelly but expect the entire dataset to be present on each worker, thus introducing extra cost for carrying out the training process. Data parallel approaches suffer from poor per-epoch efficiency and slower convergence. Task-parallel approaches suffer from no data scalability and poor memory efficiency. Cerebro's MOP strategy tries to address these challenges.

MOP combines the advantages of data and task parallel approaches. The training data is sharded across all the workers. Each model configuration is trained once on each worker in random order. While one model is training on a worker, the other workers can be utilized to train other models. Once the training is completed, the model config "hops" on to the next idle worker on which it has not trained previously. One training epoch is completed when all the model configs have been trained on all the workers. MOP benefits from the randomness order that is allowed by SGD-based optimization[29] training in neural networks. Thus the MOP strategy ensures high model throughput, high data scalability, low communication cost, and no storage wastage. The authors of Cerebro[28] implemented Cerebro using Spark[30] in backend.

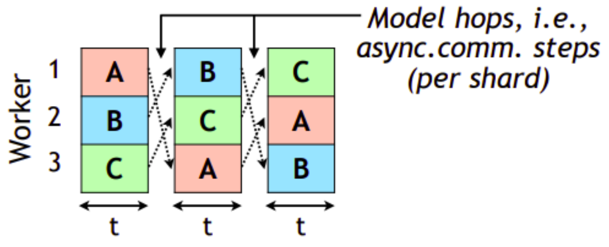


Figure 1: Model Hopper Parallelism

Dask[10] is a Python-based programming library that enables distributed computing. It supports task-parallel workloads through dynamic scheduling. Dask schedulers integrate well with a wide range of custom workloads. It also provides a wide range of data collections[25] including arrays and dataframes[12], that supports parallel programming while also being built upon existing Python interfaces including Numpy and Pandas[16]. This makes Dask a good backend alternative for the current Spark backend being used by Cerebro.

Through this project, we have integrated Cerebro with Dask. Specifically, we build a new backend completely using Dask to implement the MOP technique for parallel training of multiple model configurations. This integration will help for the wider adoption of Cerebro and ameliorate any issues observed when using Spark. The Dask backend is put to test by training multiple model configurations at once on the Criteo[1] dataset. The system consumption and results are compared with the original Spark backend. Specifically, our main contributions through this project are as follows:

(1) Developed a new backend for Cerebro using Dask that completely removes the Spark dependency while still implementing MOP.
(2) Implemented and showed the parallelism achieved by MOP using the Dask backend in a distributed cluster environment.
(3) Evaluated the system performance on the Criteo dataset and performed an extensive analysis of results and comparison with the Spark backend.

[*]Both authors contributed equally to this research.

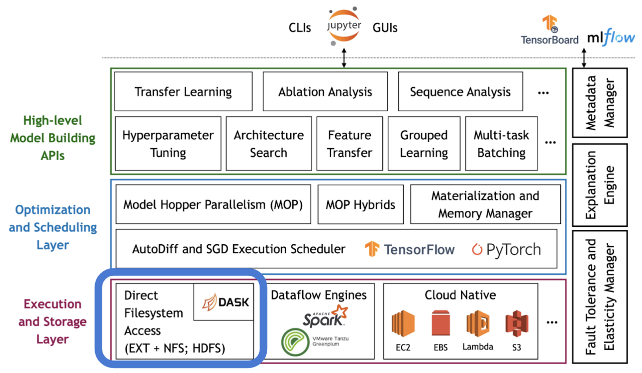


Figure 2: Cerebro architecture

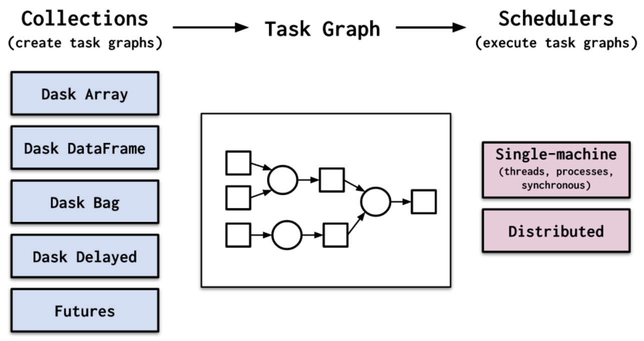


Figure 3: Dask Execution

The rest of this report is structured as follows: Section 2 talks in detail about the current implementation of Cerebro and the functionality provided by Dask. We formally define our problem in Section 3. The implementation details are explained in Section 4. The system evaluation and results are presented in Section 5 while the analysis and key takeaways are presented in Section 6. Finally, we present the future scope and conclusion in Section 7 and 8 respectively.

2 BACKGROUND WORK

2.1 Current Implementation of Cerebro

The end-to-end Cerebro architecture broadly consists of three main layers. These are model building APIs, optimization and scheduling, and execution and storage layer. The modules in each of these layers have been depicted in Figure 2.

