The VLDB Journal (2021) 30:693–712
https://doi.org/10.1007/s00778-021-00664-7

REGULAR PAPER

Model averaging in distributed machine learning: a case study with Apache Spark

Yunyan Guo[1] · Zhipeng Zhang[2] · Jiawei Jiang[4] · Wentao Wu[3] · Ce Zhang[4] · Bin Cui[2] · Jianzhong Li[1]

Received: 3 December 2019 / Revised: 26 July 2020 / Accepted: 2 September 2020 / Published online: 15 April 2021

Abstract
The increasing popularity of Apache Spark has attracted many users to put their data into its ecosystem. On the other hand, it has been witnessed in the literature that Spark is slow when it comes to distributed machine learning (ML). One resort is to switch to specialized systems such as parameter servers, which are claimed to have better performance. Nonetheless, users have to undergo the painful procedure of moving data into and out of Spark. In this paper, we investigate performance bottlenecks of MLlib (an official Spark package for ML) in detail, by focusing on analyzing its implementation of stochastic gradient descent (SGD)—the workhorse under the training of many ML models. We show that the performance inferiority of Spark is caused by implementation issues rather than fundamental flaws of the bulk synchronous parallel (BSP) model that governs Spark's execution: we can significantly improve Spark's performance by leveraging the well-known "model averaging" (MA) technique in distributed ML. Indeed, model averaging is not limited to SGD, and we further showcase an application of MA to training latent Dirichlet allocation (LDA) models within Spark. Our implementation is not intrusive and requires light development effort. Experimental evaluation results reveal that the MA-based versions of SGD and LDA can be orders of magnitude faster compared to their counterparts without using MA.

Keywords Distributed machine learning · Apache Spark MLlib · Generalized linear models · Latent Dirichlet allocation

1 Introduction

The increasing popularity of Spark has attracted many users to put their data into its ecosystem [44]. However, Spark is generally believed to be slow when it comes to distributed machine learning (ML) [49]. This implies significant data movement overhead for machine learning users since they have to migrate their datasets from Spark to specialized systems such as TensorFlow [2] or XGBoost [11].

Nonetheless, it remains unclear *why* Spark is slow for distributed ML. Previous work mainly attributes this inefficiency to the architecture that Spark adopts. Spark is architected based on the classic bulk synchronous parallel (BSP) model, where execution is divided into stages and each stage employs multiple *worker* nodes and a *driver* node, which is responsible for coordination and synchronization of workers. The driver node can be a bottleneck when training large ML models, due to the overwhelming communication overhead between the workers and the driver. Nonetheless, is

The work of the paper was performed when the first two authors were visiting students at ETH Zurich.

✉ Yunyan Guo
guoyunyan@stu.hit.edu.cn

Zhipeng Zhang
zhangzhipeng@pku.edu.cn

Jiawei Jiang
jiawei.jiang@inf.ethz.ch

Wentao Wu
wentao.wu@microsoft.com

Ce Zhang
ce.zhang@inf.ethz.ch

Bin Cui
bin.cui@pku.edu.cn

Jianzhong Li
lijzh@hit.edu.cn

1 Massive Data Computing Research Center, Harbin Institute of Technology, Harbin 150001, China

2 School of EECS, Peking University, Beijing 100871, China

3 Microsoft Research, Redmond, WA, USA

4 Department of Computer Science, ETH Zürich, 8092 Zurich, Switzerland

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 address or bypass this limitation? Meanwhile, is this the *major* reason for the inefficiency of Spark? Are there other bottlenecks that have not been identified yet? If so, are those bottlenecks again due to the fundamental limitations of BSP or just a matter of implementation issue?

In this paper, we aim to understand in more detail why ML on Spark (in particular, MLlib [31], an official Spark package for ML) is slow. We focus on analyzing MLlib's implementation of stochastic gradient descent (SGD)—the workhorse under the training of many ML models. 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 [40] and Angel [22], by slightly tweaking its implementation we are able to significantly speed up MLlib on both public and industrial-scale workloads, when training generalized linear models (GLMs), such as logistic regression (LR) and support vector machine (SVM), using SGD.

Specifically, we find that the update pattern of models in MLlib is not efficient. In MLlib, the driver node is 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* per communication step between the workers and the driver.

We can address this issue by leveraging a simple yet powerful technique called *model averaging* (MA) that has been widely adopted in distributed ML systems [50,51]. The basic idea behind MA 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 updated *many times* per communication step and therefore we can reduce the number of communication steps toward convergence.

The incorporation of MA also enables other optimizations. In particular, we can further improve the communication pattern for distributed SGD. 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 MA this bottleneck can be completely removed—*we do not need the driver per se*. In essence, MA can be performed in a distributed manner across the workers [13]. Roughly speaking, we can partition the model and have each worker maintain a partition. There are two rounds of shuffling during MA. In the first round of shuffling, each worker sends all locally updated partitions to their dedicated maintainers. Afterward, each worker receives all updates of the partition it is responsible for and therefore can perform MA 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. 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 $2 \cdot k \cdot m$, if the number of workers is k 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 industrial workloads shows that MA-SGD—our MA-based version of SGD in MLlib—can achieve significant speedup over MLlib. Furthermore, it can even achieve comparable and often better performance than specialized ML systems like Petuum and Angel, and benefits from the Spark ecosystem.

Moreover, MA is not limited to distributed SGD. As another example, we develop MA-LDA, an MA-based version of training latent Dirichlet allocation (LDA) models in MLlib. Again, we showcase that MA-LDA can significantly outperform its counterpart without using MA. In summary, this paper makes the following contributions:

- We provide a detailed and in-depth analysis of the implementations of MLlib and compare it with other existing distributed ML systems.
- We identify performance bottlenecks when running MLlib and propose using MA as a solution. As case studies, we present two concrete implementations, MA-SGD and MA-LDA, that target training GLMs and LDA using Spark.
- We show that MA-SGD and MA-LDA can achieve significant speedup over their non-MA versions implemented in MLlib. As an extreme example, on one of our datasets, we observed 1000 × speedup.
- We further compare MA-SGD and MA-LDA with specialized ML systems, such as Petuum and Angel that are powered by parameter servers. We show that MA-SGD and MA-LDA can achieve close or even better performance.

We reiterate that the goal of this paper is not to introduce new distributed ML algorithms. Rather, we study how to apply existing techniques (e.g., model averaging) in Apache Spark so that we can significantly improve Spark's performance when it comes to ML training. From a practitioner's perspective, it is vital to have an in-depth analysis for understanding the performance bottleneck of MLlib/Spark and, more importantly, how to improve it if possible. Our work is novel in the sense that it is the first systematic study on this topic, as far as we know. What we demonstrated is that, it is indeed possible to improve the performance of MLlib/Spark

on ML training, often by orders of magnitude, using well-known techniques (e.g., model averaging). We see several implications of our findings:

- MLlib/Spark can now be used to deal with a much broader variety of ML training workloads that were previously impossible. As a result, users no longer need to move data in and out of Spark for such workloads.
- Given the improved performance of MLlib/Spark, it may be worthwhile to revisit previous work that used MLlib/Spark as a baseline. An improved MLlib/Spark may also raise the bar for future research in the area of distributed ML systems and drive the state-of-the-art to a new level.
- While our focus in this work is model averaging and we have showcased its applicability and effectiveness on two representative workloads (SGD and LDA), it is by no means the end of the path. One may want to further extend the work here in two directions at least. First, one can further study whether MA can be used for other workloads. Indeed, our study on using model averaging for LDA is new and we are not aware of any previous work. Second, one can further study the applicability of other techniques beyond model averaging to further improve MLlib/Spark.

Applicability of model averaging. Although there is theoretical guarantee on the convergence when using model averaging for convex problems [51], there is no such guarantee for general non-convex optimization problems. This is a major limitation regarding the applicability of model averaging. Indeed, this is one reason that we focused on GLMs when applying MA-SGD, as we did observe convergence issues when we tried to use MA-SGD for training DNNs.[1] However, as we have showcased with MA-LDA, model averaging can guarantee convergence on certain non-convex problems, though we admit that this is on a case-by-case basis. It is actually our hope that our work in this paper can stimulate further research in this direction.

