Explore Optimal Degree of Parallelism for Distributed XGBoost Training

Abstract
The XGBoost has been an extremely popular and effective machine learning method which gained its fame through winning multiple Kaggle competitions. One of its strengths lies in parallel processing which makes the computation salable and faster than its counterparts. On the other hand, there are system configurations and model tuning parameters which need to be adjusted in order to achieve its full potential cost-effectively. In this paper, we explore how the training duration changes under different workloads, system configurations and framework parameters. By running multiple of these experiments, practical insights can be learned and applied for the future applications of XGBoost methods.

Keywords: XGBoost, Distributed Machine Learning, Big Data System

1 Introduction
Many open-source distributed gradient boosting frameworks demonstrate good properties on scalability. For example, CatBoost [1], a gradient boosting algorithm on decision trees model training, claims to deliver high performance and high-quality models with low memory usage and the least amount of parameter tuning. XGBoost [2] also claims to deliver efficiency, flexibility, and portability across languages with good support on multiple hardware. Both frameworks are out-of-the-shelf products and are widely used in many data-driven works [3–31].

Enterprise users on exploratory programming platforms have different understandings and demands for the performance and constraint of the frameworks. Exploratory workloads can have different requirements for the accuracy and latency of the tasks. For example, users at the beginning of the exploratory work might want to obtain a collection of very coarse-grained decision trees (depth ≤ 4) just to look at the possible choices forward. In comparison, the final decision tree model can be trained near-production just to verify the exploratory process.

As a result, vendors who provide platforms for general distributed machine learning must consider the various factor that minimizes the computational resources while delivering acceptable accuracy and latency for different tasks. There are many configuration spaces for vendors to consider: (1) framework parameters that determines the behavior of the training framework, (2) system configurations that determines the general performance. In addition, vendors are interested in the cost and resource utilization of these resources, and how well the system matches the expectation of the user (latency, accuracy, and cost).

In this project, we focus on the framework parameters and system configuration which directly affect the parallel operations designed for XGBoost training. We aim to study how we can achieve the most cost-effective degrees of parallelism under different workloads. The first motivation is driven by the fact that it is costly to run training on a large cluster. Therefore, it would be crucial to determine the cost-effective cluster size for a machine learning project suited for each workload. In addition, we would like to investigate the trade-offs between the latency and the accuracy of training models., and find the sweet spot which satisfies both requirements. Our goal is that through these investigations, we will gain insights into how a data science practitioner can make a good decision about system configuration and parameter tuning of their models.

The report is organized as follows. Section 2 reviews XGBoost and literature in system tuning. Section 3 introduces the basic setups and metrics we used. Section 4 describes our benchmarks and present the results. Section 5 describes some lessons learned during the environment setup to perform a big data experiment. We address limitation and future works in 6, and conclude the report in 7.

2 Background and Related Works
2.1 XGBoost
XGBoost [2] is a famous distributed gradient boosting library. It is designed to be scalable, efficient, and accurate for general machine learning problems. XGBoost follows the gradient tree boosting technique, but with a lot of optimizations that make it scalable. We briefly summarize these optimizations as follows.

(1) Split finding algorithm. Traditional exact greedy split-finding algorithm is not scalable because it enumerates all the possible splits for continuous features. To quickly find the splitting candidates in a distributed setting, XGBoost proposed an approximate algorithm to choose split candidates based on percentile, and use weighted quantile sketch to efficiently find these candidates (quantiles) in parallel. It
also uses a sparsity-aware split finding to handle all sparsity patterns in a unified way. This optimization reduces the computation for an iteration while having comparable accuracy.

(2) Column blocks and out-of-core computation. To perform split finding algorithm efficiently, the data needs to be sorted and retrieved efficiently. XGBoost organizes data into blocks, the in-memory data unit that stores data in compressed column format (CSC) where each column is sorted by the corresponding feature value. This pre-processing of data is only done once and can be reused. The data helps the approximate algorithm to perform linear scan to find the split point and helps statistics gathered efficiently through column sub-sampling.

