Distributed Gradient Boosting Decision Tree on FlexPS

Zhou Junhong
Department of Computer Science and Engineering
The Chinese University of Hong Kong
1155035315@link.cuhk.edu.hk

Abstract

Gradient boosting decision tree (GBDT) is a widely used machine learning algorithm which achieves outstanding performance in many areas, such as academia, industry and data analytics competitions. There have many existing distributed implementations of GBDT with different distributing strategies, such as XGBoost implemented with all-reduce framework and MLlib from Spark implemented with map-reduce framework. In this paper, we introduce a new distributing strategy called parameter server to implement distributed GBDT. Our preliminary result shows that parameter server provides flexible synchronization models which can speed up training process and shows better scalability compared with existing distributed GBDT implementation with other frameworks.

Introduction

Nowadays, machine learning technique is playing an important role in both fields of academic research and industrial application. In particular, gradient boosting decision tree (GBDT), which produces a prediction model in the form of an ensemble of weak prediction models, has been prevailing and shows excellent learning result recently. According to the paper of XGBoost [1] (an open-source scalable tree boosting system), in 2015 Kaggle machine learning competition, among 29 challenge winning solutions, 17 solutions used XGBoost.

However, with the explosive increase of dataset and needs of faster training time, a single machine is incapable to store and process enormous data volume for machine learning application. It is evitable to deploy machine learning applications in distributed environment. There are several distributing machine learning frameworks in state-of-the-art distributed systems, such as map-reduce framework to merge parameters in MLlib and all-reduce MPI to summarize local solutions in XGBoost. However, both map-reduce and all-reduce framework suffer from a single-point bottleneck when parameters have considerably large size. Hence, their paradigms of model aggregation are ill-suited for multi-dimensional features. In particular, parameter server [2] framework is well-suited for many distributed machine learning problems, some advantages include:

- Partitions the parameters across several machines to avoid the single-pint bottleneck.
- Provides flexible synchronization models, BSP/SSP/ASP for better trade-off between computation efficiency and algorithm efficiency.

FlexPS is a parameter server system developed by Husky team, which provides high scalability and elasticity. In this paper, we will analyse how GBDT can be implemented under parameter server framework, implement GBDT in FlexPS and finally compare and discuss the performance with different settings and systems.

1 Background

1.1 Parameter Server

Architecture: There two types of entities in parameter server architecture, worker and server. Each worker fetches only a part of training data and processes locally without interact with other workers. Servers provide a KV-store interface for workers to send local result. Moreover, servers will aggregate the local updates from workers into a global model.

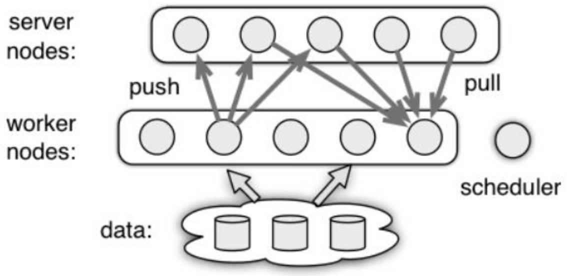


Fig. 1: Server-worker architecture in parameter server

Consistency Models: In a distributed parallel environment, multiple workers are required to generate updates to shared global parameters simultaneously. Such enforcing strong consistency will lead to frequent, time-consuming synchronization and thus limit the computation speed from parallelization. Therefore, parameter server provides three typical consistency protocols to relax this issue: Bulk Synchronous Parallel (BSP), Stale Synchronous Parallel (SSP) and Asynchronous Parallel (ASP). BSP forces each worker to be synchronized after each iteration and retain a consistent view of the global parameter model for each worker. SSP proposes a bounded delay for relaxing above strict consistency control. That is, the fastest worker is allowed to be s steps ahead of the slowest workers. It also had been proven that [3] some algorithm (such as stochastic gradient descent) will converge using SSP with staleness > 0. ASP, on the other hand, does not emphasis any synchronization on workers. Each worker updates the model at its own pace and hence there has no convergence guarantee for some ML algorithms.

1.2 State-of-the-art PS Systems

In this section, we briefly summarize recent major PS systems or systems implemented with parameter server abstraction and introduce our PS system, FlexPS.