Paper organization. We start by analyzing the implementations of MLlib in Sect. 2, to identify its performance bottlenecks. We propose to use MA as a solution to these bottlenecks and study its application on SGD in Sect. 3. MA is not tied to SGD, though. To demonstrate this, in Sect. 4 we present another application of MA for training LDA models. In Sect. 5, we further evaluate the performance of MA-SGD and MA-LDA, the two MA-based implementations studied in Sects. 3 and 4, and compare it with other competitive systems. We summarize related work in Sect. 6 and conclude the paper in Sect. 7.

[1] It remains an open question to ensure convergence when using model averaging (perhaps with an implementation different from the current MA-SGD) to train deep models.

2 Performance bottlenecks in MLlib

In this section, we examine the implementation of MLlib and analyze its performance bottleneck. We focus on distributed SGD [35], one of the most popular optimization techniques that has been widely used in ML model training. In the following, we start by providing some background.

2.1 Gradient descent and its variants

Consider the following setting when training ML models. Given a classification task with X representing the input data, find a model w that minimizes the objective function

$$f(w, X) = l(w, X) + \Omega(w) \tag{1}$$

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 Eq. 1. In practice, people usually use a variant called mini-batch gradient descent (MGD) [14]. We present the details of MGD in Algorithm 1.

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 $w_t = w_{t-1} - \eta \cdot g_t - \eta \cdot \nabla\Omega(w_{t-1})$;

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 Eq. 1 using X_B and the current model w_{t-1}; (3) use the gradient to update the model.

The executions of GD and SGD (stochastic gradient descent [24], 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.

2.2 Distributed MGD in MLlib

The sequential execution of MGD is usually not feasible for large datasets and models. Algorithm 2 outlines the implementation of a distributed version of MGD in MLlib. Specifically, there is a *master* (i.e., the driver in Spark) to partition data and schedule tasks. There are multiple *workers*, each dealing with an individual partition. Also, there is a *central node* (i.e., again the driver in Spark) to aggregate the gradients received from the workers.

As shown in Algorithm 2, the master first splits data into multiple partitions with a round-robin partitioning strategy. It then schedules each worker to load a partition and launch a training task. Execution involves multiple stages. In each stage, each worker first 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 finally aggregates the gradients received from the workers and updates the model. This procedure repeats until the model converges.

2.3 Performance analysis

We next present 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 1. 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 (see Fig. 1).[2] Figure 2a presents the *Gantt chart*[3] that tracks the execution of the nodes. The x-axis represents the elapsed time (in s) 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 Fig. 2a:

- (**B1**) Bottleneck at the *driver*—at every stage when the driver is executing, the executors have to wait. Similar observations have been made in previous work, too [8].
- (**B2**) Bottleneck at the *intermediate aggregators*, i.e., the executors that perform partial aggregations of gradient—at every stage when these executors are running, the other nodes have to wait.

[2] 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.

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

Table 1 Dataset statistics

Dataset	# Instances	# Features	Size (GB)
`avazu`	40,428,967	1,000,000	7.4
`url`	2,396,130	3,231,961	2.1
`kddb`	19,264,097	29,890,095	4.8
`kdd12`	149,639,105	54,686,452	21
`WX`	231,937,380	51,121,518	434
Dataset	**# Document**	**Vocabulary**	**Size (GB)**
`NYTimes`	269,656	102,660	0.5
`PubMed`	8,118,363	141,043	4.4
`CommonCrawl`	1,000,000	100,000	2.0

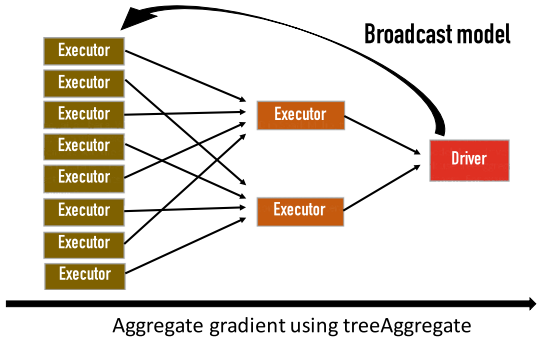


Fig. 1 Communication patterns of distributed MGD on Spark MLlib

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.

3 MA-SGD: SGD with model averaging

Model averaging (MA) is a well-known optimization strategy for speeding up SGD [50,51]. Instead of sending and aggregating gradients, each worker can actually perform 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 averaging the (local) model received from the workers. Algorithm 3 illustrates these two related primitives that replace their counterparts in Algorithm 2. It has been shown that using MA in SGD offers the same convergence guarantee as standard SGD on convex problems [36].

Remark We name the two paradigms (in Algorithms 2 and 3) `SendGradient` and `SendModel`, respectively. The dif-

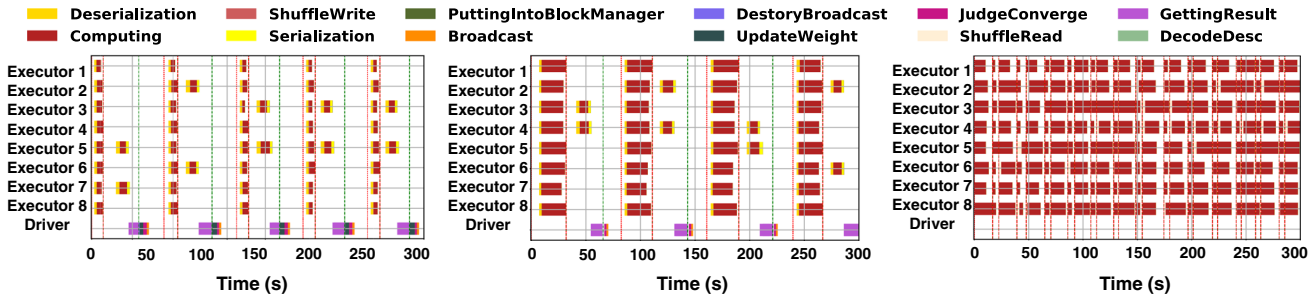


Fig. 2 Gantt charts for **a** MGD executions in MLlib (i.e., `SendGradient`), **b** MLlib with model averaging (i.e., `SendModel`), and **c** MA-SGD. A red vertical line represents the start of a stage in Spark, whereas the subsequent green vertical line represents its end

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;
  ComputeAndSendGradient(w_{t-1})
Function *ComputeAndSendGradient*(w_{t-1})*:*
  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;

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

Algorithm 3: Model-Averaging primitives

Worker r = 1,…, m:
Function *ComputeAndSendLocalModel*(w_{t-1})*:*
  Compute model w_t^r via *MGD*(T', η, w_{t-1}, X_r);
  // T' is the number of iterations in each worker.
  Send local model w_t^r to the central node;

Central node:
Function *AggregateLocalModelsAndUpdateGlobalModel()*
  Aggregate the models as $w_t \leftarrow f(\sum_{r=1}^{m} w_t^r)$;

ference 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 many more updates and thus much faster convergence.

Implementation. Most parts of our implementation are quite straightforward. We basically replace the computation of gradients in each executor by model updates. And in the communication step, we send model updates instead of gradients. 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 [10]. Our implementation does not require any change to the core of Spark. Instead, we implement our techniques leveraging primitives provided by Spark.

Analysis. Figure 2b presents the Gantt chart of MLlib after incorporating our implementation of `SendModel`, by rerunning the experiment described in Sect. 2.3. One can observe that the computation time in Fig. 2b is longer than that in Fig. 2a, since it processes the whole dataset, instead of a mini-batch. However, for each data instance, 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 Fig. 2b should be much smaller compared with Fig. 2a if we extend the x-axes of both charts to the time of convergence, which suggests a much faster convergence of `SendModel`.

3.1 Optimization powered by MA: distributed aggregation using AllReduce

The communication pattern exhibited in Fig. 2b remains the same as that in Fig. 2a: Even if we now aggregate 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

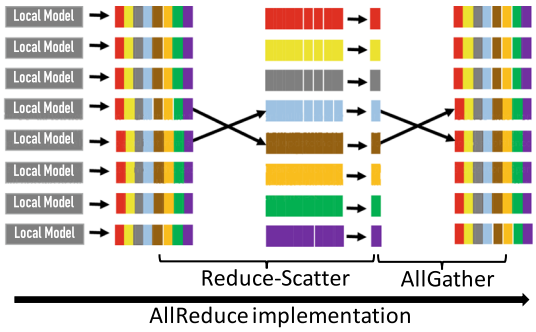


Fig. 3 *AllReduce* implementation using shuffle on Spark

`SendGradient`: Executors send the gradients to the driver; the driver sends the updated model back to the executors. However, in MA-SGD, 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 that the driver first collects the (aggregated) model updates from the executors and then broadcasts the model updates back to the executors.

