2019 IEEE 35th International Conference on Data Engineering (ICDE)

MLlib*: Fast Training of GLMs using Spark MLlib

¶§Zhipeng Zhang §Jiawei Jiang †Wentao Wu ‡Ce Zhang §Lele Yu ‡¶Bin Cui
¶*School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University*
‡*Center for Data Science, Peking University & National Engineering Laboratory for Big Data Analysis and Applications*
†*Microsoft Research, Redmond, USA* ‡*ETH Zurich, Switzerland* §*Tencent Inc.*
¶{*zhangzhipeng, bin.cui*}*@pku.edu.cn,* †*wentao.wu@microsoft.com*
‡*ce.zhang@inf.ethz.ch,* §{*leleyu, jeremyjiang*}*@tencent.com*

***Abstract*—In Tencent Inc., more than 80% of the data are extracted and transformed using Spark. However, the commonly used machine learning systems are TensorFlow, XGBoost, and Angel, whereas Spark MLlib, an official Spark package for machine learning, is seldom used. One reason for this ignorance is that it is generally believed that Spark is slow when it comes to distributed machine learning. Users therefore have to undergo the painful procedure of moving data in and out of Spark. The question why Spark is slow, however, remains elusive.**

In this paper, we study the performance of MLlib with a focus on training generalized linear models using gradient descent. Based on a detailed examination, we identify two bottlenecks in MLlib, i.e., *pattern of model update* and *pattern of communication*. To address these two bottlenecks, we tweak the implementation of MLlib with two state-of-the-art and well-known techniques, *model averaging* and *AllReduce*. We show that, the new system that we call MLlib*, can significantly improve over MLlib and achieve similar or even better performance than other specialized distributed machine learning systems (such as Petuum and Angel), on both public and Tencent's workloads.

***Index Terms*—Distributed Machine Learning, Spark, Generalized Linear Models, Gradient Descent**

I. Introduction

The increasing popularity of Spark has attracted many users, including top-tier industrial companies, to put their data into its ecosystem. As an example, we inspected daily workloads on the Tencent Machine Learning Platform and found that more than 80% of the data were extracted and transformed by using Spark [1]. However, if we look at the machine learning workloads in more details, only 3% of them actually use MLlib [2], an official Spark package for machine learning. The rest of the workloads simply run on top of other specialized machine learning systems, such as TensorFlow [3], XGBoost [4] and Angel [5], as depicted in Figure 1. This implies significant data movement overhead since users have to migrate their datasets from Spark to these specialized systems. So why not just use MLlib? One important reason is that Spark is generally believed to be slow when it comes to distributed machine learning [6].

Nonetheless, it remains unclear *why* Spark is slow for distributed machine learning. Previous works mainly attribute this inefficiency to the architecture Spark adopts. Spark is architected based on the classic Bulk Synchronous Parallel (BSP) model, where the driver node can be a bottleneck when training large models, due to the overwhelming communication overhead between the workers and the driver. Nonetheless,

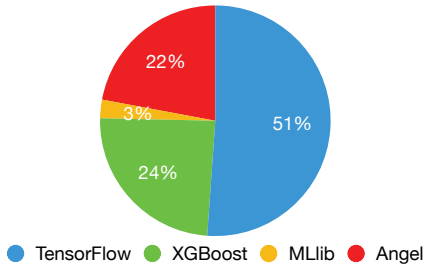


Fig. 1. ML workloads in Tencent Machine Learning Platform

is it a fundamental limitation that is not addressable within the Spark architecture? If so, what is the innovation in the architectures leveraged by the specialized systems that addresses or bypasses this limitations? Meanwhile, is this really the *major* reason for the inefficiency of Spark? Are there actually other bottlenecks that have not been identified yet? If so, are those bottlenecks again due to fundamental limitations of BSP or just a matter of implementation issue? As far as we know, none of these questions has been sufficiently studied.

In this paper, we aim to understand in more detail why Spark MLlib is slow. We focus on training generalized linear models (GLM) as a case study, which include popular instances such as Logistic Regression (LR) and Support Vector Machine (SVM). Our exploration reveals that it is actually implementation issues rather than fundamental barriers that prevent Spark from achieving superb performance. Although the original performance of MLlib is indeed worse than that of specialized systems based on parameter servers, such as Petuum [7] and Angel [5], by slightly tweaking its implementation we are able to significantly speed up MLlib on both public and industrial-scal workloads while staying in the ecosystem of Spark.

Specifically, our study identifies two major performance bottlenecks in the current MLlib implementation of gradient descent (GD), one of the most popular optimization algorithms used for training GLMs that is based on taking first-order derivatives of the objective function [8].

B1. Pattern of Model Update. First, the update pattern of model in MLlib is not efficient. In MLlib there is a *driver* node responsible for updating the (global) model, whereas the *worker* nodes simply compute the derivatives and send them to the driver. This is inefficient because the global model shared by the workers can only be updated *once*

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00194

per communication step between the workers and the driver. We address this issue by leveraging a simple yet powerful technique called *model averaging* [9] that has been widely adopted in distributed machine learning systems. The basic idea behind model averaging is to have each worker update its local view of the model and the driver simply takes the average of the local views received from individual workers as the updated global model. In this way the global model is actually updated *many times* per communication step and therefore we can reduce the number of communication steps towards convergence.

B2. Pattern of Communication. Second, the communication pattern in MLlib can be improved. In MLlib while the driver is updating the model, the workers have to wait until the update is finished and the updated model is transferred back. Apparently the driver becomes a bottleneck, especially for large models. By using model averaging this bottleneck can be completely removed — we do not need the driver per se. In essence, model averaging can be performed in a distributed manner across the workers [10]. Roughly speaking, we can partition the model and have each worker maintain a partition. There are then two rounds of shuffling during model averaging. In the first round of shuffling, each worker sends all locally updated partitions to their dedicated maintainers. Afterwards, each worker receives all updates of the partition it is responsible for and therefore can perform model averaging for this partition. The second round of shuffling then follows, during which each worker broadcasts its updated partition to every other worker. Each worker then has a complete view of the updated (global) model afterwards. Compared with the centralized implementation MLlib currently leverages, this distributed implementation does not increase the amount of data in communication — the total amount of data remains as $2km$ if we have k workers and the model size is m. However, it significantly reduces the latency as we remove the driver.

Our experimental evaluation on both public workloads and Tencent workloads shows that, MLlib*, the improved version of MLlib by removing the aforementioned two bottlenecks in MLlib, can achieve significant speedup over MLlib. Furthermore, it can even achieve comparable and often better performance than specialized machine learning systems like Petuum and Angel.

In summary, this paper makes the following contributions:

- We provide a detailed analysis of implementations of existing distributed machine learning systems, including MLlib, Petuum, and Angel.
- We identify two major performance bottlenecks when running MLlib (*i.e., inefficient pattern of model update* and *inefficient pattern of communication*). By carefully designing and consolidating two state-of-the-art techniques in MLlib, we implement a new machine learning library on Spark called MLlib*.
- We show that MLlib* can achieve significant speedup over Spark MLlib. As an extreme example, on one of our datasets we observed 1,000× speedup.
- We further compare MLlib* with specialized machine learning systems such as Petuum and Angel. We show that MLlib* can achieve close or even better performance compared with systems powered by parameter servers.

We do not view our results in this paper as a sales pitch for MLlib*, rather, we view this paper as a promising start and/or baseline that can spawn a new line of research. First, given that Spark can indeed be more effective in dealing with machine learning workloads, it might be worth to revisit reported results in the literature that were based on an inefficient MLlib. Second, the techniques we used to improve MLlib may also be used to improve other Spark-based machine learning libraries. Third, while we have focused on training GLMs in this paper, similar studies can be conducted for other models as well, as gradient descent is not tied to training GLMs. We leave these as interesting directions for future work.