Application-specific PS system. Before, when PS architecture has not been developed, researchers had used similar concept of parameter servers to tackle ML problem, such as neural network which requires billions of parameters update and storage during training. Google's DistBelief and Microsoft's Project Adam system was built specifically to support deep learning algorithm. These systems do not provide PS framework for general machine learning tasks.

General-purpose PS system. There have PS systems that support general distributed ML algorithm, such as Mu Li's Parameter Server and Petuum. These systems more focus on scalability and fault-tolerance than application-specific PS systems.

FlexPS system. Compare to existing PS systems, an important feature of FlexPS is called flexible parallelism control [4], which has not been included in existing work. With the multi-stage abstraction, a ML task can be mapped to as series of stages and the parallelism for a stage can be set according to its workload. It was discovered that this feature achieves significant performance improvement in terms of both efficiency and resource utilization.

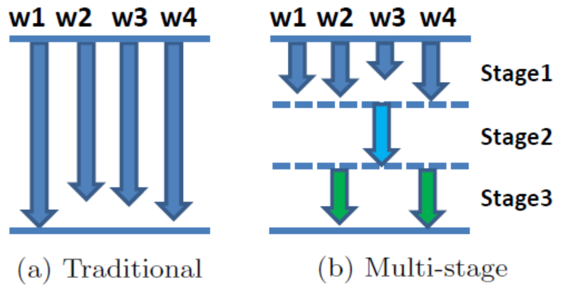


Fig. 2: Single-stage v.s. multi-stage

1.3 Related Work

As GBDT is a powerful machine learning algorithm, many researchers try to implement it in some distributed systems so that GBDT can be applied for terabyte or larger dataset. Nowadays, there have several prevailing distributed systems implementing GBDT with different distributed frameworks.

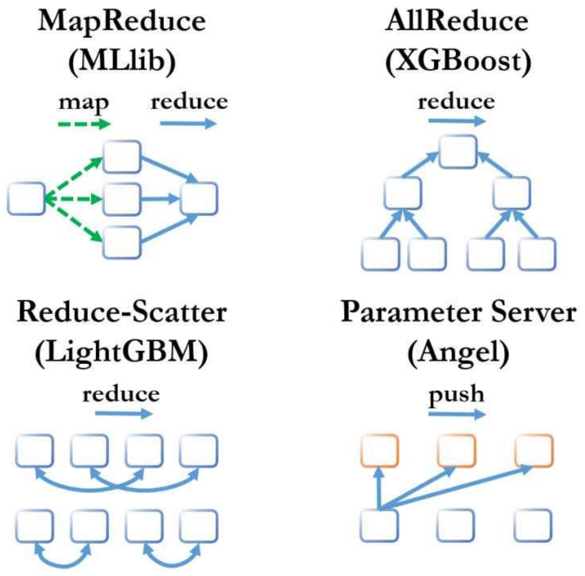


Fig. 3: Different parallelism strategies for GBDT

MapReduce. MLlib uses MapReduce framework for model aggregation. However, when processing big dataset, the single point of reducer will become a bottleneck.

AllReduce. XGBoost uses AllReduce MPI for model aggregation. AllReduce framework uses workers to construct a decision tree and aggregate parameters from bottom to top. This will also encounter a single-point bottleneck as MapReduce.

Reduce-Scatter. LightGBM uses Reduce-Scatter framework for model aggregation. Unlike AllReduce, workers in Reduce-Scatter will only aggregate parameters partially.

Parameter Server. Angel uses Parameter Server framework for GBDT where parameter models are distributively stored in several servers, thus worker only need one step communication to push and pull the parameter model.

Overall, we can notice that Reduce-Scatter and Parameter Server are more efficient than the other two frameworks (as they do not have single point bottleneck problem). Moreover, it was pointed out[1] that when the number of workers is 2^N, the total communication cost of Reduce-Scatter is double larger than that of Parameter Server. Therefore, Parameter Server is a more efficient and general choice of implementing distributed GBDT.