To address this problem, we implement a new communication pattern in Spark. The basic idea is to partition the global model and let each executor *own* one partition. 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. The owner of a model partition is responsible for its maintenance (using MA). 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 MA 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 MA architecture we shall propose. Moreover, given that we do not need the driver to take charge of MA, the hierarchical aggregation scheme presented in Fig. 1 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 Fig. 3, we have eight executors E_1 to E_8, each owning one partition, which is 1/8 of the global model M. Let us number those partitions from M_1 to M_8. Considering the partition M_1 in E_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 MA 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. 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. We use a two-phase procedure to address these two challenges (see Fig. 3 and Algorithm 4 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. Afterward, 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. Afterward, each executor now has refreshed versions of all partitions of the global model. This stage is motivated by the work of Thakur et al. [37].

Again, our implementation does not require changes to the core of Spark. Specifically, we use the `shuffle` operator in Spark to implement both stages: One can write different shuffling strategies by specifying the source(s) and destination(s) of each partition.[4] Figure 2c 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 Fig. 2c, 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 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

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

`AllGather` phases at the same timestamp, i.e., the first vertical line in Fig. 2c.

It is worth to point out that while the Gantt chart in Fig. 2c looks much more cluttered compared with Fig. 2b, 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 $2 \cdot k \cdot m$ for both cases.[5] This may seem puzzling as one may expect that the two rounds of shuffling we employed in MA-SGD 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 MA-SGD can finish the same number of model updates as a step in the "MLlib + MA" mechanism can but the latency is much shorter.

As a side note, the two names `Reduce-Scatter` and `AllGather` have borrowed from MPI (acronym for "Message Passing Interface") terminology, which represent MPI operators/primitives with the same communication patterns plotted in Fig. 3. 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. [37] for more details about these MPI primitives.

4 MA-LDA: LDA with model averaging

Model averaging is not limited to SGD. We now demonstrate this by showcasing another application of MA in distributed latent Dirichlet allocation (LDA). LDA is perhaps the most popular formulation in *topic modeling*, with wide applications in various areas such as natural language processing and information retrieval. Distributed computation is inevitable when LDA models become large. Existing work mainly focuses on paralleling Monte Carlo Markov chain (MCMC), whereas parallelism of variational inference (VI), another popular optimization technique for LDA, has yet not been studied in depth.

We propose MA-LDA, a distributed framework for training LDA models. MA-LDA is based on a stochastic version of VI (SVI), combined with MA. We can prove that applying MA in SVI does not sabotage its convergence.[6] In fact, MA significantly reduces the amount of communication between the computational workers and therefore dramatically speeds up the convergence of SVI. As before, we start by providing some background.

[5] We ignore the intermediate aggregators in Fig. 2b.

[6] However, the proof is very lengthy and technical, and thus is omitted here.

Algorithm 4: MA-SGD $\{T, \eta\ w_0, X, m\}$

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 $w^r_1, w^r_2, \ldots, w^r_m$;
  for $i = 1$ **to** m **do**
    Send partition w^r_i to worker i;
  // Perform model averaging for partition r
  // (after receiving updates from all other workers).
  $\overline{p^r} \leftarrow \frac{1}{m} \sum_{j=1}^{m} w^j_r$;
  // 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.
  $w^r \leftarrow (\overline{p^1}, ..., \overline{p^m})$;

4.1 Variational inference for LDA

Given an input corpus X and the number of topics K, LDA aims to model each topic as a distribution over the vocabulary V. We use w to represent topic-level parameters. w is a $K \times V$ matrix. Also, LDA assumes that each document is a *mixture* over topics, and LDA infers the *topic assignment* of each word in each document. z represents these data-level parameters.

Exact inference of the real distribution $p(w, z|X)$ is intractable with the massive corpus. Variational inference (VI) aims for minimizing the distance between $p(w, z|X)$ and a variational distribution $q(w, z)$ in terms of their KL-divergence. This optimization problem is equivalent to maximizing the "evidence lower bound" (*ELBO*):

$$ELBO = \mathbf{E}_q[\log p(w, z, X)] - \mathbf{E}_q[\log q(w, z)] \quad (2)$$

It is common to use SVI [16]—a stochastic, online version of VI—when dealing with huge or streaming data. SVI focuses on training w and treats z as part of data information.

In each iteration, it uses a mini-batch corpus to calculate the estimated *natural gradient* of w. Optimization of the *ELBO* is achieved by using *natural gradient ascent* on w. It is a special case of MGD, where the natural gradients are gathered in the same way as standard gradients.

4.2 Distributed SVI in MLlib and MA-LDA

MLlib implements a distributed version of SVI. As shown in Algorithm 5, its main structure is similar to the SGD implementation by MLlib (Algorithm 2), which follows the `SendGradient` paradigm. The difference lies in the additional computation of $\mathbf{E}_q[\log W]$ (via the *DirichletExpectation* function) by the central node (i.e., the driver in Spark). The computation is expensive, which makes the bottleneck at the driver (i.e., B1 in Sect. 2.3) even worse.

Inspired by MA-SGD (Algorithm 3), we design MA-LDA, an MA-based variant for distributed SVI following the `SendModel` paradigm. Algorithm 6 presents the details. The optimization techniques based on `Reduce-Scatter` and `AllGather` are again suitable here. Compared to MA-SGD, the implementation of the function *UpdateModel* alters: In MLlib's implementation (Algorithm 5), only the driver is responsible for computing $\mathbf{E}_q[\log W]$; in MA-LDA, however, the computation is distributed across the workers. *rowSum* function is to calculate the sum of each row(a parameter vector of Dirichlet distribution). The function *PartDirExp* illustrates how each worker computes its *partial* version of $\mathbf{E}_q[\log W]$ by only utilizing its local data partition. To reduce computation intensity, we further develop two optimization strategies here: (1) *lazy update* and (2) *low-precision computation*.

Lazy update. Documents in practice are usually *sparse*: Each document only covers a small fraction of words in the entire vocabulary. This sparsity property allows MA-LDA to focus on a subset with only nonzero entries in the $\mathbf{E}_q[\log W]$ matrix that corresponds to the local corpus of each worker. This has been highlighted in the implementation of *PartDirExp()*. Although the natural gradient $\tilde{g}^r$ is often dense, the update of w, which is typically a large matrix, can be acted in a lazy, sparse manner: Only entries required by the next *PartDirExp()* call need to be updated.

Low precision computation. The most time-consuming step in *UpdateModel* is the computation of $\mathbf{E}_q[\log W]$. For each topic $k \in [K]$, $q(W_k|w_k) \sim \text{Dirichlet}(w_k)$, which is a vector of length V. By definition, $\mathbf{E}_q[\log W_{k,v}] = \Psi(w_{k,v}) - \Psi(\sum_{i=1}^{V} w_{k,i})$, where Ψ is the *digamma function*, i.e., the logarithmic derivative of the gamma function.[7] To improve the computation efficiency of Ψ, we use a recursive approach as follows.