Paper Organization. We start by presenting necessary background in Section II. In particular, we present a generic architecture that illustrates common paradigms used by existing distributed machine learning systems. We then leverage this generic architecture to analyze the implementations of MLlib, Petuum, and Angel in Section III. By comparing the implementations, we identify two major bottlenecks in the execution of MLlib and propose MLlib* that addresses both bottlenecks in Section IV. We compare performance of different systems in Section V. We summarize related work in Section VI, and conclude in Section VII.

II. Preliminary

In this section, we provide a short review of GD and its distributed implementations, which will be the major focus of this paper. We present GD in the context of training GLMs, though GD has much wider applications such as training deep neural networks.

A. Gradient Descent

A common setting when training GLMs is the following. Given a linear classification task with X representing the input data, find a model w that minimizes the objective function

$$f(w, X) = l(w, X) + \Omega(w). \quad (1)$$

Here, $l(w, X)$ is the *loss function*, which can be 0-1 loss, square loss, hinge loss, etc. $\Omega(w)$ is the regularization term to prevent overfitting, e.g., L1 norm, L2 norm, etc.

Gradient descent (GD) is an algorithm that has been widely used to train machine learning models that optimize Equation 1. In practice, people usually use a variant called mini-batch gradient descent (MGD). We present the details of MGD in Algorithm 1.

Here, T is the number of iterations, η is the learning rate, and w_0 is the initial model. As illustrated in Algorithm 1, MGD is an iterative procedure. It repeats the following steps in each iteration until convergence: (1) Sample a batch of the training data X_B; (2) Compute the gradient of Equation 1 using X_B and the current model w_{t-1}; (3) Use the gradient to update the model.

Algorithm 1: MGD $\{T,\ \eta\ w_0,\ X\}$

for *Iteration* $t = 1$ **to** T **do**
  Sample a batch of data X_B;
  Compute gradient as $g_t = \sum_{x_i \in X_B} \nabla l(x_i, w_{t-1})$;
  Update model as $w_t = w_{t-1} - \eta \cdot g_t - \eta \cdot \nabla\Omega(w_{t-1})$;

The executions of GD and SGD (stochastic gradient descent [11], another popular variant of GD) are similar. Essentially, GD and SGD can be considered as special cases of MGD. Specifically, when the batch size is the entire data (i.e., $X_B = X$), it is GD; when the batch size is 1, it is SGD. Without loss of generality, we focus our discussion on MGD.

B. Distributed MGD

Sequential execution of MGD is usually not feasible for large datasets and models. People have been proposing various ways of running MGD in a distributed manner. While the details of these proposals differ, we can use a generic architecture to capture the essence of these proposals, as presented in Algorithm 2. In this generic architecture, there is a *master* to partition data and schedule tasks. There are multiple *workers*, each dealing with an individual partition. In addition, there is a *central node* to aggregate the gradients/models received from the workers.

As shown in Algorithm 2, the master first splits data into multiple partitions. It then schedules each worker to load a partition and launch a training task. Each worker can be implemented in two alternative ways:

- (`SendGradient`) Each worker pulls the latest model from the central node. It then samples a batch from its local data, computes the gradient using the latest model, and sends the gradient to the central node. The central node aggregates the gradients received from the workers and updates the model.
- (`SendModel`) Each worker performs MGD over its local partition of the data and sends the updated (local view of the) model to the central node. The central node then updates the (global) model based on updates received from the workers, using methods such as *model averaging* [9].

The difference between the two paradigms lies in the number of updates to the (global) model within *one single communication step* between the workers and the central node. If $T' = 1$, i.e., only one iteration is allowed in MGD, the number of updates made by `SendGradient` and `SendModel` will be exactly the same. However, if $T' \gg 1$, which is the typical case, `SendModel` will result in much more updates and thus much faster convergence.

III. Analysis of Existing systems

In this section, we provide detailed analysis of existing distributed machine learning systems based on the generic architecture in Algorithm 2. We focus on anatomizing the implementation of MGD in Apache Spark (MLlib) and specialized systems based on the parameter-server architecture (Petuum and Angel) from two aspects, i.e., (i) system architecture and (ii) algorithm implementation. We also briefly discuss other relevant systems, such as TensorFlow.

Algorithm 2: Distributed MGD $\{T,\ \eta\ w_0,\ X,\ m\}$

Master:
Issue LoadData() to all workers;
Issue InitialModel(w_0) to the central node;
for *Iteration* $t = 0$ **to** T **do**
  Issue WorkerTask(t) to all workers;
  Issue ServerTask(t) to the central node;

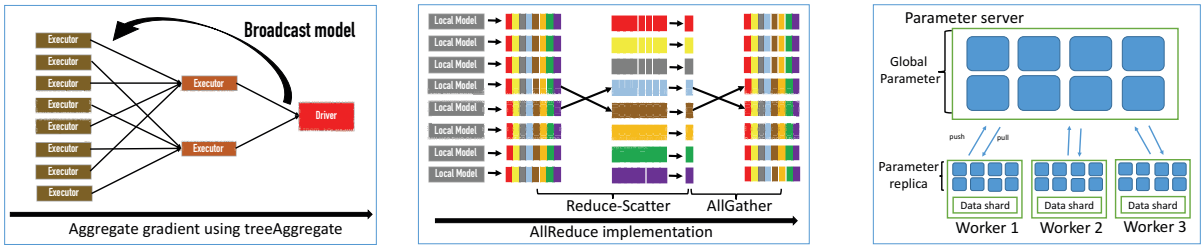
Worker r = 1, ..., m:
Function *LoadData():*
  Load a partition of data X_r;
Function *WorkerTask(t):*
  Get model w_{t-1} from the central node;
  if `SendGradient` **then**
    Sample a batch of data X_{br} from X_r;
    Compute gradient $g_t^r \leftarrow \sum_{x_i \in X_{br}} \nabla l(w_{t-1}, x_i)$;
    Send gradient g_t^r to the central node;
  else if `SendModel` **then**
    Compute model w_t^r via MGD(T', η, w_{t-1}, X_r);
    // T' is the number of iterations inside each worker.
    Send local model w_t^r to the central node;

Central node:
Function *InitialModel(w_0):*
  Initialize model as w_0;
Function *ServerTask(t):*
  if `SendGradient` **then**
    Aggregate gradient as $g_t \leftarrow \sum_{r=1}^{m} g_t^r$;
    Update the model as
    $w_t \leftarrow w_{t-1} - \eta \cdot (g_t) - \eta \cdot \nabla\Omega(w_{t-1})$;
  else if `SendModel` **then**
    Aggregate the models as $w_t \leftarrow f(\sum_{r=1}^{m} w_t^r)$;

A. Apache Spark and MLlib

Apache Spark [1] is a powerful framework for data analytics that has been adopted by enterprises across a wide range of industries, with attractive features such as fault tolerance and interoperability with the Hadoop ecosystem. Spark can cache the whole data in memory, so it fits well for iterative machine learning workloads.