[1] https://zhuanlan.zhihu.com/p/28319178

2 Gradient Boosting Decision Tree

2.1 Algorithm

Gradient Boosting Decision Tree, also known as Multiple Additive Regression Tree (MART), is a tree ensemble model that combines weak “learners” into a single strong learner in an iterative manner. To be more specific, given a training instance x_i, assuming we have T trees, each tree f_t will classify the instance to one leaf and return a predicting leaf weight w_{it}. GBDT sums up all the predictions of all the trees and return the result as final prediction.

$$\hat{y}_i = \sum_{t=1}^{T} f_t(x_i) = \sum_{t=1}^{T} w_{it}$$

Training method of a regression tree:

```
Algorithm 1: Regression Tree with Square Error
Input: Training dataset D with feature dimensions M
(1)
1:  for each feature dimension i = 1 to M do
2:      X ← i-th dimension feature values from D
3:      Y ← class values from D
4:      split Y into left and right portions, $I_L$ and $I_R$,
        with different split values from X such that
5:      Y= $I_L \cup I_R$
6:      gain = $\sum_{y\in Y}(y-\bar{y})^2 - (\sum_{y\in I_L}(y-\bar{y}_L)^2 + \sum_{y\in I_R}(y-\bar{y}_R)^2)$
7:      return the split value s and dimension i with
        maximum gain
8:  end for
(2)
Split the dataset D into $D_L$ and $D_R$ such that:
    $D_L = \{x, y \mid x^{(i)} \le s\}$
    $D_R = \{x, y \mid x^{(i)} > s\}$
(3)
Repeat (1), (2) until satisfy stop conditions
(4)
Finally, return a regression tree $f$ and the space of
dataset is split into $D_1, D_2, \dots, D_M$. The predict value
will be:
    $f(x) = \sum_{m=1}^{M} \bar{c}_m I(x \in D_m)$,
where $\bar{c}_m$ is the mean of $y$ in $D_m$, function I return 1
if $x \in D_m$ and 0 otherwise.
```

Training method of gradient boosting decision tree:

A boosting decision tree is to iterate multiple regression trees to make final prediction. If we take square error as loss function (as above), each regression tree learns from the residual between the conclusion of all previous tree and actual predict value. In general case, let the loss function (learning objective) to be $l(y, f(x))$, where y is actual predict value and $f(x)$ is the sum of predict value from trees, the training method of GBDT can be written as follows:

```
Algorithm 2: Gradient Tree Boosting Algorithm
Input: Training dataset D with size N, T number of
trees
(1) Initialize $f_0(x) = argmin_\gamma \sum_{i=1}^{N} l(y_i, \gamma)$
(2)
1:  for t = 1 to T do
2:      for i = 1 to N do
3:          $r_{it} = -[\frac{\partial l(y_i, f(x_i))}{\partial f(x_i)}]_{f=f_{t-1}}$
4:      end for
5:      Fit a regression tree to target $r_{it}$ giving
        terminal regions $R_{jt}$, j = 1, 2, …, $J_t$.
6:      for j = 1 to $J_t$ do
7:          $\gamma_{jt} = argmin_\gamma \sum_{x_i \in R_{jt}} l(y_i, f_{t-1}(x_i) + \gamma)$
8:      end for
9:      Update
10:     $f_t(x) = f_{t-1}(x) + \sum_{j=1}^{J_t} \gamma_{jt} I(x \in R_{jt})$
11: end for
(3)
Output GBDT $\hat{f}(x) = f_T(x)$
```

We can further improve this algorithm by making the following adjustments:

Regularized objective function. when training the t-th tree, we need to minimize the loss function with a penalty function **Ω** to avoid over-fitting.

$$L^{(t)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t)}) + \Omega(f_t)$$

$$= \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

Second-order approximation. The second order method is originated from Friedman et al. [1]. This approximation can be applied for faster optimization of the objective in general setting.

$$L^{(t)} \approx \sum_{i=1}^{N}[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

Then, we can remove the constant terms in the objective function to obtain simpler equation.

$$\tilde{L}^{(t)} \approx \sum_{i=1}^{N}[\, g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ and $h_i = \partial^2{}_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ are the first and second order gradient statistics on the loss function.

Further, we define tree by a vector of scores in leaf, and a leaf index mapping function that maps an instance to a leaf

$$f_t(x) = w_{q(x)}, w \in R^T, q: R^d \rightarrow \{1, 2, \dots, T\}$$

$$I_j = \{i | q(x_i) = j\}$$

where w is the leaf weight of the tree and q is the structure of the tree.