Algorithm 5: Distributed SVI LDA$\{T, \eta\ w_0, X, m\}$

Master:
Issue *LoadData()* to all workers;
Issue *InitialModel*(w_0) to the central node;
for *Iteration* $t = 0$ **to** T **do**
  Issue *DirichletExpectation*(w_t) to the central node;
  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 $\mathbf{E}_q[\log W_t]$ from the central node;
  ComputeAndSendGradient($\mathbf{E}_q[\log W_t]$);
Function *ComputeAndSendGradient*($\mathbf{E}_q[\log W_t]$)*:*
  Sample a batch of data X_{br} from X_r;
  Compute gradient $\tilde{g}_t^r \leftarrow \sum_{x_i \in X_{br}} \tilde{\nabla} l(\mathbf{E}_q[\log W_t], x_i)$;
  Send gradient $\tilde{g}_t^r$ to the central node;

Central node:
Function *InitialModel*(w_0)*:*
  Initialize model as w_0;
Function *DirichletExpectation*(w)*:*
  for *each data point* $w_{k,v}$ *in* w **do**
    $\mathbf{E}_q[\log W_{k,v}] = \Psi(w_{k,v}) - \Psi(\sum_{i=1}^{V} w_{k,i})$;
  Return *Topics* matrix $\mathbf{E}_q[\log W]$;
Function *ServerTask(t):*
  AggregateGradientsAndUpdateGlobalModel();
Function *AggregateGradientsAndUpdateGlobalModel()*
  Aggregate gradient as $\tilde{g}_t \leftarrow \sum_{r=1}^{m} \tilde{g}_t^r$;
  Update the model as $w_t \leftarrow w_{t-1} + \eta \cdot (\tilde{g}_t)$;

Given a scalar constant $c > 1$,

$$\Psi(x) = \begin{cases} \ln x - \frac{1}{2x} - \frac{1}{12x^2} + \frac{1}{120x^4} - O(\frac{1}{x^6}), & x > c; \\ \Psi(x+1) - \frac{1}{x}, & 0 < x \le c. \end{cases}$$

The precision of $\Psi(x)$ so computed depends on the choice of c [7]. MA-LDA chooses a relatively small c to seek a trade-off between computation efficiency (measured by the number of recursions) and precision.[8]

5 Experimental evaluation

In this section, we compare MA-based algorithms 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 over MLlib, but we also want to understand where MLlib- and MA-based algorithms stand in the context of state-of-the-art distributed ML systems. Although these specialized systems have claimed to be much better than MLlib, the reported results were based on different experimental settings or even

[7] https://en.wikipedia.org/wiki/Digamma_function.

[8] The default value of c is 49 to guarantee a relative error of $1e{-}8$, though $c = 9$ is enough for a relative error of $1e{-}5$.

Algorithm 6: MA-LDA $\{T, \eta\ w_0, X, m\}$

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;
  Compute $rowSum(w^r)$;
  Sample a batch of data X_{br} from X_r;
  for *each data point x in* X_{br} **do**
    Compute *PartDirExp*($w^r, x, rowSum(w^r)$);
    Compute gradient: $\tilde{g}^r \leftarrow \tilde{\nabla} l(\mathbf{E}_q[logW^r], x)$;
    Update model: $w^r \leftarrow w^r + \eta \cdot \tilde{g}^r$;
    Update sums of rows:
    $rowSum(w^r) \leftarrow rowSum(w^r) + rowSum(\eta \cdot \tilde{g}^r)$;
Function *PartDirExp*($w, x, rowSum(w)$)*:*
  for $k = 1$ **to** K **do**
    for *each v exists in x* **do**
      $\mathbf{E}_q[\log W_{k,v}] = \Psi(w_{k,v}) - \Psi(rowSum(w)_k)$;
  Return $\mathbf{E}_q[\log W]$;
Function *rowSum*(v)*:*
  // v is a vector;
  Return the sum of all values in vector v;

different ML tasks. We are not aware of any previous study with a similar level of completeness, and we hope our results can offer new insights to developing, deploying, and using distributed ML systems.

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

- We find that specialized ML 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, *model averaging* and *distributed aggregation*, we observe a significant speedup of MA-based algorithms over MLlib algorithms.
- We further show that MA-based algorithms can achieve comparable and sometimes better performance than Petuum and Angel that are based on parameter servers.

5.1 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. And each CPU has 8 cores.
- *Cluster 2* consists of 953 nodes, with 345 TB of memory in total. Each node has 2 CPUs, and each CPU has 10 cores. The nodes are connected with a 10-Gbps network.

GLMs workloads and metrics. We evaluate different distributed ML systems for training GLMs. Specifically, we train SVM on five datasets (details below), with and without L1/L2-norm regularization.[9] However, the performances of using L1 and L2 norms are very similar under our MA-based implementation. The major difference between using L1 and L2 norms is that, the updates when using L2 norms are dense, whereas the updates when using L1 norms are sparse. Nonetheless, in our MA-based implementation, we have optimized the updates of L2 norms by using *lazy update*. As a result, the updates when using L2 norms under our implementation are now also sparse. Therefore, we do not observe a performance gap between using L1 and L2 norms. For this reason, in this paper, we will only show experimental results based on L2 norms.

We use four public datasets from MLBench [47], as well as the dataset `WX` from Tencent. Table 1 presents the statistics of these datasets.[10]

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 a good chance to probe and understand the strengths and weaknesses of different systems.

We measure the value of $f(w, X)$ in Eq. 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*. In addition to the elapsed time taken by each system toward convergence, we also measure the number of communication steps when comparing MLlib-SGD and MA-SGD. The speedup is calculated when the accuracy loss (compared to the optimum) is 0.01.

LDA workloads and metrics. To evaluate different ML systems for LDA, we use the New York Times (`NYTimes`) and the Public Medicine (`Pubmed`) datasets from the UCI Machine Learning Repository, as well as the dataset from

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

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

Common Crawl.[11] Table 1 summarizes the statistics of these datasets.

The datasets contain documents with different lengths. The average length of the documents in `PubMed` is 90, whereas the ones in `NYTimes` and `CommonCrawl` are 230 and 400. This diversity in document length serves the purpose of testing both the strengths and weaknesses of different systems when training LDA. Long documents can test the inference quality of different algorithms; on the other hand, short documents can result in lower computation costs but higher communication costs.

When comparing LDA in MA-LDA with MLlib-LDA, we use the *perplexity* on the held-out test set as the metric, which evaluates the performance and accomplishment of model fitting. Lower perplexity means the model has less confusion to infer topic proportions for each word in test data.

Angel-LDA (LDA*) is a state-of-the-art system that implements the distributed MCMC algorithm for training LDAs with parameter servers [43]. Using the parameter-server architecture, Angel-LDA pushes certain computation to the (parameter) server side to balance between computation and communication costs. Angel-LDA keeps calculation for certain types of data points on the servers. As a result, the amount of calculation assigned to the workers becomes less. Moreover, the amount of data transmitted to the workers is also less. It also introduces a well-designed hybrid sampler. As reported by Yu et al. [43], the performance of Angel-LDA is much better than Petuum-LDA. We therefore choose Angel-LDA as a representative of parameter-server solutions and compare it with MA-LDA. Unlike MLlib-LDA, Angel-LDA uses *log-likelihood* as the metric over the training set. For fairness, we therefore also evaluate log-likelihood when comparing MA-LDA and Angel-LDA.

Participating systems and configurations. In our evaluation, we compare four distributed ML systems: (1) Petuum 1.1, (2) Angel 1.2.0, (3) Spark MLlib 2.3.0, (4) MA-MLlib (i.e., MLlib with our MA-based algorithms). 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.[12]

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

[11] https://archive.ics.uci.edu/ml/machine-learning-datasets/ and http://commoncrawl.org.

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

5.2 Evaluation on public datasets

We evaluate MA-MLlib using the public datasets in Table 1 with the following goals in mind:

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

5.2.1 Model quality

We first study the quality of the ML models returned by different systems. Since the objective function of GLMs is convex, all participating systems should converge to the same global optimum. On the other hand, the objective function of LDA is non-convex. As a result, different systems may result in models converging to different local optima. Therefore, we need to compare the LDA training quality of MA-LDA with MLlib-LDA and Angel-LDA.

We start by comparing the model quality of MLlib-LDA and MA-LDA. We fix the topic size and set other hyperparameters (i.e., document concentration and topic concentration) using default values to make comparison fair. Since different batch sizes can lead to different model quality [17], we use various batch sizes in our experiments. For each batch size, we ran 8 workers for four hours and then compute the final perplexity using the test data sets. Table 2 reports the final perplexity values with different batch sizes. We make the following observations:

- MA-LDA can achieve lower perplexity than MLlib-LDA for most of the cases.
- Across different batch sizes, the lowest perplexity of MA-LDA is lower than that of MLlib-LDA.
- The proper batch size (w.r.t. the lowest perplexity) of MA-LDA is larger than that of MLlib-LDA.

We see that MA-LDA has better model quality (w.r.t. the proper batch size) than MLlib-LDA. The proper batch size of MA-LDA is also larger, implying more local computation work on each worker but less communication overhead.

We next compare MA-LDA with Angel-LDA. We run Angel-LDA on 8 machines (8 workers and 8 parameter servers) and stop after 100 iterations (i.e., epochs) with tuned hyperparameters. For MA-LDA, we chose the best batch size and tuned learning rate for each data set.[13] MA-LDA converges after 10 iterations, and we then record its log-likelihood over the training set. Table 3 summarizes

[13] We chose batch size as follows: 16k for `NYTimes`, 64k for `PubMed`, and 100k for `CommonCrawl`.