MLlib [2] is one of the most popular machine learning libraries built on top of Spark, which uses MGD to train generalized linear models. Conceptually, the execution of MGD in MLlib can be outlined in Figure 2(a). There is a *driver* and multiple *executors*. The driver plays the role of both the master and the central node in Algorithm 2, which is responsible for scheduling the executors and maintaining the (global) model. The executors serve as the workers in Algo-



(a) MLlib on Spark (b) AllReduce implementation using shuffle on Spark (c) ML on Parameter Server

Fig. 2. Communication patterns of machine learning on Spark and Parameter Server.

rithm 2, which perform local computation over the partitioned data. The execution of MGD leverages the `SendGradient` alternative: (1) The driver first broadcasts the current model to the executors; (2) Each executor computes the gradients using its local data based on the received model, and sends the gradients to the driver; and (3) The driver aggregates gradients harvested from the executors and updates the model.

The driver can be overloaded when there are many executors and the model is large. To alleviate the workload of the driver, MLlib implements the aggregation of gradients following a hierarchical, distributed style. As depicted in Figure 2(a), the driver first employs some of the executors to perform local aggregation. It then pulls the aggregated (i.e., sum of) gradients from these intermediate executors. This hierarchical dispatch of aggregation is called `treeAggregate` in MLlib.

B. Parameter Servers

Figure 2(c) presents a typical parameter-server architecture [12] based machine learning system. Unlike MLlib where a single driver is responsible for maintaining the (global) model, in parameter-server architecture the model is stored across multiple machines called *parameter servers*. In other words, there are multiple nodes that serve the role of the central node in Algorithm 2. Moreover, unlike the Bulk Synchronous Parallel (BSP) execution model that backs up the implementation of Spark, workers can communicate with parameter servers asynchronously. Parameter servers can leverage different consistency controllers to implement different communication schemes such as BSP, SSP (acronym for "Stale Synchronous Parallel"), and ASP (acronym for "Asynchronous Parallel"), by enabling or disabling requests from workers. It has been shown that asynchronous communication can be beneficial for distributed machine learning [13].

In the following, we briefly review two instances of the parameter-server architecture: Petuum [7] and Angel [5].

1) Petuum: Petuum is one of the state-of-the-art distributed machine learning systems with the parameter-server architecture, implemented in C++. Unlike MLlib that follows the `SendGradient` paradigm, Petuum instead adopts the `SendModel` alternative in Algorithm 2 to train GLMs. That is, the workers in Petuum will update local model immediately after computing the gradients and send the updates to the parameter servers. In more detail, the workflow for each iteration is: (1) Workers pull the latest model from parameter servers, and update the model using their local data; (2) Afterwards, workers send their local updates of the model to parameter servers; (3) Finally, the parameter servers sum up all the updates from the workers.

The local computation that each worker of Petuum performs depends on whether the regularization term in Equation 1 is zero or not — L2 regularization will result in dense updates to the model that can be expensive [14]. If the regularization term is zero, Petuum workers instead conduct parallel SGD inside each batch; each communication step between workers and parameter servers therefore actually contains many local updates to the model. If, on the other hand, there is a nonzero regularization term, workers perform gradient descent over the batch data in each iteration. Each communication step thus contains only one update to the model. When updating the model, the parameter servers sum up the model updates from the workers. We refer to this model aggregation technique as *model summation* in the rest of this paper.

2) Angel: Angel [5] is another representative instance of distributed machine learning systems based on the parameter-server architecture, implemented in Java. Angel can read data directly from HDFS and run on Yarn clusters.

Angel also adopts the `SendModel` paradigm to train GLMs. It uses parameter servers to maintain the global model and uses multiple workers to compute local updates of the model. The difference between Angel and Petuum on training GLMs comes from two aspects. First, the frequency of communication is different. Workers in Angel communicate with the parameter servers *per epoch*, whereas workers in Petuum communicate with the servers *per batch*. Second, the local computation performed on one batch of data is different. Angel always performs gradient descent on each batch whereas the implementation of Petuum depends on the regularization term, as we have described above.

C. Other Systems

Clearly, MGD has been implemented in many other distributed machine learning systems, in particular systems developed for deep learning, such as TensorFlow, as GD is essential for the backpropagation algorithm that trains deep neural networks. However, we decide to not include these systems in this study because it has been pointed out in the literature that TensorFlow is slower than Petuum and Angel when training GLMs because of too many abstractions,

which will significantly raise system complexity as well as runtime overhead [5], [6]. Furthermore, TensorFlow does not provide mechanism for partitioning models. This also leads to inefficiency when handling large models in GLMs.

IV. MLlib*: Understanding and Improving Performance of MLlib

Based on our analysis in the previous sections, the implementation of MGD in MLlib is clearly not optimal. By using the `SendGradient` paradigm instead of the `SendModel` paradigm that has been widely implemented in, e.g., parameter servers, MLlib is likely to suffer from a much slower convergence within the same number of communication steps. This inferior performance has been noticeably documented in the literature [6]. A natural follow-up question is whether this apparently flawed implementation can be improved within the BSP framework that Spark is based on.

In this section, we identify performance bottlenecks of MLlib and study state-of-the-art, well-known techniques that can significantly improve the performance of MLlib when running MGD. Our implementations of these techniques piggyback on the existing Spark primitives and only require minor changes to the current MLlib implementation.

A. Bottlenecks in MLlib

We start by giving a more detailed analysis to understand bottlenecks in MLlib. We ran MGD to train a linear support vector machine (SVM) using the `kdd12` dataset described in Table I. The experiment was conducted on a cluster of nine nodes with one node serving as the driver and the others serving as the executors in Spark (recall Figure 2(a)). [1] Figure 3(a) presents the *gantt chart*[2] that tracks the execution of the nodes. The x-axis represents the elapsed time (in seconds) since the start of the execution. The y-axis represents the activities of the driver and the eight executors as time goes by. Each colored bar in the gantt chart represents a type of activity during that time span that is executed in the corresponding cluster node (i.e., driver or executor), whereas different colors represent different types of activities.

We can identify two obvious performance issues by examining the gantt chart in Figure 3(a):

- **(B1)** Bottleneck at the *driver* — at every stage when the driver is executing, the executors have to wait.
- **(B2)** Bottleneck at the *intermediate aggregators*, i.e., the executors performing intermediate aggregations of gradients — at every stage when these executors are running, the other nodes have to wait.

As discussed in Section III-A, MLlib uses the `SendGradient` paradigm. The bottleneck at the driver is therefore easy to understand: the executors simply cannot proceed because they have to wait for the driver to finish updating the model. Moreover, the bottleneck at the intermediate aggregators is also understandable due to the hierarchical aggregation mechanism employed by MLlib, although it shifts some workload from the driver — the latency at the driver can be even worse without this hierarchical scheme.

[1] We assign one task to each executor because when we increase the number of tasks per executor, the time per iteration increases due to the heavy communication overhead.

[2] https://en.wikipedia.org/wiki/Gantt_chart

B. Implementation of MLlib*

In MLlib*, we use two well known techniques to deal with the two bottlenecks in MLlib's implementation: (1) model averaging and (2) distributed aggregation. We now discuss them in detail.

1) Model Averaging: Model averaging is the essential reason for the efficiency of the `SendModel` paradigm in Algorithm 2. If we can implement model averaging in MLlib, the driver will remain as a bottleneck but much less frequently, due to the reduced number of communication steps between the driver and the executors.

Most part of our implementation is quite straightforward. We basically replace the computation of gradients in each executor by model updates, and change the data being sent from gradients to model updates, too. However, `SendModel` can be inefficient when the regularization term (typically L2 norm) is not zero. In this case, frequent updates to the local view of the model can be quite expensive when the model size is large. To address this, we use a threshold-based, lazy method to update the models following Bottou [14]. Our implementation does not require any change to the core of Spark. Instead, we implement our techniques leveraging primitives provided by Spark.