Then we can redefine equation as below:

$$\tilde{L}^{(t)} = \sum_{i=1}^{N}[\, g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \lambda \frac{1}{2} \sum_{j=1}^{T} w_j^2$$

$$\tilde{L}^{(t)} = \sum_{i=1}^{N}[\, (\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

The optimal weight $w_j^*$of leaf f is given by

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

and the corresponding optimal value is given by

$$\tilde{L}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T$$

The equation can be used as scoring function to measure the impurity within a set of data when we find the splitting value during constructing a decision tree.

In this project, we adopt greedy algorithm to split the tree nodes successively. Suppose that I_L and I_R are the splitting sets when we train a node, let $I = I_L \cup I_R$,we can define the gain of split as follows:

$$L_{gain} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

Equation (1)

split candidate with maximum gain with be chosen as the best splitting feature.

2.2 Quantile Sketch

One of the key issue in decision tree learning is to find the best split feature to split the dataset into left child and right child for successive learning. Tradition strategy of finding this split spot in GBDT is to first sort all the features and then to enumerate all the possible splitting candidates based on the sorted feature lists. However, for a large dataset in a distributed environment, iteratively sorting of the dataset will be a severe problem. One reason is that workers need to communicate over the network during sorting and cause large overhead of bandwidth usage. The other is that creating sorted feature lists (or sort indexes) will cost a lot of memory when the original dataset is large.

Here, quantile sketch offers an alternative approximate approach to find the split candidates through there is a trade-off. The trade-off is that we cannot scan all the possible split points but can only find the points that are roughly located at some percentiles of the dataset. It turns out to have much faster convergence speed and still remain satisfactory accuracy. The idea of weighted quantile sketch as follows:

- Step 1: Specify the desired range percentiles and sketch points for each percentile, say C candidates of percentiles and S sketch points for each percentile;
- Step 2: Create a histogram based on sketch spots, in this case, we have S * C bins in the histogram;
- Step 3: Normalize the feature values. If the normalized value falls in some bin, that bin accumulate by 1 (or by gradient of loss function in general case);
- Step 4: For each candidate of percentile, return the sketch point that most “approximates” that percentile and de-normalize it to corresponding feature value.

We can notice that this algorithm is particularly suitable for parameter server framework as each worker can push its local statistics of histogram and pull the global histogram from server without interrupting other workers compared to sorting. In addition, the computation cost of quantile sketch algorithm is O(k*N) where k is some constant and N is number of data whereas that of sorting algorithm is O(N*logN). Our training process will run much faster when N is large.

3 GBDT on FlexPS

3.1 Design

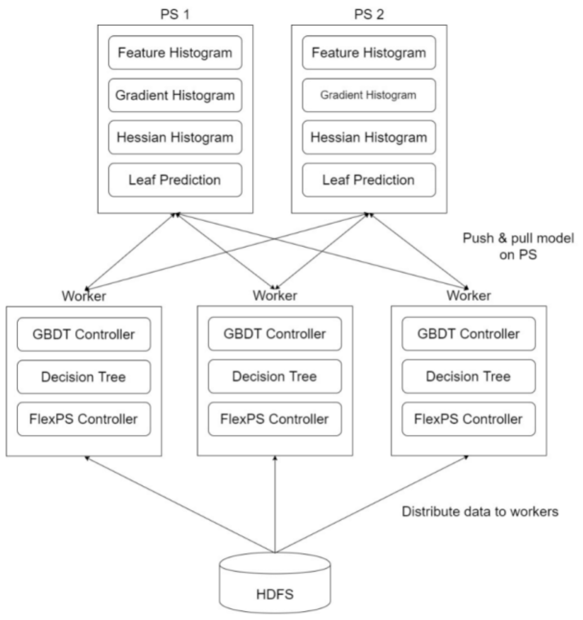


Fig. 4: Architecture of GBDT under PS framework

Data Parallelism. Data is stored distributively using HDFS. Each worker will fetch a portion of the dataset according to its node id and worker id.

Model Parallelism. FlexPS supports model parallel by allowing each worker pushes its local model and forming a global model.