Out-of-core computation (i.e. big datasets that need to be stored on disk) also relies on blocks as computation units. To reduce cost in I/O, XGBoost pre-fetches the data and multiplex I/O with computation to hide the cost. In addition, block compression is used to reduce data transfer, and block sharding to increase I/O throughput.

(3) Cache awareness. Non-continuous memory access can induce a significant amount of cache misses, thus harms the performance. XGBoost provides cache-aware access by a cache-aware prefetching algorithm that allows a thread prefetch and batch-accumulate gradient statistics. This mechanism works well when a dataset is large in memory.

2.2 System Tuning

Traditional system tuning (i.e. hand tuning) is very time consuming and heavily rely on experience and expert knowledge. In addition, the configuration changes are usually limited in a few dimensions and are hard to change given the dynamic changes of workload. Nevertheless, hand-tuning is still the most accessible and intuitive way for developers in a different stack to understand the system behavior, and flexibly tune for better performance. Konstantinos et al [32] also show the importance of a few knobs that can determine the overall performance of the database system. Grasping the few knobs at the first iteration in exploration is still very important.

Recent development in machine learning optimized systems provides feasible tools to continuously optimize the system by observing system behavior. The recently published Machine Learning Optimization System (MLOS) [33] uses machine learning techniques to optimize systems performances by providing optimal configuration with respect to the observations of the workloads. User register a sender in the application that continuously sends the current system configuration, workloads, and observed metrics (e.g. finish time, bandwidth) to the server. The optimizing server analyzes these data, and construct a surrogate model to estimate the optimal system configuration. The server also sends what it thinks is the optimal configuration to the system and see if the performance improves. This framework is open-sourced and under development.

3 Metrics and Experiments

XGBoost is quite versatile and is supported on a variety of platforms. This portability makes it very attractive as a tree boosting system. We understand the parallelism behavior under various settings and with different system configurations. This gives more insight on how a practitioner could use our recommended settings based on their situation.

3.1 Dataset

We work with the criteo terabyte click log data set [1]. We use this data set since it offers us to evaluate various aspects of XGBoost. The data contains 13 numeric features and 26 ID features of user, item and advertiser information. Following [2] we perform prepossessing of the ID features. We replace the ID features their count statistics and mean CTR. The entire training set consists of 1.7 billion instances with 67 features (replacing ID features with count and mean statistics). For experiments where we change the number of columns in a dataset, we perform other feature prepossessing step. This is mentioned in the experiment details.

3.2 Preliminary Experiments on small sampled data set

As a preliminary experiment, we measured the execution time of running XGBoost on smaller datasets of 4GB or less on the cluster size from 1 to 5 nodes. Each machine has 8 cores with 32 GB. We run this experiment on Spark cluster on the Azure Databricks platform.

3.3 Experiments to identify the optimal cluster size

To determine the optimal cluster sizes for different workloads, we will measure execution time for training a XGBoost for 10 iterations for data set of 0.4M, 0.8M, 4M, 8M and 85M entries with the prepossessed 40 numerical columns for cluster size of 4, 8, 16 and 25 nodes. We set up Spark cluster on Azure Databrick platform. Each node contains 8 cores with 32GB memory(Standard_D8_v3). The dataset is stored on the Databrick file system (DBFS). Note that 200 cores were the maximum cores we could rent from Azure due to our subscription policy and thus 25 was the maximum number of nodes we could use for this experiment.

3.4 Experiments with Distributed Dask Cluster

To further understand the parallelism properties and behavior of XGBoost, we set up a Dask cluster to execute the model training and the data is stored in a separate Hadoop cluster. The hadoop cluster consist of 2 nodes (Cloudlab c220g5). Each machine has 40 cores, 188 GB memory, and 2TB HDD
local storage. All the sharded files of Criteo dataset stores in these two nodes, with replication factor = 1.