Figure 3(b) presents the gantt chart of MLlib after incorporating our implementation of `SendModel`, by rerunning the experiment described in Section IV-A. One can observe that it is very similar to Figure 3(a). It implies that our implementation does not impact *per-node* computation time much. This is not surprising, as the computational tasks of `SendModel` are similar to those of `SendGradient` — it is just computing weights of the model versus computing gradients! Nonetheless, the number of stages in Figure 3(b) should be much smaller comparing with Figure 3(a) if we extend the x-axes of both charts to the time of convergence, which suggests a much faster convergence of `SendModel`.

Remark: Note that model averaging is just one of the model aggregation schemes that can be adopted in the `SendModel` paradigm. For example, Petuum instead leverages the model summation scheme, which simply sums up but does not take the average of the received model updates. As was pointed out by Zhang and Jordan [15], there are pros and cons between model averaging and model summation. Roughly speaking, model summation can lead to potential divergence; however, when it converges, it can converge faster than model averaging. They actually proposed a technique that further improves the convergence rate of model averaging by "reweighting" the samples taken when combining the model updates. Implementing their techniques may therefore further improve the performance of MLlib*. However, as we will see, even with the current implementation, the performance of

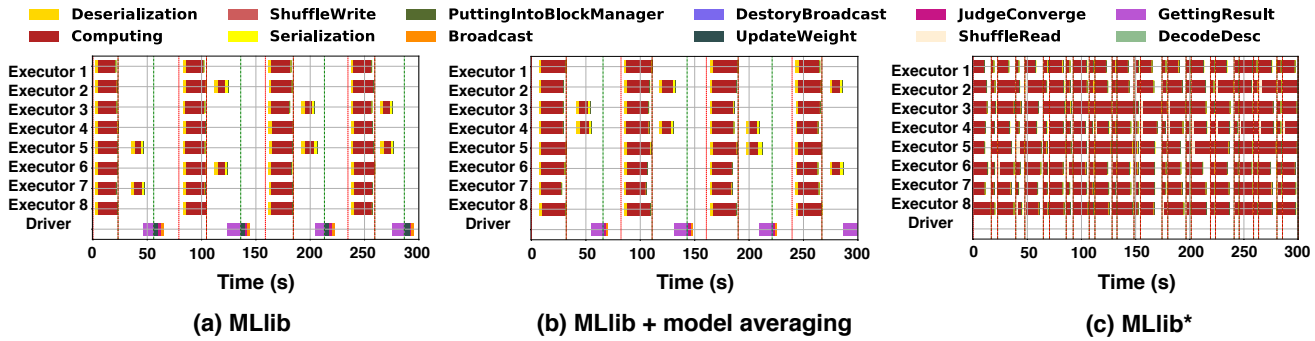


Fig. 3. Gantt charts for MGD executions in MLlib, MLlib with model averaging and MLlib*. A red vertical line represents the start of a stage in Spark, whereas the subsequent green vertical line represents its end.

MLlib* is already comparable with state-of-the-art distributed machine learning systems when training GLMs.

2) Distributed Aggregation using AllReduce: The communication pattern exhibited in Figure 3(b) remains the same as that in Figure 3(a): Even if we now aggregates the (weights of) the models instead of the gradients, we still follow the hierarchical aggregation in MLlib using the `treeAggregate` function. This, however, turns out to be unnecessary. Recall the communication pattern in `SendGradient`: executors send the gradients to the driver; the driver sends the updated model back to the executors. However in MLlib*, the communication pattern remains the same but the gradients are replaced by the models. Because each executor is in charge of its local model, it seems redundant to have the driver to first collect the (aggregated) model updates from the executors and then broadcast the model updates back to the executors.

The basic idea is to partition the global model and let each executor *own* one partition.[3] The owner of a model partition is responsible for its maintenance (using model averaging). Note that the ownership is *logical* rather than physical: Each executor still stores a physical copy of the current version of the model (which includes all partitions), but it only performs model averaging over the logical partition it owns. In this way, the executors no longer need to wait for the driver to broadcast the averaged model. In other words, the driver no longer needs to play the role of the central node in Algorithm 2. In fact, there will be no notion of a specific central node in the distributed model-averaging architecture we shall propose. Moreover, given that we do not need the driver to take charge of model averaging, the hierarchical aggregation scheme presented in Figure 2(a) is also not necessary. As a result, we can have all the executors participate in the distributed maintenance of the global model simultaneously and homogeneously. There are two main technical challenges in our design. First, updates to a partition of the global model can be scattered across the executors. For example, in Figure 2(b), we have eight executors E_1 to E_8, each owning one partition, that is 1/8 of the global model M. Let us number those partitions from M_1 to M_8. Consider the partition M_1. Although it is owned by E_1, updates to M_1 can come from all E_1 to E_8, because data and model are partitioned independently. Data points that can contribute to the weights of M_1 (i.e., data points with nonzero feature dimensions corresponding to those weights) can be located on all the executors. To perform model averaging over a local partition, the owner has to collect updates to this partition from all the other executors. Second, to compute the model updates (for local views of all model partitions) as in the `SendModel` paradigm, an executor has to compute the gradients, which depend on not only the local data points but also the latest version of the *entire* global model (not just the local partition of the model), again due to the "inconsistency" between data partitioning and model partitioning.[4] We use a two-phase procedure to address these two challenges (see Figure 2(b) and Algorithm 3 for illustration):

- (`Reduce-Scatter`) In the first stage, after each executor has done with updating its model locally, it sends partitions other than the one it owns to their owners, respectively. Continuing with our previous example, E_1 updates its local copies of M_1 to M_8, and sends the updated versions of M_2, ..., M_8 to E_2, ..., E_8, respectively. Afterwards, each executor has received all updates to the partition it owns — it can then perform model averaging for the partition.
- (`AllGather`) In the second stage, after each executor finishes model averaging over the partition it owns, it broadcasts that partition to everyone else. Again, using the previous example, E_1 sends M_1 (after finishing model averaging) to E_2, ..., E_8. Afterwards, each executor now has refreshed versions of all partitions of the global model. This stage is motivated by the work of Thakur et al. [16].

Again, our implementation does not require changes to the core of Spark. Specifically, we use the `shuffle` operator

[3]It is also possible to have one executor own multiple partitions. However, for ease of exposition, we assume that each executor just owns one partition.

[4]One could in theory avoid this inconsistency issue by carefully partitioning data points based on their nonzero feature dimensions and then partitioning the model with respect to the data partitions. However this is data dependent and is difficult to achieve in practice due to issues such as data skew — one may end up with too many partitions with a highly skewed distribution of partition sizes. Moreover, data need to be randomly shuffled and distributed across the workers.