The above diagram illustrates the architecture of GBDT in FlexPS. Servers and workers have different responsibilities during training GBDT. Each worker will only process the data locally that fetched from HDFS. On the other hand, parameter servers are responsible for receiving and aggregating parameters from workers to form a global view of the training model.

Now we will explain the definitions that showed in the architecture diagram.

Server side

Sketch Feature Histogram Table. it is used to store sketch histogram in calculating quantile sketch stage.

Gradient and Hessian Histogram Table. they are used to store gradient and hessian histograms in finding the best split feature stage.

Basically, they are the terms $g_i$and h_i in equation (1). In each round, each worker pushes $\sum_{i\in I_w} g_i$ and $\sum_{i\in I_w} h_i$ where I_w standards for dataset in this worker, and pulls $\sum_{i\in I} g_i$ and $\sum_{i\in I} h_i$, where $I = \bigcup_{w\in W} I_w$, W is the set of workers.

We can notice that $I = I_L \cup I_R$, then we have

$$\sum_{i\in I} g_i = \sum_{i\in I_L} g_i + \sum_{i\in I_R} g_i$$

similar for hessian histogram. This implies that we can reduce the size of sending message to half by only sending gradient and hessian in I_L and $I_R$to parameter servers.

Leaf Prediction Table. it is used to store prediction sums and prediction counts in finding prediction stage.

Worker side

GBDT Controller. it is used to control the parameter and option settings, load dataset, train forest, and import and export forest file.

Decision Tree. decision tree is built by many nodes. It is used to train model and make prediction.

FlexPS Controller. it is used to provide PS framework abstraction for the algorithm by calling FlexPS API, such as KV-store, ML tasks and consistency controller.

3.2 Distributed Execution Plan

As GBDT algorithm is written in FlexPS framework, there have some terminologies and API we need to define to make the algorithm compatible with it.

Engine. The core PS engine for server and worker. Parameter server can be launched by calling engine.StartEveryThing() and be terminated by calling engine.StopEveryThing().

Range. The parameter range sets on servers. This can be set to PS system by passing as parameter to engine.CreateTable().

Worker_alloc. The assignment of workers (threads) in each node.

MLTask. Users can program their ML algorithms in this part by implementing function in task.SetLambda(). After completed, echo engine.Run(task) to launch.

Distributed GBDT

GBDT algorithm is highly iterative. Namely, it trains each tree with the same training method. Each tree is constructed by many nodes which also have the same constructing method. To retain consistent progress of workers, workers will train on the same node of the same tree each time so that even through the program is written in recursive fashion, it is guarantee that each worker has the same progress.

Pre-processing data. In pre-processing stage, the maximum and minimum values of each feature are required as we need to normalized feature value at building quantile sketch stage.

The workflow of training a node can be discussed in several phases, each phase can be summarized by two key steps operation on KV table: Add to table and Get from table.

Phase 1 - Find Split Candidate. Each worker generates local quantile sketches and then call sketch_feature_histogram_table.Add() to push it to the PS. Next, each worker pulls the merged sketches from PS by calling sketch_feature_histogram_table.Get() and proposes split candidate for each feature.

Phase 2 - Find Max Split Gain. Each worker computes the local gradient and hessian histogram and push to PS by calling gradient_histogram_table.Add() (similar for hessian histogram). Next, each worker pulls the global histogram by calling gradient_ histogram_table.Get() and hessian_histogram_ table.Get() and finds the best splitting feature id and value.

Phase 3 - Find Leaf Prediction. If the stopping condition arrives (such as exceeding max depth of tree), worker sums up local class values and counts local data size in that node (leaf) and pushes to PS by calling leaf_prediction_table.Add(). Then, it calls leaf_prediction_table.Get() to get the global sum and number and calculate predict value.

Phase 4 – Reset PS. After finished training a node, each worker will send negative local values of histograms to PS for the next use.

Before training the next nodes (left child node and right node), the dataset will be separated according to current best split value. The maximum value of splitting feature of left child and the minimum value of splitting feature of right child will be update to the split value.

4 Experiment

The datasets are downloadable from internet[2]. We have use two datasets, one for local usage and the other for clusters usage.