The high-level model building APIs consist of various functions related to the training of models, including transfer learning, hyperparameter tuning, architecture search, and multi-task learning. These are accompanied by utilities including ablation analysis, sequence analysis, feature transfer, and other related features. This high-level layer is exposed to Jupyter notebooks and other application-level functions. It should be noted here that not all of these functionalities are currently available in the Cerebro architecture, and this is rather the vision of the Cerebro creators.

The intermediate layer consists of the MOP functionality and its associated hybrids. Various schedulers required for AutoDiff and SGD execution would be present here. Other managers required for materialization and memory management will also find a place in this layer.

The lowermost layer is the execution and storage layer that consists of the core backend required to run the Cerebro architecture. This includes direct file system access and required engines like Spark[30] and Dask[10]. In the future, this layer would also be able to support other cloud-native backends like Lambda, S3, AWS. This execution and storage layer will work in tandem with the intermediate layer to achieve fault tolerance and provide elasticity.

The current implementation of Cerebro uses Spark as its backend. This backend is exposed to the other layers using a set of APIs, including data sharding, worker communication, epoch training, validation, and data analysis. Such a modularized approach also makes it convenient for the end-user to continue using the functionalities without worrying about the internal processes running in the lower layers.

2.2 Dask: Framework overview

The three main utilities of Dask include collections, task graphs, and schedulers. Their purpose is shown in Figure 3. Dask provides a set of collections including array[3], dataframes[12], delayed instances[13], and futures[18]. These collections can be used to create a task graph[23]. Task graph consists of the sequence between each of the processes being executed parallelly. This task graph is operated upon and executed by Dask schedulers[22]. Dask schedulers can operate in both single-machine and distributed cluster environments.

2.3 Implementing parallel programming in Dask

For integrating Dask with Cerebro, we went through different distributed ML algorithms which have been integrated with Dask to compare previous approaches. First, we checked the XGBoost algorithm[17]. XGBoost processes the Dask collections to create a DMatrix, a data structure used by XGBoost for storing data. XGBoost handles the distributed training using the DMatrix and the Dask scheduler. The XGBoost Dask interface gives a good starting point for understanding the Dask interface, although the docs were not clear in explanations. Another framework for performing distributed ML is Dask ML[9]. The user needs a fair amount of understanding of Scikit-Learn APIs to use the scalable ML algorithms support provided by Dask ML. Dask ML not only increases the complexity of the code by increasing the number of dependencies, but also requires the user to be familiar with all these multiple libraries. Working with Dask ML would also require changes to the Dask codebase itself to natively support Cerebro. All these challenges help us to formulate our problem better, described in the next section.

3 PROBLEM DESCRIPTION

The task at hand is to create a new backend for Cerebro built using Dask. The main challenge is to implement MOP while respecting the constraints of Dask.