Algorithm 3: Distributed MGD $\{T,\ \eta\ \ w_0,\ X,\ m\}$ in MLlib*

Master:
Issue LoadData() to all workers;
Issue InitialModel(w_0) to all workers;
for *Iteration* $t = 0$ **to** T **do**
  Issue UpdateModel() to all workers;
  Issue Reduce-Scatter() to all workers;
  Issue AllGather() to all workers;

Worker r = 1, ..., m:
Function *LoadData():*
  Load a partition of data X_r;
Function *InitialModel(w_0):*
  Initial local model as w_0;
Function *UpdateModel():*
  // We assume local model is w^r;
  for *each data point* x *in* X_r **do**
    Compute gradient: $g^r \leftarrow \nabla l(w^r, x)$;
    Update model: $w^r \leftarrow w^r - \eta \cdot g^r - \eta \cdot \nabla\Omega(w^r)$;
Function *Reduce-Scatter():*
  // Break the model into pieces and shuffle them.
  Partition local model w^r into m pieces, namely $w_1^r, w_2^r, ..., w_m^r$;
  for $i = 1$ **to** m **do**
    Send partition w_i^r to worker i;
  // Perform model averaging for partition r
  // (after receiving updates from all other workers).
  $\overline{p^r} \leftarrow \frac{1}{m}\sum_{j=1}^{m} w_r^j$;
  // The size of $\overline{p^r}$ is $1/m$ of the size of whole model w^r.
Function *AllGather():*
  // Send $\overline{p^r}$ to all workers.
  for $i = 1$ **to** m **do**
    Send $\overline{p^r}$ to worker i;
  // Concatenate partitions from all the workers in order.
  $w^r \leftarrow (\overline{p^1}, ..., \overline{p^m})$;

in Spark to implement both stages: One can write different shuffling strategies by specifying the source(s) and destination(s) of each partition.[5] Figure 3(c) presents the gantt chart of MLlib* when repeating the previous experiment. As expected, all executors are now busy almost all the time without the need of waiting for the driver. By just looking at Figure 3(c), one may wonder if this is a correct BSP implementation. For example, in the first communication step, it seems that E1 started its `AllGather` phase before E8 finished its `Reduce-Scatter` phase, which should not happen in a BSP implementation. We note here that this is just an illusion: E8 was the slowest worker in the first communication step, and therefore its `AllGather` phase immediately started after its `Reduce-Scatter` phase — there is no visible gap shown on the gantt chart. In other words, all workers started their `AllGather` phases at the same timestamp, i.e., the first vertical line in Figure 3(c).

It is worth to point out that, while the gantt chart in Figure 3(c) looks much more cluttered compared with Figure 3(b), the actual amount of data exchanged within each communication step actually remains the same: If we have k executors and the model size is m, then the total amount of data communicated is $2km$ for both cases.[6] This may seem puzzling as one may expect that the two rounds of shuffling we employed in MLlib* would significantly increase the data exchanged. This is, however, just an illusion. In both scenarios, the global model is exactly sent and received by each executor twice. The net effect is that a communication step (with two rounds of shuffling) in MLlib* can finish the same number of model updates as a step in the "MLlib + model averaging" mechanism can but the latency is much shorter.

As a side note, the names of the two phases, `Reduce-Scatter` and `AllGather`, are borrowed from MPI (acronym for "Message Passing Interface") terminology, which represent MPI operators/primitives with the same communication patterns plotted in Figure 2(b). Moreover, the entire communication pattern combining the two stages is akin to `AllReduce`, another MPI primitive. We refer readers to the work by Thakur et al. [16] for more details about these MPI primitives.

V. Experimental Comparison

In this section, we compare MLlib* and other systems by conducting an extensive experimental evaluation using both public datasets and Tencent datasets. Our goal is not only to just understand the performance improvement of MLlib* over MLlib, but we also want to understand where MLlib and MLlib* stand in the context of state-of-the-art distributed machine learning systems. Although these specialized systems have claimed to be much better than MLlib, the reported results were based on different experimental settings or even different machine learning tasks. We are not aware of any previous study with similar level of completeness, and we hope our results can offer new insights to developing, deploying, and using distributed machine learning systems.

Before we present the details, we summarize our results and observations as follows:

- We find that machine learning systems based on parameter servers, Petuum and Angel, do significantly outperform MLlib, as was documented in the literature.
- By breaking down the improvements from the two techniques we used, i.e., *model averaging* and *distributed aggregation*, we observe a significant speedup of MLlib* over MLlib.
- We futher show that MLlib* can achieve comparable and sometimes better performance than Petuum and Angel that are based on parameter servers.

[5] https://0x0fff.com/spark-architecture-shuffle/

[6] We ignore the intermediate aggregators in Figure 3(b).

A. Experimental Settings

Clusters. We used two different clusters in our experiments:

- *Cluster 1* consists of 9 nodes (connected with a 1-Gbps network), where each node is configured with 2 CPUs and 24 GB of memory. Each CPU has 8 cores.
- *Cluster 2* consists of 953 nodes, with 345 TB of memory in total. Each node contains 2 CPUs with 10 cores each. The nodes are connected with a 10-Gbps network.

Workloads. We evaluate different machine learning systems on training generalized linear models. Specifically, we train support vector machine (SVM) on five datasets, with and without L2 regularization.[7] We use four public datasets from MLBench [17][8], whereas the dataset `WX` is obtained from Tencent. Table I presents the statistics of these datasets.

The diversity of these datasets lies in the following two aspects. First, the dimensions of the features differ: the datasets `avazu` and `url` have relatively lower dimensions, whereas the datasets `kddb`, `kdd12`, and `WX` have higher dimensions. Second, the datasets `avazu`, `kdd12`, and `WX` are determined, whereas the datasets `url` and `kddb` are underdetermined (i.e., there are more features than data points). For the case of training GLMs, the diversity presented in these datasets offers good chance to probe and understand the strength and weakness of different systems.

Participating Systems and Configurations. We compare four distributed machine learning systems in our evaluation: (1) Petuum 1.1, (2) Angel 1.2.0, (3) Spark MLlib 2.3.0, (4) MLlib*. TensorFlow is not included because it is reported to be slower in existing studies [5], [6]. To ensure fairness when comparing different systems, we tune the configuration of each system in our best effort. For example, we tuned all parameters specified in the official guidelines for tuning Spark, such as the number of tasks per core, serialization method, garbage collection, etc.[9]

Metrics. We measure the value of $f(w, X)$ in Equation 1 as time elapses, since model training aims for minimizing the objective function. Note that the comparison is *fair* in the context of training GLMs: All participating systems should eventually converge to the same (global) minimum as the objective function is *convex*. This is another reason for us to focus on training GLMs in this paper. In addition to the elapsed time taken by each system towards convergence, we also measure the number of communication steps when comparing MLlib and MLlib*. Speedup is calculated when the accuracy loss (compared to the optimum) is 0.01.

Hyperparameter Tuning. For each system, we also tune the hyper-parameters by grid search for fair comparison. Specifically, we tuned batch size, learning rate for Spark MLlib. For Angel and Petuum, we tuned batch size, learning rate, as well as staleness.

[7]SVM is a representative for GLMs. In fact, linear models share similar training process from a system perspective.

[8]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

[9]http://spark.apache.org/docs/latest/tuning.html

Dataset	#Instances	#Features	Size
`avazu`	40,428,967	1,000,000	7.4GB
`url`	2,396,130	3,231,961	2.1GB
`kddb`	19,264,097	29,890,095	4.8GB
`kdd12`	149,639,105	54,686,452	21GB
`WX`	231,937,380	51,121,518	434GB

TABLE I
DATASET STATISTICS.