Dataset	Record Num	Feature Num	Usage
cadata	20.6k	8	Local
YearPredictionMSD	463.7k	90	Cluster

Table 1: Dataset information

We use root-mean-square error (RMSE) to measure the accuracy of training result. Lower RMSE value means that the ML model more fits to the dataset. The formula is given as below:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

where $\hat{y}_i$ is the evaluate result, y_i is the actual result and n is the number of records.

[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression.html

4.1 Scalability

This experiment is to check scalability of GBDT in FlexPS. That is, to check the effect of increasing number of running threads on the running time of the program.

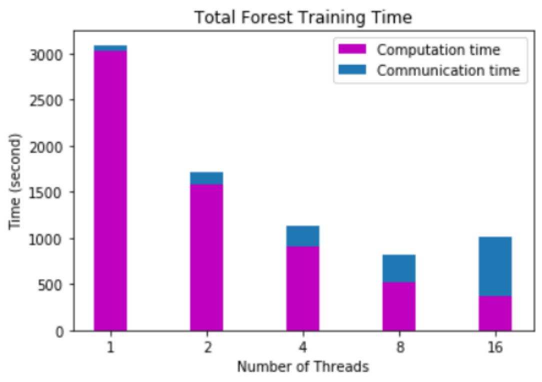


Fig. 5: Relationship between number of threads and execution time

We use YearPredictionMSD dataset to test the scalability. The parameters of GBDT are set to:

- Number of tree: 20
- Max depth per tree: 6

The result is shown as above diagram. The experiment is ran with different thread numbers from 1 to 16. We can notice that the total training time is decreasing from thread number 1 to thread number 8 as computation time decreases dramatically while the communication time increases slightly. However, when thread number equals 16, the total training is even higher than that of thread number 8 due to the rapid growth in communication time, despite that it has lower computation time.

In conclusion, it proves that our implementation is indeed distributable. Adding more workers can speed up the training time by distributing computation workload to workers. However, it does not mean that more workers are guaranteed to speed up the program. We also need to aware the communication time between server and worker. When there are too many workers that need to be synchronized but computation workload is small, synchronization will become a barrier for distributed algorithm.

4.2 Stale Synchronous Parallel

This experiment aims to study the effect of SSP model on GBDT. One major different between BSP and SSP of GBDT is that trees built by each worker in BSP must be identical as each worker will be synchronized at each phase and pull the same global model from PS. In this case, the trained decision tree is meaningful. We can make prediction and export it from any worker. However, for SSP with staleness > 0, there has no guarantee on the constructed decision tree in each worker as some workers are allowed running faster without waiting for slow workers. For those fast workers, it is possible that they only use partially global model or even only the local model to train the tree.

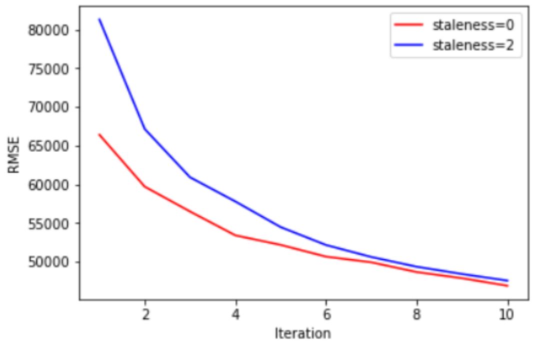


Fig. 6: Performance of RMSE in different staleness settings

Staleness	Computation Time (Sec)	Communication Time (Sec)
0	90.14	23.32
2	79.56	18.61

Table 2: Execution time with different staleness

In this experiment, we use cadata dataset with 5 threads. The program is executed with staleness=0 (BSP model) and 2 (SSP model) two settings. From the line chart above, we can notice that initially the RMSE of two models are obviously different but it becomes similar as the number of trees in the forest grows. This suggests that with sufficient amount of trees in the forest, the prediction result of BSP and that of SSP will converge.

On the other hand, from the above table of training time per tree of two models, we can notice that SSP not only has faster computation time but also achieves shorter communication time. In short, this experiment offers us an insight that we can use training accuracy to trade training speed in GBDT algorithm.

4.3 Comparison with Other Systems

This part of experiment aims to compare our implementation with existing distributed GBDT system in two aspects, training time and training accuracy, and understand the performance and limitation of our current implementation among industrial standard distributed systems.