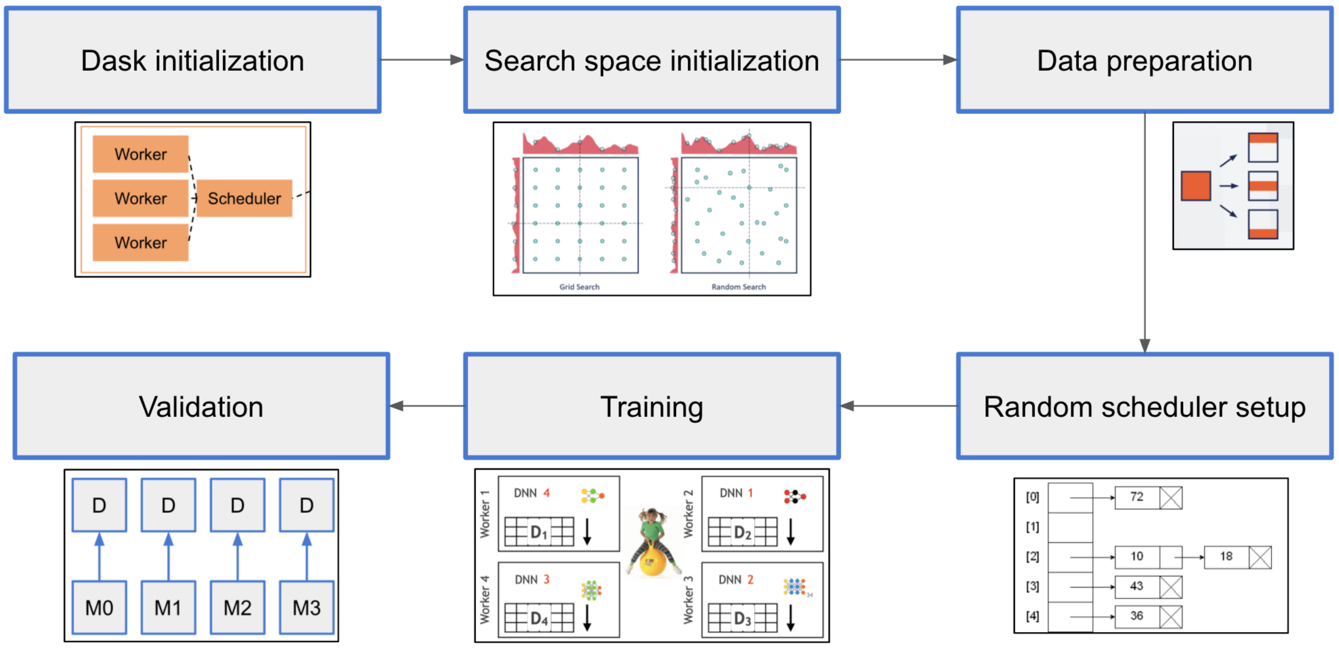


Figure 4: Cerebro Dask Overview

3.1 Problem Motivation

Cerebro makes use of Spark[30] in its backend. Spark has several limitations, including a steep learning curve, a large set of dependencies, and poor support for streaming. Further, the addition of new backends will help to improve the flexibility and wider adoption of Cerebro. A Dask backend can also make the functionality of Cerebro readily available to its rapidly growing community. All these factors motivate us to pursue this project.

3.2 Objectives

As shown in Figure 2, our focus is on the execution and storage layer of Cerebro. The objectives of our project are as follows:

- To implement the model hopper parallelism (MOP) technique using Dask APIs.
- To demonstrate the successful working of MOP in both single-node and distributed cluster environments.
- To perform benchmarking and reproducibility checks against the previously obtained results by the Spark backend in the original Cerebro implementation.

We now explain the implementation details of our proposed system in detail in the next section.

4 IMPLEMENTATION DETAILS

4.1 Overview

For our implementation, we decided to implement the functionalities provided by Cerebro-Spark API using vanilla Dask methods. We choose to use the same function names and endpoints to enable easier switching between different backend for users. They will not have to worry about different endpoint names or conventions. We also keep Dask and Cerebro loosely coupled in the sense that we can independently initialize the Dask scheduler and workers which can be then be called from the Cerebro backend for further processing.

Since Dask natively supports task parallelism, we cannot directly use Dask task graphs[23] for submitting our model training tasks. This approach will not work because a pre-computed task graph cannot predict idle workers. Worker status can change from idle to training (or vice versa) in real-time. Hence we used the approach used in Cerebro-Spark to schedule the training task on workers as soon as we find an idle model-worker pair.

This section describes the system design of integrating Dask with Cerebro. We go through all the components of the pipeline as illustrated in Figure 4. We first start with setting up the Dask client and then move to the data preparation for training and validation data. After that, we describe the data structures used in the random scheduler and the training process. Then we describe the model validation process. Finally, we describe the additional considerations for running the setup on a distributed cluster.

4.2 Dask client

To set up Dask, we need three main modules: Dask scheduler[22], Dask workers[24] and Dask client[5]. The complete Dask client setup and process is illustrated in Figure 5.

- **Dask scheduler**[22]: To initialize a Dask scheduler, we use the command-line interface. The scheduler gets initialized on port 8786 by default, but if required, we can assign it to a different port. Dask scheduler handles the assignment of tasks to different Dask workers.

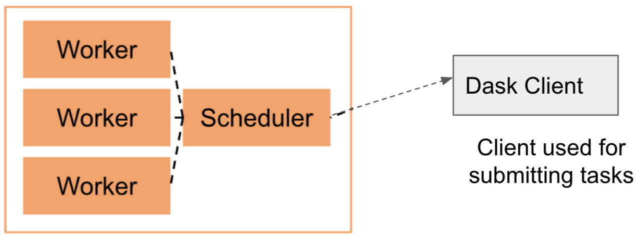


Figure 5: Dask Client Setup

- **Dask workers**[24]: After initializing the scheduler, we can now spawn workers. To spawn workers from the command line we need to pass the scheduler IP to the command so that the Dask worker can be registered with the corresponding scheduler. We initialize one Dask worker on each cluster node based on a distributed setting. Each Dask worker stores its local data in its respective space folder. So, in a shared network file system, it has to be ensured that workers have their separate worker spaces to avoid using locks.
- **Dask client**[5]: Dask client is the interface between the Dask API and the Dask scheduler. After the scheduler and workers are set up, the Dask client is initialized by passing in the Dask scheduler address.

4.3 Data preparation

We are given the training and validation data that needs to be sent to Dask workers. The scattering of training and validation data is handled differently. For partitioning the training data, the Dask client's scatter[11] method is used, which works on Dask collections to send a dataset to a particular worker. We can provide a list of worker IPs to send the data. In our case, since each partition has to be sent to only one worker, we give only one worker IP for a partition. We get back the Dask future object[18] pointing to the Dask dataframe partition. This future is used during training to assign a particular model training to a data partition. Such partitioning ensures that the complete training data is never loaded on the client. Even the first data load occurs in a delayed manner. Dask loads the data in chunks[4] and distributed manner. If required, the dataset can be chunked back so that the chunks can fit in memory.

In case of the validation dataset, the complete dataset is copied to each worker's memory. There are two main reasons to do this: First is that model evaluation will become an easy task-parallel task. We do not need to worry about how to aggregate model evaluation statistics. This is because one model will be evaluated on one worker on the complete validation set in a single go. Secondly, validation data is usually much smaller in size than the training data and hence can fit in memory, and hence can be broadcasted to all the workers.