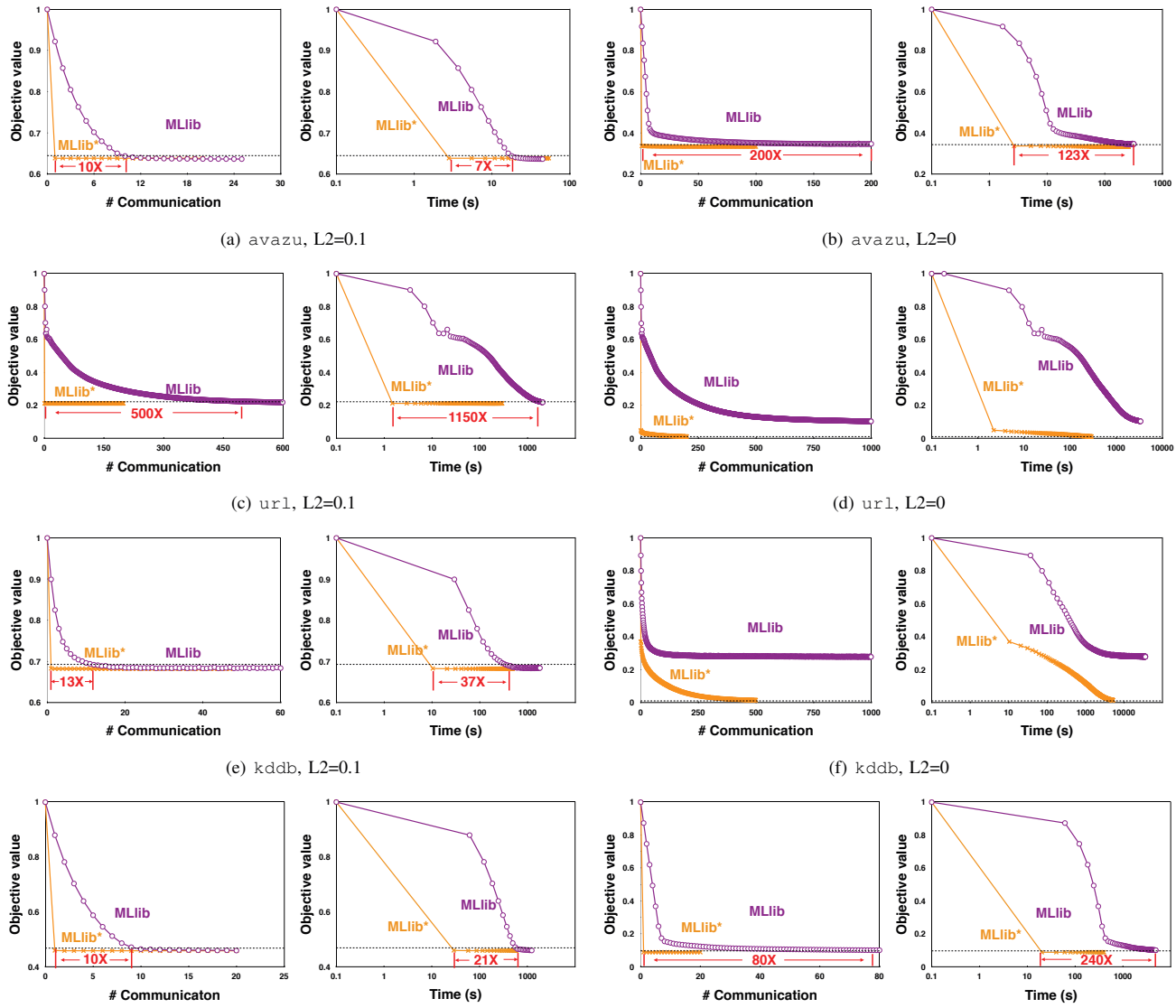
B. Evaluation on Public Datasets

We conduct extensive experimental evaluation of MLlib* using the four public datasets in Table I with the following goals in mind:

- Revisit results in previous work regarding the performance gap between MLlib and parameter servers.
- Study improvement of MLlib* over MLlib brought by model averaging and distributed aggregation.
- Compare MLlib* and parameter servers.

1) MLlib vs. MLlib:* Figure 4 compares MLlib* and MLlib using the four public datasets. We trained SVMs with and without L2 regularization. In each subfigure, the left plot shows the change on the value of the objective function as the number of communication steps increases, and the right plot shows that change corresponding to the elapsed time.

We can observe several facts. First, compared to MLlib, MLlib* converges much faster. As Figure 4(h) indicates, MLlib needs 80× more steps upon convergence on `kdd12` dataset, when L2 regularization is omitted. (Note that the x-axis is in logarithmic scale.) This demonstrates the significant improvement of the `SendModel` paradigm over the `SendGradient` paradigm used by MLlib — notice that the second technique employed by MLlib*, i.e., AllReduce implementation, does not change the number of communication steps. Furthermore, we note that the overall speedup is more than linear: the convergence of MLlib* is 240× instead of 80× faster if the speedup were just linear. This extra speedup is attributed to the AllReduce technique implemented in MLlib*. It is a bit surprising at a first glance — one may not expect that the improvement from AllReduce is actually more significant than `SendModel`. This essentially implies that the computation workload at the driver node is really a big deal, regardless of it is aggregation of gradients or models. Of course, the severity of the bottleneck depends on the sizes of the data and the model — the larger they are the worse the bottleneck is. For example, as shown in Figure 4(b), MLlib needs 200× more iterations to converge while is only 123× slower than MLlib*. This implies that time spent on each iteration of MLlib* is actually longer than that of MLlib. There are two reasons. First, the batch size of MLlib is significantly smaller than the dataset size. Typically, the batch size is set as 1% or 0.1% of the dataset by grid search. On the other hand, MLlib* needs to pass the entire dataset in each iteration. As a result, the computation overhead per iteration of MLlib* is larger. Second, the model size of `avazu` is smaller than that of `kdd12`, by 54×. Therefore, the communication overhead on the driver in MLlib is less severe and the benefit from using AllReduce in MLlib* is smaller.



(a) avazu, L2=0.1 (b) avazu, L2=0

(c) url, L2=0.1 (d) url, L2=0

(e) kddb, L2=0.1 (f) kddb, L2=0

(g) kdd12, L2=0.1 (h) kdd12, L2=0

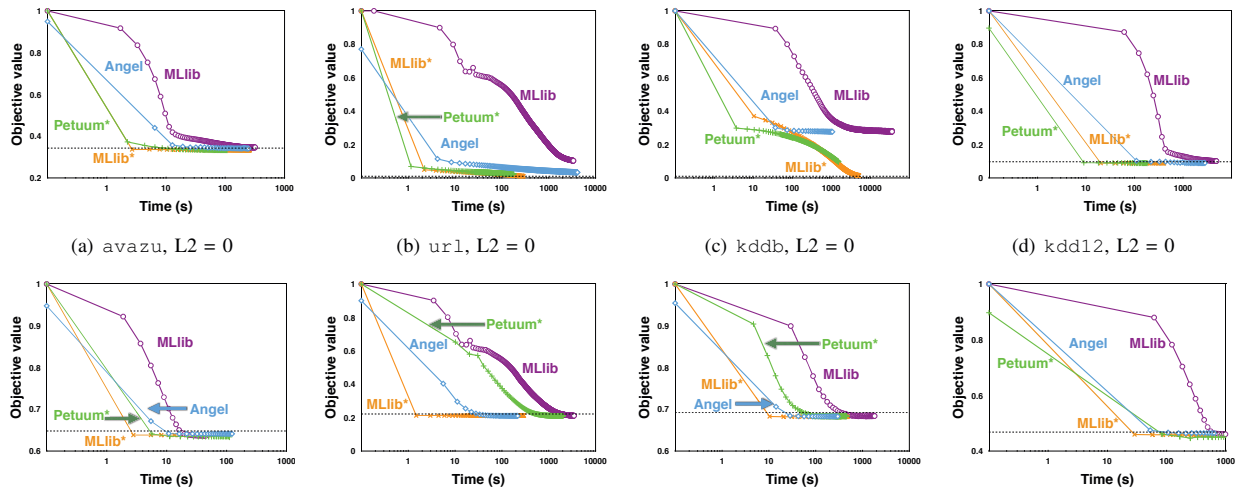
Fig. 4. Comparison of MLlib and MLlib* on four datasets with and without L2 regularization. The dotted line in each figure represents 0.01 accuracy loss.