To analyze the behavior of XGBoost, we use a distributed Dask Cluster setup on Cloudlab machines. We use 32 virtual machines nodes (hosted on c220g5) in the cluster. Each virtual machine node consist of 8 cores and 29 GB memory. Dask cluster starts 8 threads per worker. XGBoost does not have out of memory support for Dask distributed clusters. This limits our analysis to working with a fraction of criteo dataset consisting of approximately 196 million rows. For training and validation we consider a 80 : 20 split.

For this setup, we evaluate the framework behavior on following axis:

1. **Block size** used in creating dataframe. We consider following block sizes for our experiments: {16MB, 32MB, 64MB, 128MB}.

2. **Number of columns**: For this particular setup, we consider different column counts as permitted by the memory of the workers. We follow a different prepossessing to control the number of columns. The original ID columns are removed for convenience. Using the 13 numerical columns, we generate more columns by adding pairwise product of numerical columns as new features in the input. This form of feature engineering is popular when the model does not implicitly perform non-linear operations and is referred to as polynomial preprocessing. We consider the dataset with {13, 52, 78} columns.

3. **Thread count**: A common practice to achieve parallelism involves increasing the number of threads used by XGBoost for running the model training. We consider following thread counts for analyzing the system behavior: [1, 2, 4, 6, 8, 12, 16, 20].

For the various combinations of configurations generated using the above parameters, we train a tree regression using 10 rounds of tree boosting. For tree construction, we use the 'hist' option, which is a faster histogram optimizer approximate greedy algorithm. For all the experiments we plot the training time and dask matrix creation time for numerous configurations.

4 **Benchmark and Result**

In this section, we present our benchmarks and observations using the XGBoost on the Criteo dataset.

4.1 Small data set has Diminishing return of parallelism

We run several experiments on running XGBoost on smaller data set with cluster sizes from 1 to 5. The result is shown in figure 1. We observe that there is a diminishing return of parallelism. With increasing cluster size, the execution time seems to get better with a smaller margin each time. This finding is contrary to the experiment result from the paper [2], where they showed a linear improvement in execution time against cluster size for 1TB Criteo data set. This means that increasing the cluster size is not always a cost-effective approach to shorten the execution time especially when the data set is smaller.

4.2 Trade-off between error and latency

As a side experiment, we ran an experiment to observe the impact of bucket counts on the execution time and the accuracy of the final model. We ran the experiment with the same environment like the one above but with a fixed cluster size of 8. From figure 2 shows the result showing the execution time and MSE with different bucket counts. Note that Sketch_eps can be approximately converted to 1/sketch_eps bucket counts. We observed that with fewer buckets, the accuracy improves while the execution time worsens. Since with the less bucket, the estimation of the split point will deviate more from the optimal split point chosen by a greedy policy, the result brings us no surprise. However, it reiterates the importance of tuning for an optimal bucket count for the user to achieve the optimal execution time which achieves acceptable accuracy.

4.3 Optimal cluster size for different input data sizes

We would like to evaluate what might be an appropriate number of nodes to use for different data sizes. The results are shown in figure 3. The experiment shows that with all the data sizes, the execution time improves with an increasing number of nodes up to a certain point. After that turning point, the execution time starts increasing again. There seems to be a sweet spot for worker numbers for each data
size. We can also see that the sweet spot locates at a smaller cluster size for smaller data size. For instance the sweet spot for a data set of 0.4M entries, the sweet spot lies around 8 nodes while the sweet spot for a 8.5M entries dataset is located around 16 nodes. Figure 4 shows the optimal number of nodes used for different data sizes which could be used as a reference when deciding on the cluster size given a data size. Figure 5 shows the the same result as the figure 3 but now we see the execution time per 1 million entries. We can observe that the per million entry training execution is a lot better for larger data set at all cluster sizes. It is likely that XGBoost is giving more parallelism for the datasets with more entries.