Table 2 Perplexity of NYTimes and PubMed with different mini-batch sizes

Dataset	System	1k	4k	16k	64k
NYTimes	MLlib-LDA	**8797**	8981	11,181	16,852
NYTimes	MA-LDA	8967	8352	**8110**	8274
Dataset	System	4k	16k	64k	256k
PubMed	MLlib-LDA	8813	**8649**	9648	12,083
PubMed	MA-LDA	8866	8351	**8180**	8300

The bold values are the best ones from the corresponding comparisons

Table 3 Log-likelihood of NYTimes, PubMed and CommonCrawl

System	Iters	NYTimes	PubMed	CommonCrawl
Angel-LDA	100	− 8.28e8	− 6.31e9	− 4.46e9
MA-LDA	10	**− 7.71e8**	**− 6.09e9**	**− 4.21e9**

The bold values are the best ones from the corresponding comparisons

the results. The results Angel-LDA achieves on NYTimes and PubMed is consistent with that in [43]. On the dataset CommonCrawl, Angel-LDA converges after 100 iterations. We observe that MA-LDA achieves higher log-likelihood than Angel-LDA over all three datasets.

5.2.2 Efficiency comparison with MLlib

Gradient descent. Figure 4 compares MA-SGD and MLlib-SGD using the four public datasets on GLMs. We trained SVMs with and without L2 regularization. In each subfigure, the left plot shows the change in the value of the objective function as the number of communication steps increases, and the right plot shows that change w.r.t. the elapsed time.

We can observe several facts. First, compared to MLlib-SGD, MA-SGD converges much faster. As Fig. 4h indicates, MLlib-SGD 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 MA-SGD, 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 MA-SGD is 240 × instead of 80 × faster if the speedup were just linear. This extra speedup is attributed to the AllReduce technique implemented in MA-SGD. It is a bit surprising at a first glance—one may not expect that the improvement from AllReduce is more significant than SendModel. This essentially implies that the computation workload at the driver node is expensive, regardless of whether it is the aggregation of gradients or models.

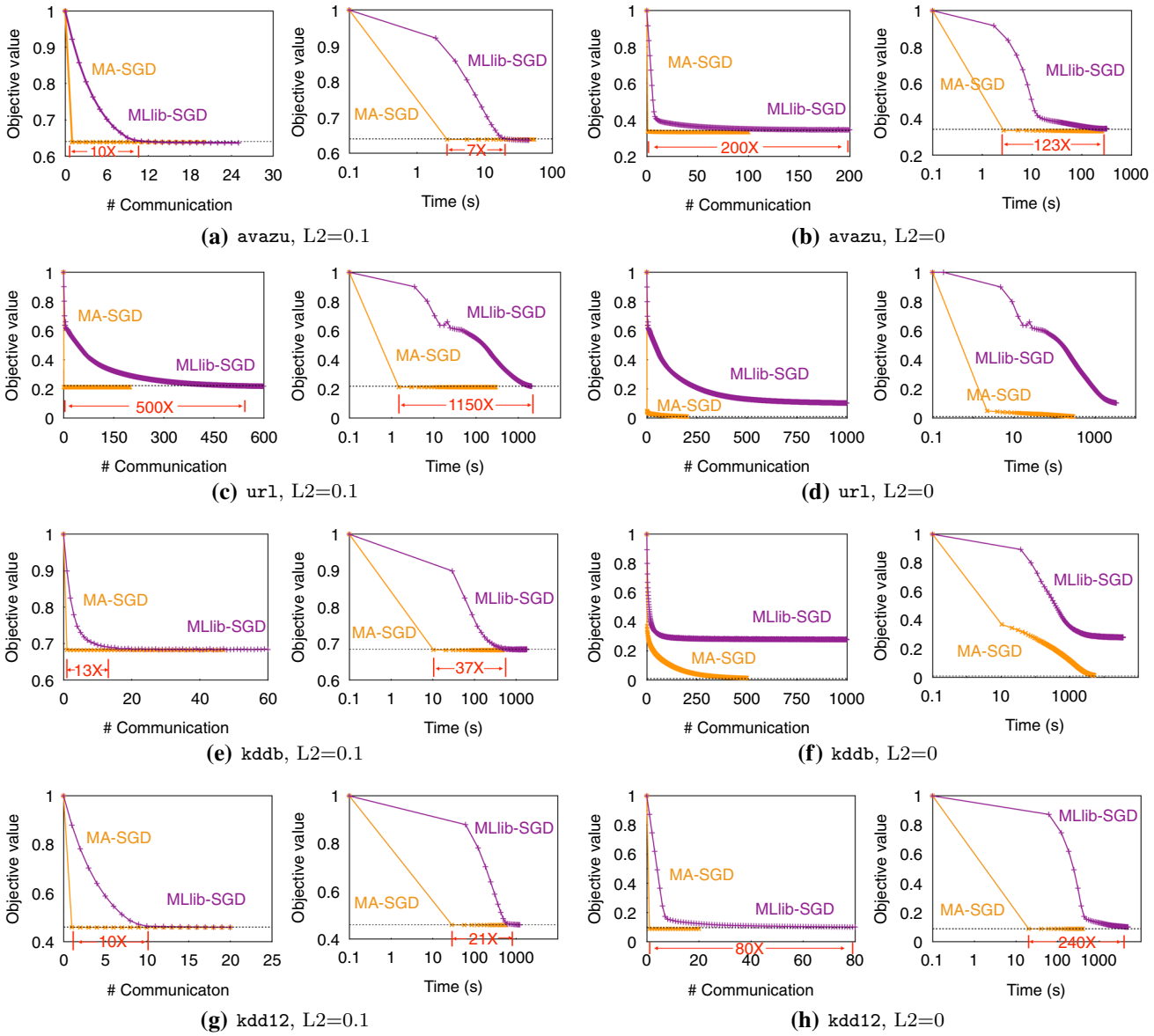
Of course, the severity of the bottleneck depends on the sizes of the data and the model—the larger they are the worse the bottleneck is. For example, as shown in Fig. 4b, MLlib-SGD needs 200 × more iterations to converge while is only 123 × slower than MA-SGD. This implies that the time spent on each iteration of MA-SGD is longer than that of MLlib-SGD. There are two reasons. First, the batch size of MLlib-SGD 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, MA-SGD needs to pass the entire dataset in each iteration. As a result, the computation overhead per iteration of MA-SGD 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-SGD is less and the benefit from AllReduce in MA-SGD is smaller.

Second, MLlib-SGD performs worse when the problem becomes more ill-conditioned. As shown in Fig. 4b, d, f, h, MLlib-SGD converges 123× and 200 × slower than MA-SGD on the two determined datasets avazu and kdd12, while they cannot get to the optimal loss even after 1000 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 Fig. 4a, c, e, g, respectively. We can observe that the performance gap between MLlib-SGD and MA-SGD becomes smaller when the training objective becomes more determined. For example, the speedups decrease to 7 × and 21 × on avazu and kdd12. Meanwhile, on url and kddb, MLlib-SGD can now converge to the same loss as MA-SGD.

Third, distributed aggregation is more beneficial for large models. As we can infer from comparing Fig. 4e with Fig. 4a, the speedup per iteration of MA-SGD over MLlib-SGD on high-dimensional dataset like kddb is more significant than that on low-dimensional dataset like avazu.[14] 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 1).