4.4 Random Schdeduler

A random scheduler is used in the Cerebro system to schedule model training on different workers. SGD is robust to randomness, so model training on workers in a random order will not affect the model performance. To support random scheduling, we have used the following data structures.

- $Models[m1, m2, ...m]$: Randomly shuffled list of model configurations. The configurations are shuffled in every epoch.
- $Workers[w1, w2...w]$: List of worker IP addresses. This is required to send a model training task to a particular worker.
- $Trained[m][w]$: A 2D boolean matrix detailing if a model m has been trained on worker w. This will be used to check if a model has been trained on all workers, and also to find an idle model to train on a worker.
- $Mapping\{m- > (w, < Daskfuture f >)\}$: A mapping from a model to the corresponding worker on which the model is training. Along with the worker, a Dask future object is also stored. Whenever a model training task is submitted to a worker, a Dask future[18] is saved corresponding to the train function call. This future is used to check if a model training is complete on a worker.

4.5 Training

Algorithm 1 presents the pseudocode for training one epoch. The data structures described in the previous subsection (4.4) are all input to the training process. The function runs until all models have been trained on all workers once. The function iterates over all the workers to find an idle worker. If an idle worker is found, we search for a runnable model i.e. a model currently idle and also not trained on the idle worker. Once the idle model worker pair is found, Dask client's submit[5] method is called to train the model on that worker. The training algorithm is implemented such that all the four invariants of MOP are taken care of (completeness, model training isolation, worker/partition exclusive access, and non-preemptive execution).

If a worker is not idle, that would mean that some model is training on the worker. Hence its training status is checked using the Dask future object. If the training is complete, the training matrix is updated for the model worker pair, and the model and worker are set to idle. If the model is trained on all workers, it is removed from the set of models to be trained.

4.6 Validation

Model validation is done using task parallelism. All the workers have the complete validation dataset. So, each model is sent to one worker selected at random to perform model validation. Once all the models are validated, we can return the best-performing model. We preferred task parallelism over data parallelism because, with data parallelism, we would have had to handle the aggregation of validation statistics for different evaluation metrics, which is not maintainable. Using task parallelism ensures that the deep learning library handles validation.

4.7 Distributed Execution

For supporting distributed execution, two main things have to be supported: a shared file system and a distributed cluster. Other than these changes, no other changes were required to the Dask setup described before.

Algorithm 1 Training for 1 epoch

```
Input: List of model configs $M$, List of worker IPs $W$, Training status $TS$
Output: Trained models $T$
  while $len(models\_to\_train) > 0$ do
      for each $w$ in W do
          if $w$ is idle then
              $m \leftarrow$ Idle model not trained on $w$
              $w.future$ = client.submit($w$,$m$,train_model()) x
              $Mapping[m]$ = {w, w.future}
          else
              if $w.future$ == "finished" then
                  $TS[w][m]$ = True
                  update($T(m)$)
                  Set $m$ and $w$ as idle
                  Remove $m$ from $models\_to\_train$ if trained on all workers
              end if
          end if
      end for
  end while
  return $T$
```

Firstly, we had to set up a Network File System. This was done by setting up passwordless ssh on the cluster nodes. Then the network file system was set up using sshfs[19]. Using a network file system has two advantages:

- **Share data**: The training and validation data can be placed in the NFS. This ensures that workers can easily read data from the shared file system. This also avoids copying data repeatedly to each worker.
- **Reduce communication cost**: The setup also ensures minimal communication between the client and workers. The Dask client communicates with the Dask workers to assign a train sub-epoch task. Each worker has its own partitioned local training data loaded in its memory. First, we thought that the client could send the model every time to a worker, and the worker can return the updated model to the client. However, this can lead to the client becoming a bottleneck. So, instead of the client communicating the models, we set up a Network File System from where each worker can read and update models, thus avoiding communication between client and workers.

Secondly, to execute Dask-Cerebro on a distributed cluster, we had to set up a Dask cluster[7]. Dask supports two types of schedulers single machine and distributed scheduler. The single machine scheduler initializes all Dask workers on a local process or thread pool. However, the issue with a single machine scheduler is that it is limited by a machine's resources and does not scale. The second type is a distributed scheduler which is required for our task. There are different ways of setting up a distributed Dask cluster[14]. Dask cluster can be set up using the command-line interface to set up the Dask-scheduler and Dask-worker processes. Dask cluster can also be set up using python API using SSHCluster[14]. But the issue with Python API is that it does not provide support for changing the location of local Dask worker space. The Dask worker space defaults to the home directory. This can be an issue during experimentation because the home directory has very limited space. Hence, a command-line interface is used to set up the Dask clusters.