4.4 Optimal number of threads for different column sizes
We evaluate the XGBoost system on Dask cluster to identify the optimal number of threads for datasets with various column counts. We use setup described in Section 3.4 for this. Figure 6 shows the training times against number of threads for datasets with various column counts. We make following observations from this experiment (a) The runtimes seem to reach optimum value for around 6 – 8 number of threads irrespective of the number of columns used. (b) XGBoost does not offer more parallelism with increasing column counts. The behavior of the training times is the same for all the column counts considered. Hence a practitioner should use the same optimum number of threads irrespective of the number of columns and expect higher training times for datasets with larger number of columns.
4.5 Trade-offs with block size

The block size used for storing the dataframe objects influences the parallelism behavior of XGBoost. We evaluate the dask DMatrix construction times and training times for different block sizes and column counts using the setup of 3.4. The block size can be specified while reading the data from hadoop. Based on previous subsections, we fix the number of threads used in training step of XGBoost to be 8. Figure 7 show the time needed to make DaskDMatrix ready before the training process. This step includes computing all the lazy evaluations like dropping matrices, adding additional columns, performing train/test split etc. We observe that as the block size becomes smaller that 32MB this step takes more time. We also find that for 16MB block size and smaller Dask runs into out of memory issues when number of columns are increased more than 52. The creation times remain relatively similar for other block sizes.

Next, we evaluate the training times for various block size and column size combinations. Figure 8 shows this evaluation. We observe that model training is faster for smaller block sizes and it seems that XGBoost is able to utilize larger thread counts to parallelize the tasks. However considering the time taken to construct the DaskDMatrix the optimum block size seems to be [32MB, 64MB]. These block sizes seem to provide balance between preprocessing the data and performing model training.

4.6 Behavior of training times with thread counts for different block sizes

We also look at how the training time behaves with respect to thread count for different block sizes. We again use the same setup as in Section 3.4 and fix the number of columns to
Figure 9. Comparison of training times for various block sizes and thread counts for the criteo dataset.

13. This corresponds to no preprocessing in terms of adding new columns. Figure 9 shows the behavior of training times for this experiment. The trend with thread counts is similar to that of other experiments. The optimum performance is obtained with thread count around 8. Additionally, we observe that for block sizes 16MB and 32MB the training times are relatively lower. However from the discussion in the previous subsection, we recommend a practitioner to use block size of 32MB in the experiments.

4.7 Result
As a summary of the section, we list the major results from the benchmark above.

- There is a limit to the degree of parallelism, after certain point, overheads of exceeds the gain from parallelism. For a smaller dataset with less than 4GB numerical dataset, use up to 8 nodes for training. For medium sized dataset up to 20GB of numerical data set, use up 16 nodes.
- Bucket size should be set by sketch_eps between the values 0.01 to 0.1 which should achieve the optimal combinations of MSE and execution time.
- The efficiency of training increases with the size of the data set regardless of the cluster size. Consider other approaches when your data set is small.
- Be aware of the implementation of XGBoost, if it does not support the out-of-core computation support, be sure to have nodes with enough memory capacity.
- There is a trade off between the training time and the time taken for constructing Dask Matrix. 32MB or 64MB block sizes seem to achieve relatively good performance for both matrix construction and training.
- Under many different situations, thread count of 8 seem to perform well by reaching the potential of the parallelism. The result was unchanged even with higher or lower dimensionality.

5 Environment Setup - Lesson learned
As greenhands into the world of big data systems, we had encountered many unexpected issues at environment setup and actual experiment. These are the real lessons we have learned frustratingly during our experiment, and by outlining these struggles we hope our readers will not step into the same river twice.