We had compared with MLlib[3] and XGBoost[4], while XGBoost is one of the most popular distributed GBDT library in open-source community. For fairness, we use the same parameters setting of decision tree among all systems. Although systems might have different training methods and terminology, we only evaluate the resulting RMSE and the training time of attaining that RMSE.

In addition, we run the comparison using 1 thread and other number of threads separately. The reason is that 1 thread can help us compare the algorithm implementation fundamentally without effect of distributed system (algorithm v.s. algorithm), while more threads can help us understand how well can our algorithm be run distributively (distributed system + algorithm v.s. distributed system + algorithm)

Compare with MLlib in Spark

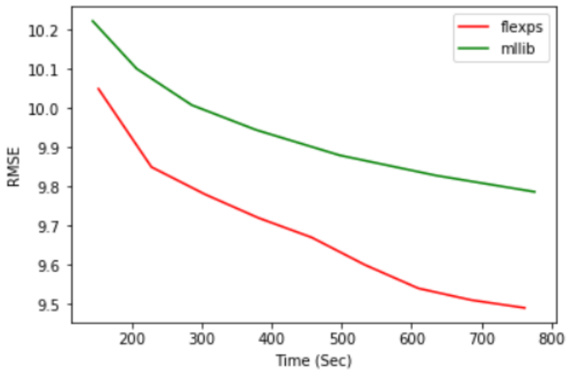
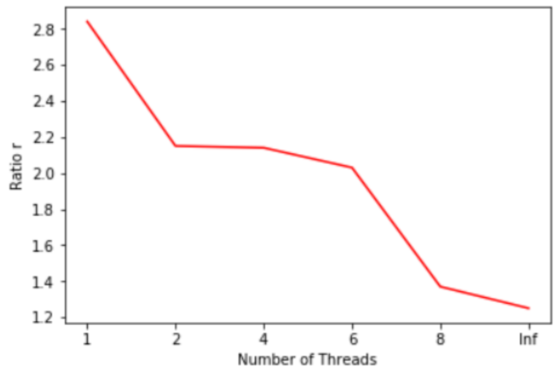


Fig. 7: FlexPS v.s. MLlib in 1 thread

By comparing algorithm itself, we can notice that our algorithm not only continuously outweighs the one in MLlib, but also has a better learning efficiency. That is, over this period, our algorithm improves the RMSE by 0.84, while MLlib improves by 0.43.

In order to compare the performance of two systems in different number of threads, we define a ratio r_t = (execution time of MLlib of obtaining the same RMSE with t threads) / (execution time of FlexPS of obtaining the same RMSE with t threads). If r_t > 1, it means our systems performs better than MLlib with t threads, and vice versa. If r_t=1, it means that two systems perform the same with t threads. Below graph plots r_t versus different number of threads. The target RMSE is 9.66.



[3] https://spark.apache.org/mllib/

[4] https://github.com/dmlc/xgboost

Fig.8: Ratio r v.s. number of threads (MLlib)

In above, we use 'Inf' to represent that sufficient threads are provided (maximum 24 threads in our experiment, but already sufficient for this task) to programs and each program achieves its best performance. We can notice that our distributed method runs faster than MLlib by more than twice when number of threads is less or equals to 6. Moreover, the eventual ratio is 1.25, which implies that given sufficient resource of workers, the scalability of our program still overcomes that of MLlib (that is, if some task takes us 1 second to finish, it would take 1.25 seconds for MLlib to finish).

Compare with XGBoost

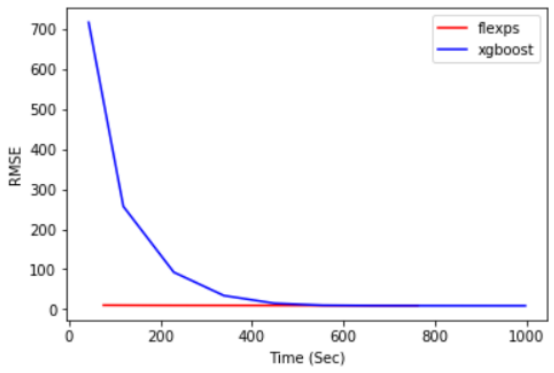


Fig. 9: FlexPS v.s. XGBoost in 1 thread

First, we eliminate the effect of distributing system by running the programs in only one thread, the result is illustrated as above. We can notice that XGBoost dramatically improves its RMSE from around 700 at 43 seconds to 8.99 at 800 seconds. Our implementation, however, obtains a very good result initially, namely RMSE equals 10.33 at 75 seconds, but the final RMSE is 9.49 at 900 seconds.