Table 1: Workload configuration for training the system on the Criteo dataset

Parameter	Values
Model arch.	3 layer, 1000-500-1
Model size	90 MB
Batch size	{32, 64, 256, 512}
Learning rate	{1e-3, 1e-4}
Regularization	{1e-4, 1e-5}
Epochs	5

5 EVALUATION AND RESULTS

5.1 Experimental Setup

We set up a 9 node distributed cluster environment on Cloudlab[1] [26]. These nodes include a scheduler node and 8 worker nodes. Each of these nodes was configured to have 256 GB RAM and 2.2 GHz Intel E5 10-core CPUs. Approximately 200 GB of storage space was allotted to each of the workers. Tensorflow v2.3[20] was used for model building.

We evaluate the architecture by training a neural network architecture on the Criteo[1] dataset. The Criteo dataset is a binary classification task to predict whether a particular ad was clicked through or not. We use a subset of the original dataset, similar to the one used to evaluate the original Cerebro implementation. This subset consists of over 100 million samples, each containing a mix of categorical and numerical features. We have 7306 input columns per sample post the one-hot encoding and data densification. We load the dataset in Parquet[2] format.

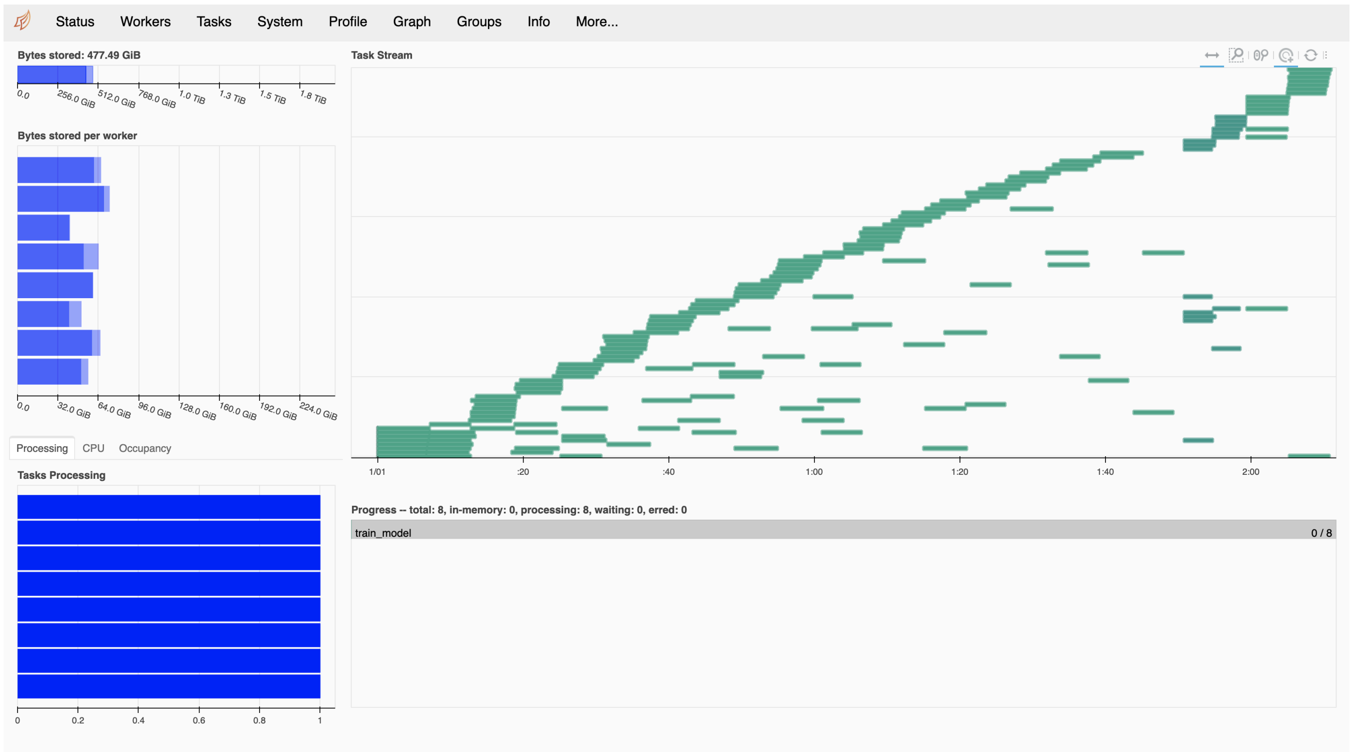
We use a simple three-layer neural network for training purposes. It consists of 1000 and 500 nodes in its hidden layers and one output classifier node. 16 different model configurations are generated by varying the three hyperparameters: learning rate, lambda, and batch size. Each of these model configurations was trained for 5 epochs. The workload architecture is defined in Table 1.

5.2 Dask Diagnostics: Overview

Dask provides an in-built diagnostics dashboard[15] which constantly monitors an active system and provides information related to the same. The main monitoring insights provided by the Dask dashboard are as follows:

- **Task Stream**: A real-time graph indicating the active tasks and their duration on each of the workers. It also shows the progress of each task including their status wiz. one of in-memory, processing, waiting, or failed.
- **Bytes stored**: The dashboard provides information about the total bytes stored in memory currently, including the bytes stored per worker. Further statistics are provided about the worker's active and unmanaged memory.