Beyond MLlib-SGD. We further compare MA-SGD with spark.ml, an ML library in Spark 2.3 that implements L-BFGS [28], a popular second-order optimization technique (i.e., it utilizes both first-order and second-order derivatives of the objective function). It is well known that there is a trade-off between first-order and second-order optimization techniques [3]. Although both are iterative, first-order

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



(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-SGD and MA-SGD on four public datasets. The dotted line in each figure represents 0.01 accuracy loss. MA-SGD reaches the target loss in a single iteration in most cases

techniques tend to be more efficient in each iteration but may take a larger number of iterations before converging to the optimum. Interestingly, it is the "last mile" phenomenon that perplexes first-order techniques: Despite that, they can quickly find a small neighborhood containing the optimum, they often get stuck and make slow progress in that neighborhood. Second-order techniques overcome this issue. Motivated by these observations, Agarwal et al. [3] have proposed *hybrid* approaches that first use first-order techniques to reach a good neighborhood and then switch to second-order techniques when searching for the optimum within that neighborhood. We also implemented a `Hybrid` approach that first runs model averaging SGD for an epoch and then switches to `spark.ml`.[15]

Figure 5 presents the results on four datasets (L2 = 0). MA-SGD outperforms L-BFGS and `Hybrid` across most of the datasets. This may seem a bit contradictory to observations made in the literature that the hybrid approach should be better in general. We emphasize that our results do not rebut previous results, for the following two reasons: (1) The implementation of L-BFGS can be further optimized, using techniques such as ones introduced by Chen et al. [12]; (2) the performance gap between MA-

[15] Since L-BFGS in spark.ml performs normalization, we evaluate the models with the normalized data.

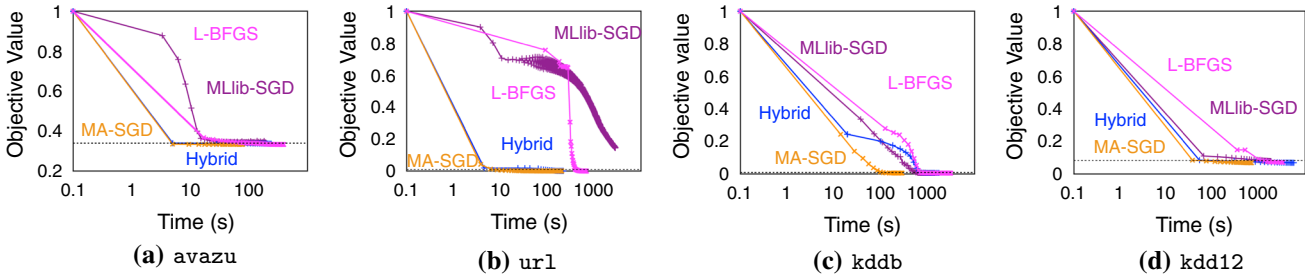


Fig. 5 Comparison of MA-SGD, L-BFGS, and the hybrid approach (the dotted line in each plot means 0.01 accuracy loss)

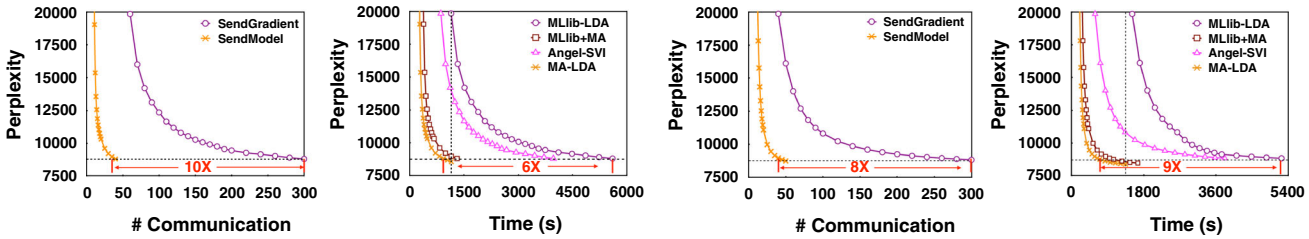


Fig. 6 Comparison of training LDA with MLlib-LDA, MLlib+MA, Angel-SVI, and MA-LDA on NYTimes and PubMed

SGD and L-BFGS is perhaps more due to the fact that MA-SGD uses the SendModel paradigm, whereas L-BFGS uses the SendGradient paradigm.

Variational inference. Figure 6 compares MLlib-LDA, MLlib+MA, Angel-SVI, and MA-LDA using two public datasets. We use the SendModel strategy in MLlib-LDA to implement MLlib+MA. We also implement Angel-SVI, an SVI implementation on top of Angel, with the same algorithm of MLlib-LDA using the AllReduce strategy. With SendGradient, MLlib-LDA and Angel-SVI achieve the same perplexity after the same number of iterations. In addition, with SendModel, MLlib+MA and MA-LDA also achieve the same perplexity after the same number of iterations. As shown in Table 2, the batch size of SendGradient is 1K for NYTimes and 16K for PubMed, with which we achieve the lowest perplexities. On the other hand, the batch size of the SendModel is 16K for NYTimes and 64K for PubMed, respectively. In our experiments, we controlled each run to stop after 300 communication steps. In each subfigure, the left plot shows the change of the perplexity when the number of communication steps increases, and the right plot shows that change corresponding to the elapsed time.

We have the following two observations. First, MA-LDA converges faster than MLlib-LDA. The left plots in Fig. 6a, b indicate that MLlib-LDA needs 300 communication steps to converge, whereas MA-LDA needs no more than 40 communication steps. It demonstrates that the SendModel paradigm in MA-LDA significantly improves LDA training, compared with the SendGradient paradigm in MLlib-LDA.

Second, the right plots in Fig. 6a, b show that the running time of MA-LDA is roughly 6 × and 9 × faster than MLlib-LDA on two datasets. From Table 5, MA-LDA processes 13 × more documents for NYTimes more documents per second for PubMed, compared to MLlib-LDA.

Detailed analysis of MA and AllReduce. We further perform an in-depth analysis to understand the individual contributions by MA and AllReduce to the overall performance improvements. Specifically, we compare the performances of MLlib-SGD, MLlib+MA, and MA-SGD when running over kdd12 for a fixed duration (i.e., 300 s in our experiment). Table 4 summarizes the results, and the Gantt charts are shown in Fig. 2.

We have the following observations regarding the effectiveness of MA:

- MLlib+MA converges to a loss of 0.459 and MLlib-SGD converges to a loss of 0.588 in 300 s. The convergence of MLlib+MA is faster because MA reduces the number of iterations while updating the model more times in each iteration.
- For each iteration, the communication costs of MLlib-SGD and MLlib+MA are similar since they both use TreeAggregation and Broadcast.
- On the other hand, the computation time of MLlib+MA is longer than MLlib-SGD in one single iteration. MLlib-SGD processes a mini-batch, while MLlib+MA needs to process the whole training set in each iteration.

Table 4 Comparison of MLlib-SGD, MLlib+MA, and MA-SGD on kdd12 with $L2 = 0.1$ in 300 s

kdd12	MLlib-SGD	MLlib+MA	MA-SGD
# iterations	5	4	**13**
Loss value	0.588	**0.459**	**0.459**

The bold values are the best ones from the corresponding comparisons

Table 5 Comparison of training LDA with MLlib-LDA, Angel-SVI, MLlib+MA, and MA-LDA on NYTimes in 1150 s and PubMed in 1350 s

NYTimes	MLlib	Angel-SVI	MLlib+MA	MA-LDA
# iters	55	80	35	50
# docs (K)	55	80	560	800
ppl	19,859	14,187	8970	**8547**
PubMed	MLlib	Angel-SVI	MLlib+MA	MA-LDA
# iters	30	100	60	100
# docs (M)	0.48	1.6	3.84	6.4
ppl	28,947	10,818	8612	**8372**

The bold values are the best ones from the corresponding comparisons

Regarding the effectiveness of using AllReduce, we further have the following observations by comparing MLlib+MA with MA-SGD:

- Compared to MLlib+MA, MA-SGD can finish more iterations (13 vs. 4) in 300 s, since our AllReduce implementation significantly reduces the communication cost of each iteration.
- With MA, both MLlib+MA and MA-SGD converge to the same target loss after one iteration, as shown in Fig. 4g. However, MA-SGD achieves the target loss in just 24.8 s, whereas it takes 81.2 s for MLlib+MA to achieve the same loss.

On the other hand, if we only enhanced MLlib by introducing the AllReduce strategy to aggregate the gradients, the benefit would be rather limited. As Table 5 shows, although the AllReduce strategy speeds up MLlib-LDA, the perplexity values are unacceptable, due to the fact that the SendGradient paradigm needs hundreds of iterations to converge. However, MA-LDA can converge faster and achieve the target perplexity with the same running time.

In summary, both MA and AllReduce aim at accelerating communication for ML training though they look quite independent. MA achieves this by redesigning the communication architecture, whereas AllReduce improves the optimization algorithm. As a result, MA and AllReduce should be viewed as one holistic rather than two separate solutions.