To compare the scalability performance of two systems in different number of threads, we replace the ratio r_t = (execution time of XGBoost of obtaining the same RMSE with t threads) / (execution time of FlexPS of obtaining the same RMSE with t threads). Below graph plot r_t versus different number of threads. The target RMSE is 10.05.

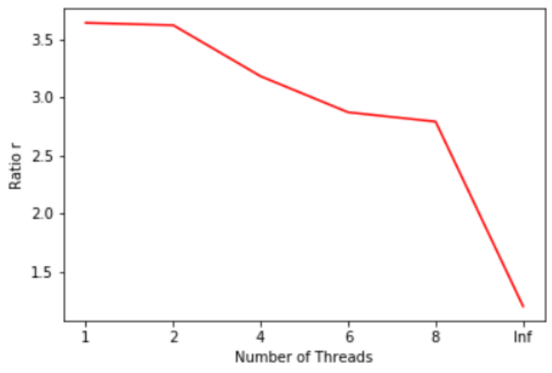


Fig. 10: Ratio r v.s. number of threads (XGBoost)

We can notice that our program is over 2.5 times faster than XGBoost when number of threads is less or equals 8. The final ratio is 1.20, thus our scalability is still better than XGBoost.

To sum up, it seems that the current bottleneck of our system is that it has good predicting result at the beginning, but it fails to obtain a much more precise model compared with XGBoost. Our next focus will study on improving this.

5 Conclusion

In this paper, we described a new distributing strategy, parameter server framework, for GBDT algorithm. Over several experiments on our system, we retrieved some representative results listed below:

(1) We found that adding bounded delay among workers can increase the training speed of tree but the training accuracy will decreases. Nonetheless, with sufficient among of iterations, the predict accuracy will converge eventually.

(2) We compared our system with two prevailing distributed systems and found that our system (or parameter server) shows better scalability for paralleling GBDT algorithm with sufficient resource of workers.

Therefore, these discoveries suggest that parameter server provides a more flexible and scalable distributing strategy for GBDT implementation.

6 Future Work

1. In the experiment of comparing with XGBoost, our system fails to achieve the same or even better result of training accuracy. Thus, we will analyse and learn from XGBoost and other literatures to improve our model accuracy;
2. Our current GBDT implementation on FlexPS is still not fully featured. Next, we will continuously improve the program to make it more integrated, such as improving function interfaces, supporting model import and export and supporting categorical classification.

7 Acknowledgement

I would like to thank below people in this project:

- Prof. James Cheng, project supervisor, for offering research opportunity and kindly supervision.
- Yuzhen Huang, PhD student of James and author of FlexPS, for guidance and many suggestions in this project.

8 References

[1] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in KDD, 2016, pp. 785–794.

[2] Mu Li, “Scaling Distributed Machine Learning with the Parameter Server” in OSDI, 2014.

[3] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. “More effective distributed ml via a stale synchronous parallel parameter server” in NIPS, 2013.

[4] J. Wei, W. Dai et al. Managed communication and consistency for fast data-parallel iterative analytics. In SoCC, 2015

[5] Y. Huang, T. Jin et al. “FlexPS: Flexible Parallelism Control in Parameter Server Architecture”.

[6] Angel. https://github.com/tencent/angel.

[7] Apache Spark. https://spark.apache.org/.

[8] LightGBM. https://github.com/Microsoft/LightGBM.

[9] XGBoost. https://github.com/dmlc/xgboost.

[10] Jie Jiang, Jiawei Jiang et al. “TencentBoost: A Gradient Boosting Tree System with Parameter Server” in IEEE 33rd International Conference on Data Engineering, 2017.

[11] Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters.” December 2004, http://labs.google.com/papers/mapreduce.html

[12] Rajeev Thakur, Rolf Rabenseifner et al. “Optimization of collective communication operations in MPICH”. The International Journal of High Performance Computing Applications, 2005.