Second, MLlib performs worse when the problem becomes more ill-conditioned. As shown in Figures 4(b), 4(d), 4(f), and 4(h), MLlib converges 123× and 200× slower than MLlib* on the two determined datasets avazu and kdd12, while they cannot get to the optimal loss even after 1,000 iterations on the two underdetermined datasets url and kddb. To make the problem less ill-conditioned, we also report the results with L2 regularization equal to 0.1 on these four datasets in Figures 4(a), 4(c), 4(e), and 4(g), respectively. We can observe that the performance gap between MLlib and MLlib* becomes smaller when the training objective becomes more determined. For example, the speedups decrease to 7× and 21× on avazu and kdd12, respectively. Meanwhile, on url and kddb, MLlib can now converge to the same loss as MLlib*.

Third, distributed aggregation is more beneficial for large models. As we can infer from comparing Figure 4(e) with Figure 4(a), the speedup per iteration of MLlib* over MLlib on high dimensional dataset like kddb is more significant than that on low dimensional dataset like avazu.[10] Distributed aggregation can distribute the communication overhead on the driver evenly to all the executors. Furthermore, the speedup per iteration on kdd12 is slightly worse than that on url, because the time spent on each iteration consists of two parts, computation and communication. The computation overhead on kdd12 is heavier as kdd12 contains more data points than url (see Table I).

2) MLlib vs. Parameter Servers:* Figure 5 compares the performance of MLlib* with Petuum* and Angel over the four

[10]The speedup per iteration is computed by dividing the elapsed time (the right plot of each figure) by the number of iterations (the left plot).



(a) avazu, L2 = 0 (b) url, L2 = 0 (c) kddb, L2 = 0 (d) kdd12, L2 = 0

(e) avazu, L2 = 0.1 (f) url, L2 = 0.1 (g) kddb, L2 = 0.1 (h) kdd12, L2 = 0.1

Fig. 5. Comparison of MLlib* and parameter servers on different datasets with and without L2 regularization. The dotted line in each figure represents 0.01 accuracy loss.

datasets, with and without L2 regularization. Here, Petuum* is a slightly tweaked implementation of Petuum. The original implementation of Petuum uses model summation instead of model averaging, which has been pointed out to be problematic [15], [18] — it suffers from potential divergence. We therefore replaced model summation in Petuum by model averaging and call this improved version Petuum* — we find that model averaging is always faster than model summation based on our empirical study. As a reference pointer, we also present the performance of MLlib.

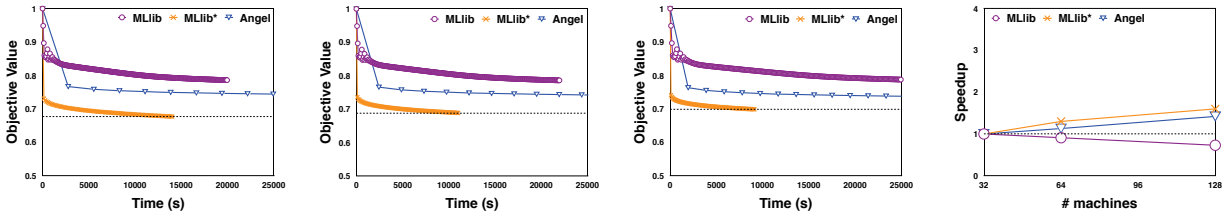
We have the following observations. First, Figure 5 confirms that MLlib can be significantly slower than Petuum* and Angel, as evidenced by previous studies [5]–[7]. As was analyzed in Section III, both Petuum* and Angel employ the `SendModel` paradigm in Algorithm 2 and therefore are understandably more efficient than the `SendGradient` paradigm used by MLlib.

Second, as Figures 5(a), 5(b), 5(c), and 5(d) indicate, MLlib* can achieve comparable or better performance as those of Petuum* and Angel, when L2 regularization vanishes. Specifically, MLlib* and Petuum* have similar performance because both of them converge fast: They both perform parallel SGD and model averaging. The performance may be slightly different because of some implementation issues. For example, Petuum* is implemented in C++ while MLlib* is implemented using Scala. Also, Petuum* uses SSP to alleviate potential latency from stragglers. On the other hand, MLlib* is faster than Angel, because Angel cannot support small batch sizes very efficiently due to flaws in its implementation. Roughly speaking, Angel stores the accumulated gradients for each batch in a separate vector. For each batch, we need to allocate memory for the vector and collect it back. When the batch size is small, the number of batches inside one epoch increases because Angel workers communicate with parameter servers every epoch, i.e., it needs more vectors to store the gradients every epoch. Hence, there will be significant overhead on memory allocation and garbage collection.

Third, MLlib* is faster than both Petuum* and Angel when L2 regularization is nonzero on the four datasets, as shown in Figures 5(e), 5(f), 5(g), and 5(h). Sometimes the performance gap between MLlib* and parameter servers is quite significant, for example, on the `url` and `kddb` datasets as shown in Figures 5(f) and 5(g). Moreover, Angel outperforms Petuum* (also significantly on the `url` and `kddb` datasets). We note down a couple of implementation details that potentially explain the performance distinction. When the L2 regularization is not zero, each communication step in Petuum* contains only one update to the model, which is quite expensive. In contrast, workers in Angel can communicate with servers once *per epoch* (i.e., a pass of the entire dataset) — they only need to update their local models at every batch without pinging the servers. As a result, each communication step in Angel contains many more updates to the global model, which, as we have seen several times, can lead to much faster convergence. Meanwhile, in MLlib* when L2 regularization is nonzero, it actually performs parallel SGD (i.e., with batch size 1) with a lazy, sparse update technique designed for SGD [14], which can boost the number of updates to the model per communication step even further.

C. Evaluation on Tencent Datasets

To report how MLlib* performs in an industrial environment, we compare MLlib* with other systems on *Cluster 2* using the Tencent dataset (i.e., the `WX` dataset), which is orders of magnitude larger than the other datasets. Apart from the comparison of convergence, we also report the scalability results of different systems using `WX` dataset. The dataset cannot fit into a single machine's memory, therefore we performed scalability tests with 32, 64, and 128 machines. We use grid search to find the best hyperparameters for each participating system. We do not have results for Petuum*,



(a) #machines = 32 (b) #machines = 64 (c) #machines = 128 (d) #machines vs. speedup

Fig. 6. Comparison of MLlib*, MLlib and Angel on Tencent dataset. The dotted lines in Figures 6(a), 6(b) and 6(c) represent the best objective values achieved among the systems under the corresponding experimental settings. Figure 6(d) shows the speedups of these systems with respect to the number of machines, normalized by the time cost using 32 machines.

because the deployment requirement of Petuum is not satisfied on *Cluster 2*. Figure 6 shows the results.

First, Figure 6(a) demonstrates that MLlib* converges much faster than Angel and MLlib on the `WX` dataset when using 32 machines. The loss of Angel and MLlib is still decreasing, but they need much longer time. The reason is similar to what we have explained in Section V-B: Compared to MLlib and Angel, MLlib* contains many more updates to the global model in each communication step and the communication pattern is more efficient.

Second, the scalability in terms of the time spent on each epoch is poor for all these systems. Figures 6(a), 6(b), 6(c), and 6(d) show the convergence using different number of machines and the corresponding speedup. As we can see, when we increase the number of machines from 32 to 128, the speedups of all these systems are poor: Angel becomes $1.5\times$ faster and MLlib* becomes $1.7\times$ faster, and MLlib evens gets slower. This is far below the $4\times$ speedup one would expect if the scalability were linear.