(1) Hadoop Security. By default configuration, Hadoop opens the public ports to the world, inviting hackers to attack the system. We had once suffered a malware attack, where an infamous bitcoin miner 2 intrude the system and occupies half of the cores to perform computation over the 4 host nodes in the Wisconsin cluster. We discovered the malware at a later time and had to discard the experiment results. This also influenced our timeline as the Cloudlab becomes crowded for a long period of time, and we were not able to perform any large-scale experiment until early December.

Solution: Thanks to Shivaram’s suggestion, we used the non-routable IP in the Cloudlab profile to prevent port exposure to the outer world. We also install a simple script to prevent this malware. So far, we have not been visited by malware.

(2) Dask (Pyarrow) Incompatibility with HDFS3. Dask is a popular framework in the data science community to manage parallel computing in a cluster-level environment. By default, Dask depends on pyarrow for data management, but pyarrow does not support HDFS3 for various reasons. As we had setup a Hadoop 3.10 cluster with 1TB data, we tried many hacks, but in vain. Further research shows that the community does not intend to maintain support for HDFS3.

Solution: Switching back to Hadoop 2.10, Dask worked as expected.

(3) Criteo 1TB Dataset. Migrating the 1TB Criteo dataset on Hadoop is not an easy task. The dataset host 3 only provide limited bandwidth (4 MB/s) for public users. We only used the 1 machine that host Hadoop to download the files, and it took about 2 days to download the whole Criteo dataset (compressed for 360GB). We then spent 2 days gradually upload the data onto the Hadoop server, and host it to the clients. Due to the malware, we also migrated the data to

---

2BitCoin Miner Virus syndrome in a casual StackOverflow post: https://askubuntu.com/questions/1225410/my-ubuntu-server-has-been-infected-by-a-virus-kdevtmpfsi
3Criteo dataset website hosted by Azure: https://labs.criteo.com/2013/12/download-terabyte-click-logs/
different machines using normal-speed linkage. Carrying 1TB of data around was not easy.

**Better Solution:** Allocate more machines connected with a high-speed link, use them to download the Criteo dataset in parallel, and release them later on. If it is needed to host a limited number of servers for Hadoop, then transfer those data to the designated Hadoop servers.

6 Limitation and Future Work

6.1 Framework Limitation

Dask and Spark are the two frameworks we chose to work with in our experiments simply because they are easily deployed on our machines. But both systems impose some visible overheads when working with XGBoost. Empirically, we see management overhead in DAG construction and maintenance, parallel data processing, and failure handling (in Dask). We were not able to quantify these phenomena as we did not store the logs for these experiments. We did not include any such failed experiments in our statistics for analysis either.

Ray [23] is also one of our option as the primary backend. Due to technical difficulty, we were not able to run any task successfully (e.g. workers fail arbitrarily, connection close exception, etc). Time permitted, we will try to see how Ray may provide less overhead in our workload.

6.2 Model Building Use MLOS

As mentioned in section 2, MLOS is a framework that learns from the observation and deliver suggestions to optimize systems overtime. The system is useful in the long term, but bootstrapping the model can take a long time, and it is nevertheless expensive to bootstrap. Besides, MLOS performs better if a programmer can provide more efficient encodings of data and specify implicit relationships between parameters. We hope this work can facilitate building a continuously optimized XGBoost computing cluster using MLOS.

7 Conclusion

In this project, we explored how the cluster size and different model tuning parameters can affect the execution times under different workloads. Although, there is no one right answer, there are multiple practical lessons learned which could be applied for the future application of XGBoost on datasets of various sizes. We have learned that the most cost-effective cluster size is mainly defined by the sample size. Generally, dataset with more entries benefit more from more parallelism. On the other hand, increasing dimensionality did not seem to gain more from the parallelism. Block size also seems to be an important factor which affects both the performance of matrix creation and actual training. By combining these insights, we hope that the future data science practitioner will be able to configure their system and tune their model in a more cost-effective manner.

Acknowledgments

To Shivaram, who enlightened our way to the world of big data system with joyful classes.

References