[1]https://www.cloudlab.us/



Progress -- total: 8, in-memory: 0, processing: 8, waiting: 0, erred: 0

Figure 6: Sample image of the Dask dashboard

- **Worker information:** Detailed information about each worker is provided including the currently used memory, active processes, number of threads being executed, and its associated logs.
- **Graphs:** Information related to any sub-processes that are being executed within a process is provided, including its consumption and execution times.
- **Other logs:** Other logs related to the occupancy of each task, their profiles, and grouping is provided. The user also can download some of this information, including stream graphs, logs, and system consumption.

A screenshot of the Dask dashboard during a sample run has been shown in Figure 6.

5.3 Results

One of the main objectives of our work was to benchmark the obtained results and perform reproducibility checks against the original implementation. To this end, we compare our results with the results obtained by the Spark backend[31] used in the original Cerebro implementation. Specifically, results are compared for the runtimes and the system utilization during the end-to-end tests. This comparison has been shown in Table 2.

It can be observed that we have managed to replicate the results obtained using the Spark backend, in fact even outperforming it across certain metrics. The CPU and DRAM utilization and Disk write operations are almost in line with the original implementation. The Dask backend performs slightly more disk reads per worker,

Table 2: Comparison of system utilization with Spark backend

Metric	Cerebro-Dask (Proposed)	Cerebro-Spark (Original)
CPU Utilization (%)	**36.9**	35.2
DRAM Utilization (%)	23.48	**28.50**
Per worker disk read (GB)	0.55	**0.2**
Per worker disk write (GB)	**1.02**	**1**
Network traffic (TB)	**0.03**	0.2
Runtime (hrs.)	**10.2**	22.5

which could be attributed to the reads performed for maintaining custom logs and monitoring them. We observe a tremendous reduction in network traffic as NFS has been used, leading to minimum sharing between workers and scheduler as both of them read directly from the NFS. Further, memory has been persisted hence not requiring any further exchange. We observe almost a 2x speedup in terms of the runtime of the entire experiment. This could be attributed to the optimizations in subsequent versions of Tensorflow[20] post the original implementation and the advantages provided by the internal optimizations of Dask.

To show the parallelism in Dask, we also show the Gantt chart indicating the process sequence for training the 16 different model configurations across the eight workers. This chart has been visualized in Figure 7. It can be seen that MOP has been implemented

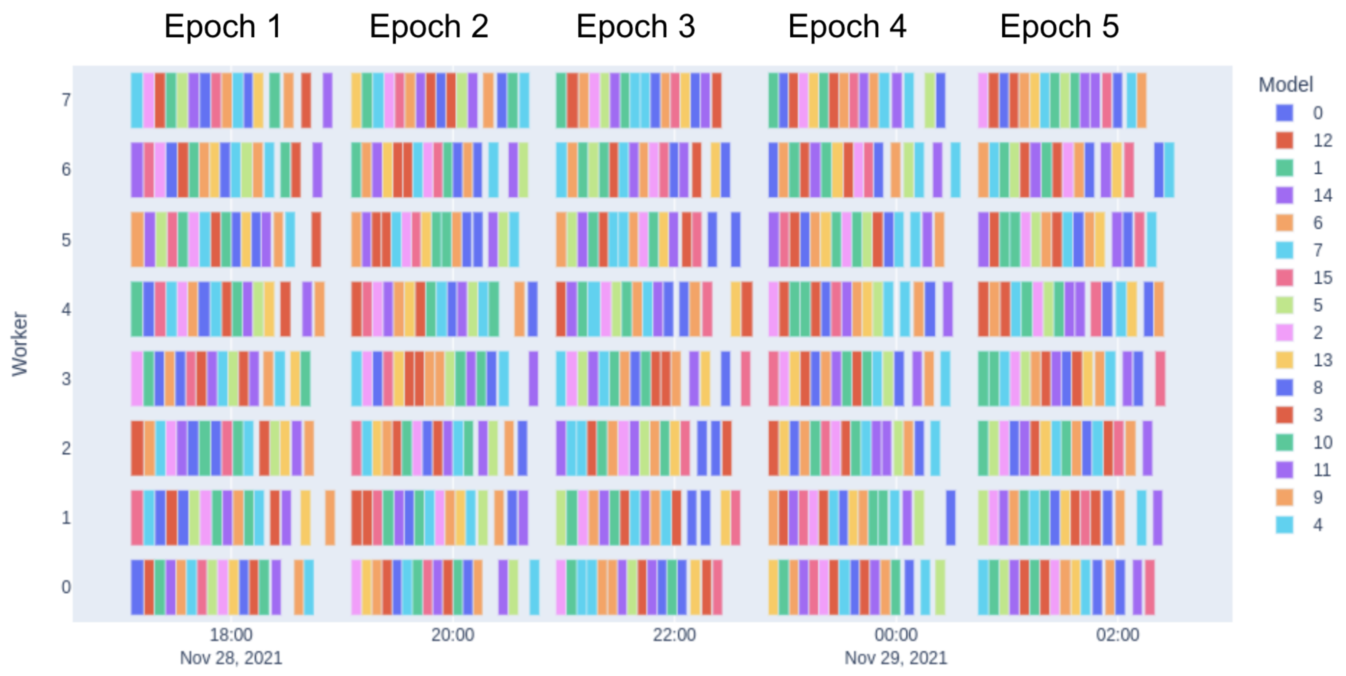


Figure 7: Gantt chart of the Criteo training process (5 epochs)