There are two possible reasons for this poor scalability:

(1) When increasing the number of machines, the communication cost becomes more expensive and starts to dominate, although the computation cost on each machine decreases. We take MLlib as an example. MLlib adopts the `SendGradient` paradigm and the batch size we set is 1% of the full dataset via grid search. When increasing the number of machines from 32 to 128, the time cost per epoch even increases by $0.27\times$. Clearly, communication overhead starts to dominate the time cost. This is actually interesting — it indicates that using more machines may not always be a good choice.

(2) Workers in these systems need to synchronize every iteration and thus the elapsed time of each iteration is determined by the slowest worker — when the number of machines increases it is more likely to have a really slow worker show up, especially in a large and heterogeneous environment (e.g., *Cluster 2*) where the computational power of individual machines exhibits a high variance. One may argue that assigning multiple tasks to one executor (i.e., multiple waves of tasks) can reduce the overhead brought by BSP. However, this is not always true when it comes to distributed machine learning. We tuned the number of tasks per executor, and the result turns out that one task per executor is the optimal solution, due to heavy communication overhead.

VI. Related Work

We discuss some other related works in addition to the ones that have been covered in previous sections.

Tradeoff between computation and communication: To balance the tradeoff between computation and communication, there are serveral lines of work. The first is to determine how many machines to use, given a distributed workload. Using machines more than enough can increase the communication cost while using not enough machines can increase the computation cost on each machine. Following this line, McSherry [19] argues that distributed computing should at least beat the single machine implementation. Huang [20] uses as small number of machines as possible to ensure the performance and efficiency.

Second, there are many proposals on reducing communication cost by performing local computation as much as possible. For example, Grape [21] is a state-of-the-art distributed graph processing system, which tries to do as much computation as possible within a single machine and reduce the number of iterations in distributed graph processing. As another example, Gaia [22] is a geo-distributed machine learning system using parameter servers. It tries to reduce communication cost by favoring communications within local-area networks over wide-area networks. The parallel SGD and model averaging techniques in MLlib* falls into the same ballpark — it performs as many local model updates as possible within each single node, which significantly reduces the number of communication steps required. There are also some works on reducing the communication cost by partitioning the workloads for better load balance [23]–[25].

Parameter Server vs. AllReduce: In general, Parameter Server can be viewed as an architecture that manages a distributed shared memory hosting the machine learning model and supports flexible consistency controls for node communications. It provides primitives such as `pull` and `push`, which allow us to update part of the model (a-)synchronously using a user-defined consistency controller, such as BSP, SSP, and ASP. Parameter Server has become quite popular since its invention, due to its flexibility and superb performance.

Another popular architecture for distributed machine learning is `AllReduce` [16]. It is an MPI primitive, which first

aggregates inputs from all workers and then distribute results back (to all workers). We do not compare with systems based on `AllReduce`, because there is few system using `AllReduce` for training linear models.

Other Machine Learning Systems on Spark: Kaoudi [11] built a cost-based optimizer to choose the best GD plan for a given workload. In our work, we use grid search to find the best parameters for each workload and thus do not need the optimizer. Anderson [26] integrated MPI into Spark and offloads the workload to an MPI environment. Basically, they transfer the data from Spark to MPI environment, use high performance MPI binaries for computation, and finally copy the result back to HDFS for further usage. However, their system is not optimized for training GLMs in Spark.

VII. Conclusion

In this paper, we have focused on the Spark ecosystem and studied how to run machine learning workloads more efficiently on top of Spark. With a careful study over implementations of existing distributed machine learning systems, we identified two bottlenecks in Spark MLlib (i.e., *inefficient pattern of model update* and *inefficient pattern of communication*). Utilizing two state-of-the-art, well-known techniques (i.e, *model averaging* and *AllReduce*), we can significantly improve MLlib without altering Spark's architecture. In fact, MLlib*, our improved version of MLlib, not only outperforms MLlib, but also performs similarly to or better than state-of-the-art distributed machine learning systems that are based on parameter servers, such as Petuum and Angel, over both public and Tencent workloads that we have tested.

As next steps, we plan to open-source MLlib* and contribute it back to the Spark community if possible. Moreover, Spark recently introduced `spark.ml`, its second-generation machine learning library that implements L-BFGS [27], a popular second-order optimization algorithm. Unlike GD and its variants that only utilize first-order derivatives of the objective function, L-BFGS further uses second-order derivatives to help revise the search direction. An interesting question is whether the techniques we have developed for speeding up MLlib could also be used for improving `spark.ml`, though this is beyond the scope of the current paper.

VIII. Acknowledgement

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004403), NSFC(No. 61832001, 61702015, 61702016, 61572039), and PKU-Tencent joint research Lab. CZ and the DS3Lab gratefully acknowledge the support from Mercedes-Benz Research Development North America, Oracle Labs, Swisscom, Zurich Insurance, Chinese Scholarship Council, and the Department of Computer Science at ETH Zurich.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010.

[2] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, pp. 34:1–34:7, 2016.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *ODSI*, 2016, pp. 265–283.

[4] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *SIGKDD*, 2016, pp. 785–794.

[5] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, p. nwx018, 2017.

[6] K. Zhang, S. Alqahtani, and M. Demirbas, "A comparison of distributed machine learning platforms," in *ICCCN*, 2017, pp. 1–9.

[7] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *SIGKDD*, 2015, pp. 1335–1344.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[9] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *NIPS*, 2010, pp. 2595–2603.

[10] J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang, and G. Song, "Bigdl: A distributed deep learning framework for big data," *CoRR*, vol. abs/1804.05839, 2018.

[11] Z. Kaoudi, J. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal, "A cost-based optimizer for gradient descent optimization," in *SIGMOD*, 2017, pp. 977–992.

[12] M. Li, D. G. Anderson, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014, pp. 583–598.

[13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013, pp. 1223–1231.

[14] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade - Second Edition*, 2012, pp. 421–436.

[15] Y. Zhang and M. I. Jordan, "Splash: User-friendly programming interface for parallelizing stochastic algorithms," *CoRR*, vol. abs/1506.07552, 2015.

[16] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

[17] H. Zhang, L. Zeng, W. Wu, and C. Zhang, "How good are machine learning clouds for binary classification with good features?" *CoRR*, vol. abs/1707.09562, 2017.

[18] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *SIGMOD*, 2017, pp. 463–478.

[19] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *15th Workshop on Hot Topics in Operating Systems, HotOS'15*, 2015.

[20] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng, "Flexps: Flexible parallelism control in parameter server architecture," *PVLDB*, vol. 11, no. 5, pp. 566–579, 2018.

[21] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian, "Parallelizing sequential graph computations," in *SIGMOD*, 2017, pp. 495–510.

[22] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *NSDI*, 2017, pp. 629–647.

[23] M. Onizuka, T. Fujimori, and H. Shiokawa, "Graph partitioning for distributed graph processing," *Data Science and Engineering*, vol. 2, no. 1, pp. 94–105, Mar 2017.

[24] N. Xu, L. Chen, and B. Cui, "Loggp: A log-based dynamic graph partitioning method," *PVLDB*, vol. 7, no. 14, pp. 1917–1928, 2014.

[25] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, "Efficient breadth-first search on massively parallel and distributed-memory machines," *Data Science and Engineering*, vol. 2, no. 1, pp. 22–35, Mar 2017.

[26] M. J. Anderson, S. Smith, N. Sundaram, M. Capota, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between HPC and big data frameworks," *PVLDB*, vol. 10, no. 8, pp. 901–912, 2017.

[27] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Math. Program.*, vol. 45, no. 1-3, pp. 503–528, 1989.