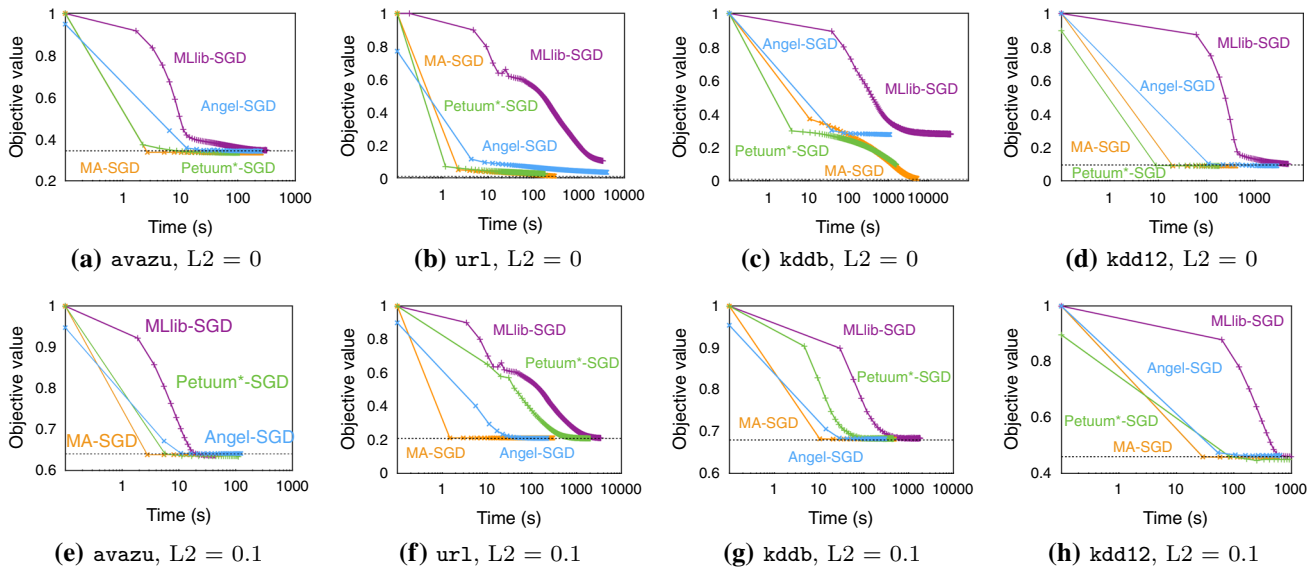
5.2.3 Efficiency comparison with parameter servers

Gradient descent. As shown in Fig. 7, we compare the performance of MA-SGD with Petuum*-SGD and Angel-SGD over the four datasets, with and without L2 regularization. Here, Petuum*-SGD is a slightly tweaked implementation of SGD in Petuum. The original implementation of SGD in Petuum uses model summation instead of model averaging, which has been pointed out to be problematic [20,50]—it suffers from potential divergence. Therefore, we replaced model summation in Petuum by model averaging and call this improved version Petuum*-SGD—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-SGD.

We have the following observations. First, Fig. 7 confirms that MLlib-SGD can be significantly slower than Petuum*-SGD and Angel-SGD, resonating previous studies [22,40,49]. Both Petuum*-SGD and Angel-SGD employ the SendModel paradigm and therefore are understandably more efficient.

Second, as Fig. 7a, b, c, d indicates, MA-SGD can achieve comparable or better performance as those of Petuum*-SGD and Angel-SGD when L2 regularization vanishes. Specifically, MA-SGD and Petuum*-SGD 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*-SGD is implemented in C++, while MA-SGD is implemented using Scala. Also, Petuum*-SGD uses SSP to alleviate potential latency from stragglers. On the other hand, MA-SGD is faster than Angel-SGD, because Angel-SGD cannot support small batch sizes very efficiently due to flaws in its implementation. Roughly speaking, Angel-SGD 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.

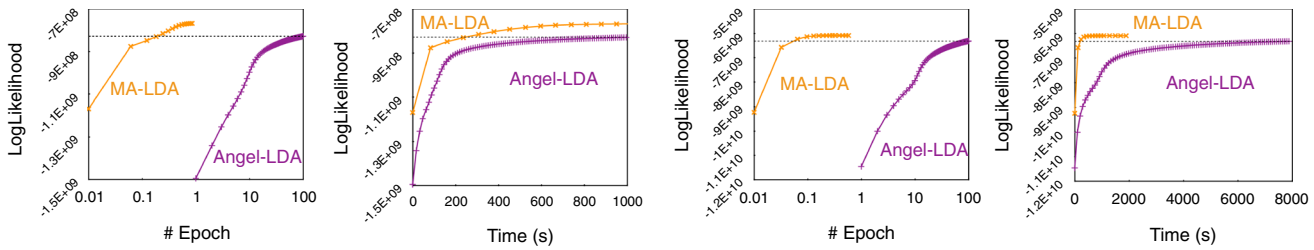
Third, MA-SGD is faster than both Petuum*-SGD and Angel-SGD when L2 regularization is nonzero on the four datasets, as shown in Fig. 7e, f, g, h. Sometimes the performance gap between MA-SGD and parameter servers is quite significant, for example, on the url and kddb datasets as shown in Fig. 7f, g. Moreover, Angel-SGD outperforms Petuum*-SGD (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*-SGD contains only one update to the model, which



(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. 7 Comparison of MA-SGD and parameter servers. The dotted line in each figure represents 0.01 accuracy loss



(a) `NYTimes`, 200 topics **(b)** `PubMed`, 100 topics

Fig. 8 Comparison of MA-LDA and Angel-LDA on `NYTimes` and `PubMed`, with various topic sizes

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 MA-SGD 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 [10], which can boost the number of updates to the model per communication step. The results of the model are the same with/without *lazy update*. However, *lazy update* can convert dense updates into sparse updates (corresponding to the model size). Therefore, in each communication step, we need fewer write operations of the model with *lazy update*.

Variational inference. Angel-LDA implements LDA* [43], and its performance is much better than Petuum-LDA and MLlib-LDA. Angel-LDA is the state-of-the-art distributed MCMC algorithm with parameter servers for training LDAs. We now compare MA-LDA with Angel-LDA.

Figure 8 presents the results. The models trained by Angel-LDA are worse than MA-LDA since they are using two different training algorithms—MCMC and VI. Although MCMC and VI can asymptotically achieve similar losses in theory, empirically the model quality of MCMC is sometimes unsatisfactory [29,45]. Moreover, MA-LDA also converges faster than Angel-LDA. We can understand this from two perspectives. First, Angel-LDA processes more data than MA-LDA. In each iteration, Angel-LDA needs to access the whole dataset, whereas MA-LDA can calculate estimated natural gradients from a mini-batch. As a result, Angel-LDA needs 100 epochs to converge, whereas MA-LDA converges after less than one epoch, as evidenced by the left plots in each subfigure of Fig. 8. Second, although Angel-LDA runs faster with a well-designed MCMC sampler, MA-LDA also speeds up the calculation of natural gradients with the lazy update and low-precision computation. Moreover, in each iteration of MA-LDA, the `SendModel` paradigm offers higher-quality topics w^r for documents, which makes the local inference more effectively and efficiently.

Another interesting observation is that the gap of running time between Angel-LDA and MA-LDA on NYTimes is smaller than that on PubMed, whereas the gap of model quality between Angel-LDA and MA-LDA is larger. This is can be understood by the fact that the average length of the documents in NYTimes is larger than PubMed, which implies that the LDA model on NYTimes is more complicated. Although LDA assumes that a word can belong to multiple topics, MCMC techniques in general focus on just one topic [25]. Therefore, the impact on Angel-LDA due to the difference between two datasets is small. On the other hand, MA-LDA, as a VI algorithm, can perform more complicated inferences with the mixture of multiple topics to find a better model. The downside is that it may take more time (e.g., NYTimes vs. PubMed), though it is still faster than MCMC.

5.3 Scalability test

Gradient descent. To report the scalability of MA-SGD performs when training GLMs, we compare MA-SGD 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 the WX dataset. The dataset cannot be fit into the memory of a single machine. Therefore, we performed scalability tests with 32, 64, and 128 machines. We use the grid search to find the best hyperparameters for each participating system. We do not have results for Petuum*, because the deployment requirement of Petuum is not satisfied on *Cluster 2*. Figure 9 presents the results.