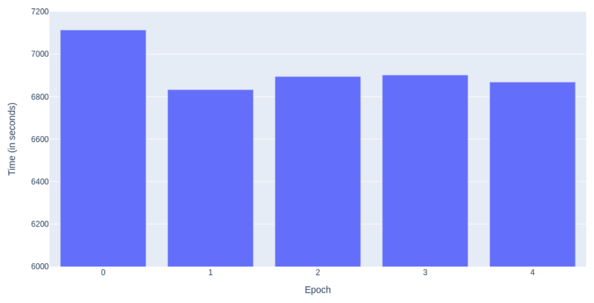


Figure 8: Per epoch training times

successfully with the level of parallelization achieved. The intermediate gaps at intervals are due to the validation process running between the epochs to note down the evaluation results.

Finally, we also analyze the per epoch training times in Figure 8. It can be seen that the first epoch requires a higher time than the other epochs. This can be attributed to the fact that each worker is loading their respective data shards for the first time in local memory. However, as this memory is persisted for subsequent epochs, the training times decrease and remain consistent later.

6 ANALYSIS OF SYSTEM ARCHITECTURE

We analyze the system across five main characteristics. These are namely: usability, manageability, efficiency, scalability, and developability [6][8]. For a disinterested view, we note down both the advantages and disadvantages of the proposed architecture across each of these characteristics:

(1) **Usability**: Our observation was that while Dask schedulers are easy to use, ETL operations are difficult to perform. The best use of the framework would be by trying to perform as much data cleaning and processing before feeding the data into the pipeline, to benefit from the scheduling abilities of Dask. A core advantage of Dask is the easy integration of the Dask scheduler with custom workloads. The disadvantage is every dataset has to be converted to a Dask collection (array/dataframe) before partitioning the dataset across workers. This adds additional overhead.

(2) **Manageability**: The built-in dashboard provided by Dask provides a rich set of diagnostics including process charts, memory consumption, logs, and per worker statistics. The dashboard can be run locally without any external dependencies. A core disadvantage of this dashboard is that it does not detect any background fault occurrence and hence keeps running without any notice to the user.

(3) **Efficiency**: We observed that the efficiency of the Dask backend is comparable to the Spark backend. In fact for some parameters like network traffic and memory consumption, it even outperformed the Spark backend. This could be attributed to changes like optimization in the latest versions of the Tensorflow[20] library and the internal optimizations of the Dask framework. However, we consider this analysis to be a good analysis problem in itself and would consider it to be a good future scope for extension.

On the negative front, the Dask backend seems to be performing more memory write operations as compared to its Spark counterpart.

(4) **Scalability**: It is fairly easy to scale the Dask backend to multiple workers. All of them would require the Dask distributed framework and required libraries to be installed, and the scheduler would need access to their IP addresses. Maintaining consistent versions and updating all the workers to have the same setup would require some extra consideration.

(5) **Developability**: Dask is still a developing framework and hence has its own set of bugs and limitations. There is a learning curve involved with getting acquainted with playing around with Dask. However, once a user gets a fair hold on the concepts and is offered Dask utilities, Dask has a broad range of services to offer for enabling distributed computing across a broad range of applications.

7 SCOPE FOR IMPROVEMENT

- **Improving ETL operations:** The *xarray* utility can be used for better supporting parallel and streaming computations in the case of Dask. Xarray provides a rich set of functions for ETL making it a better choice as compared to the standard utilities in the Dask library. Also, in general, the ETL can be handled in a more streamlined manner and not be Dask collection dependent.
- **Achieving large-scale data sharding in memory:** We faced considerable difficulties in reading large-sized parquet files. While this could be attributed to our system limitations to some degree, Spark had a better tolerance and debugging mechanism for the same, and a similar one is desired for the Dask backend.
- **Scripts for automating setup and model runs:** Currently, we had to manually execute the same script on each of the workers, including installation of libraries, setting up of their SSH configs, and clearing and maintaining directories. This was a fairly monotonous and trivial activity and could be automated[21] for better control of the network right from the scheduler or the client.
- **Fault tolerance and elasticity:** As mentioned previously, the Dask backend currently does not have the best support for fault tolerance. However, this should be easy to implement as checkpointing has been maintained. We would just need to maintain a log of the sequence in which each of the models was trained, to be able to fetch their latest complete train in case of any node failure. Some work would be needed to address elasticity in the network. This includes the ability to include a new node while training is online and making it a part of the training process in between the epochs. An additional loop can be used to keep a watch on all the active workers currently a part of the network. We leave this implementation for future work.
- **Open source integration:** With the objectives focusing on implementation of MOP and result benchmarking, we did not implement all the other Cerebro services that were provided in the Spark backend. These include advanced data preparation modules, metadata analysis, column breakdown, etc. As we integrate our Dask backend into the open-source implementation of Cerebro, we will need to include these supporting blocks in our implementation.

8 CONCLUSION

In conclusion, through this project, we have managed to extend Cerebro to support a new Dask backend. We were able to replicate the working of the model hopper parallelism approach in both single and distributed-machine environments, without modifying any existing implementations of Dask. Using the Dask scheduler and distributed computing utilities, we demonstrated the working of MOP for training multiple model configurations on the standard Criteo dataset. The obtained results and system consumption patterns of our backend were similar to the original Spark backend implementation, hence proving the consistency. The addition of this backend would help to make the adoption of Cerebro easier, especially with organizations that currently use Dask in their backend pipelines.

ACKNOWLEDGMENTS

We would like to thank Professor Arun Kumar for his valuable inputs and feedback throughout this project. We would also like to thank Yuhao Zhang and Supun Nakandala for their assistance in setting up the original Cerebro backend and for helping us out with the data processing and loading modules. Finally, we extend our sincere gratitude to *Cloudlab* for providing us with the environment in which these experiments were performed.

REFERENCES

[1] 2014. Kaggle contest dataset is now available for academic use! https://ailab.criteo.com/kaggle-contest-dataset-now-available-academic-use/

[2] 2021. Apache Parquet. https://parquet.apache.org/

[3] 2021. *Array — Dask documentation*. https://docs.dask.org/en/stable/array.html

[4] 2021. Chunks — Dask documentation. https://docs.dask.org/en/stable/array-chunks.html

[5] 2021. Client — Dask.distributed 2021.11.2 documentation. http://distributed.dask.org/en/stable/client.html

[6] 2021. Comparison to Spark — Dask documentation. https://docs.dask.org/en/latest/spark.html

[7] 2021. Configuring a Distributed Dask Cluster. https://blog.dask.org/2020/07/30/beginners-config

[8] 2021. CSE 234: Data Systems for Machine Learning UCSD. https://cseweb.ucsd.edu/classes/fa21/cse234-a/index.html

[9] 2021. Dask-ML — dask-ml 2021.11.17 documentation. https://ml.dask.org/

[10] 2021. Dask — Dask documentation. https://docs.dask.org/en/latest/

[11] 2021. Data Locality — Dask.distributed 2021.11.2+24.g0d6407e5 documentation. http://distributed.dask.org/en/latest/locality.html

[12] 2021. DataFrame — Dask documentation. https://docs.dask.org/en/stable/dataframe.html

[13] 2021. Delayed — Dask documentation. https://docs.dask.org/en/stable/delayed.html

[14] 2021. Deploy Dask clusters — Dask documentation. https://docs.dask.org/en/latest/how-to/deploy-dask-clusters.html

[15] 2021. Diagnostics (distributed) — Dask documentation. https://docs.dask.org/en/latest/diagnostics-distributed.html

[16] 2021. Distributed Pandas on a Cluster with Dask DataFrames. https://matthewrocklin.com/blog/work/2017/01/12/dask-dataframes

[17] 2021. Distributed XGBoost with Dask — xgboost 1.6.0-dev documentation. https://xgboost.readthedocs.io/en/latest/tutorials/dask.html

[18] 2021. Futures — Dask documentation. https://docs.dask.org/en/stable/futures.html

[19] 2021. How To Use SSHFS to Mount Remote File Systems Over SSH. https://www.digitalocean.com/community/tutorials/how-to-use-sshfs-to-mount-remote-file-systems-over-ssh

[20] 2021. Module: tf | TensorFlow Core v2.3.0. https://www.tensorflow.org/versions/r2.3/api_docs/python/tf
[21] 2021. Python API (advanced) — Dask documentation. https://docs.dask.org/en/latest/how-to/deploy-dask/python-advanced.html
[22] 2021. Scheduling — Dask documentation. https://docs.dask.org/en/latest/scheduling.html
[23] 2021. Task Graphs — Dask documentation. https://docs.dask.org/en/latest/graphs.html
[24] 2021. Worker — Dask.distributed 2021.11.2+24.g0d6407e5 documentation. http://distributed.dask.org/en/latest/worker.html
[25] 2021. Working with Collections — Dask documentation. https://docs.dask.org/en/stable/delayed-collections.html
[26] Dmitry Duplyakin et al. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19
[27] Arun Kumar, Supun Nakandala, Yuhao Zhang, Side Li, Advitya Gemawat, and Kabir Nagrecha. 2021. Cerebro: A Layered Data Platform for Scalable Deep Learning.. In *CIDR*.
[28] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.
[29] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.
[31] Yuhao Zhang, Frank McQuillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. 2021. Distributed deep learning on data systems: a comparative analysis of approaches. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1769–1782.