First, Fig. 9a demonstrates that MA-SGD converges much faster than Angel-SGD and MLlib-SGD when using 32 machines. The loss of Angel-SGD and MLlib-SGD is still decreasing, but they need a much longer time. Compared to MLlib-SGD and Angel-SGD, MA-SGD 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. Figure 9a, b, c, d shows the convergence using different numbers 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-SGD becomes 1.5× faster and MA-SGD becomes 1.7× faster, and MLlib-SGD evens gets slower. This is far below the 4× speedup one would expect if the scalability were linear.

The poor scalability comes from two reasons:

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-SGD 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 ×. Clearly, communication overhead starts to dominate the time cost. This is interesting—it indicates that using more machines may not always be a good choice.
2. Workers in these systems need to synchronize in 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 ML. 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.

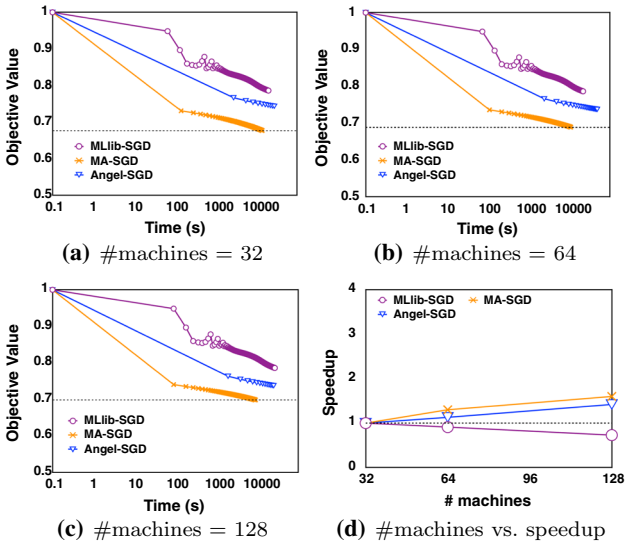
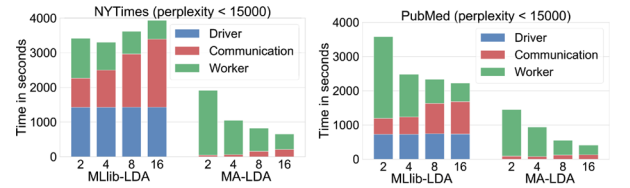


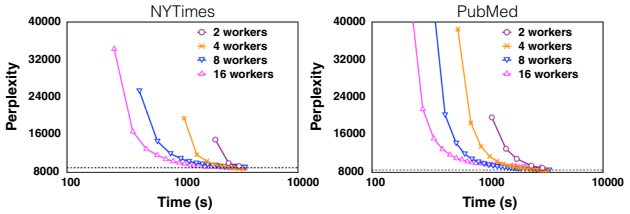
Fig. 9 Comparison of three systems MA-SGD, MLlib-SGD, and Angel-SGD on WX. The dotted lines in Fig. 9a–c represent the best objective values achieved among the systems under the corresponding experimental settings. Figure 9d shows the speedups of these systems with respect to the number of machines, normalized by the time cost using 32 machines. Note that the x-axes are in log scale

Variational inference. For scalability comparison, our experiments are in twofolds: (1) the speedup to achieve a specific perplexity with the increasing number of workers and (2) the impact on the inference quality.

Figure 10a shows that MA-LDA scales better than MLlib-LDA with more workers. In MLlib-LDA, the driver spends



(a) Scalability Comparison



(b) Quality of MA-LDA

Fig. 10 Scalability comparison of MA-LDA and MLlib-LDA

Table 6 Comparison of MA-SGD and MA-LDA with single-node baselines

	Machine × core	`url`	`kddb`
MA-SGD	8 × 1	(0.89 s, 0.54 s)	(5.45 s, 4.07 s)
VW	1 × 1	(7.1 s, 0 s)	(44.3 s, 0 s)
DW	1 × 8	(1.048 s, 0 s)	(0.99 s, 0 s)
	Machine × core	**`NYTimes`**	**`PubMed`**
MA-LDA	8 × 1	(683 s, 136 s)	(460 s, 115 s)
Gensim	1 × 8	(3836 s, 0 s)	(6937 s, 0 s)

Vowpal Wabbit(VW) and DimmWitted(DW) run single-node SGD. Gensim-LDA runs single-node LDA. Each cell presents the time spent on individual task. For example, (0.89 s, 0.54 s) for MA-SGD on the `url` dataset indicates that computation per epoch takes 0.89 second and communication per epoch takes 0.54 second

significant time to compute the Dirichlet expectation of the model, which cannot be scaled with more workers. On the other hand, although we can alleviate the computation work on the individual worker when more workers are involved, communication overhead may increase to override the benefits. The scalability of MA-LDA is much better than that of MLlib-LDA. The adoption of model averaging significantly reduces the communication overhead, whereas the distributed aggregation strategy based on AllReduce can easily scale out with more workers.

To see the impact on the quality of inference when scaling out, we run MA-LDA until it converges with different numbers of workers on `NYTimes` and `PubMed`. Figure 10b presents the result. As we can see, having more workers leads to faster convergence, whereas the final perplexity remains acceptable.

5.4 Comparison with single-node ML systems

Given the intensive resource consumption of MA-SGD, MA-LDA, and other distributed ML systems, it is natural to ask whether it is even worth to consider them. That is, would a single-node ML system perform equally well? Clearly, there is some trade-off here for a given dataset in terms of computational efficiency and resource allocated. However, it is difficult to predict which setting we should resort to. In the following, we offer some insights into this trade-off perspective using empirical results.

Gradient descent. We compare MA-SGD with Vowpal Wabbit (VW) and DimmWitted (DW) [46]. We ran MA-SGD with eight machines, each with only one CPU core. On the other hand, we ran Vowpal Wabbit and DimmWitted using one single machine but with eight CPU cores. Therefore, the computation resource in terms of CPU cores are the same in MA-SGD and single-node ML systems, for a fair comparison. Table 6 presents the computation resource in terms of CPU cores, as well as the time spent on computation and communication per epoch, respectively.

MA-SGD outperforms VW in terms of time per epoch. The rationale is the following. First, VW adopts an online algorithm, which means that it needs to stream input data from disk. Second, VW only uses a single core for training.

The benefit of DimmWitted mainly comes from intelligent uses of the hardware, in particular CPU cores and main memory. DimmWitted also designed special data and model replication mechanisms for efficient training. Each CPU core is assigned to one partition of the data (e.g., 1/8 of the dataset in our experiments), and the model is updated by the gradients calculated by all CPU cores in parallel. As given in Table 6, we allocated enough memory in both cases so that the datasets can be hosted in memory, and the training time of DimmWitted is less than MA-SGD on two small datasets.

We further compare the *scalability* of MA-SGD with single-node ML systems (e.g., DimmWitted), using various datasets with increasing sizes. Specifically, we sample data instances from the dataset `kdd12` to generate different datasets while maintaining the same model size. Figure 11 presents the results.

We observe that DimmWitted scales well when dataset can reside in memory. However, when dataset is larger than the available amount of memory, the performance of DimmWitted degrades significantly as the overhead of accessing data on disk offsets the CPU overhead of computing gradients.

On the other hand, MA-SGD does not suffer from the scalability issue observed on DimmWitted due to insufficient memory, as long as the total combined amount of available memory of all workers exceeds the size of the dataset. Although one may argue that DimmWitted could have used a more powerful machine with memory size equal to that used by MA-SGD, such a machine would be more expensive compared to the combined cost of all nodes used by MA-SGD

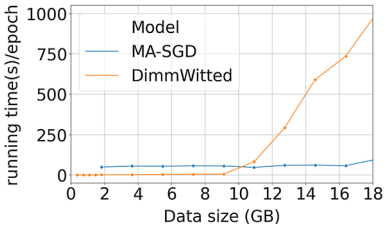


Fig. 11 Comparison between MA-SGD and DimmWitted with increasing data sizes

(as an example, check out Amazon EC2 on-demand instance pricing available at [1]).

Variational inference. Gensim [34] is a single-node system that processes SVI-LDA with multiple cores. To compare the performance of Gensim-LDA and MA-LDA, we run algorithms until the models converge to the same perplexity. As given in Table 6, we have decomposed the execution time of MA-LDA into computation time and communication time. Gensim-LDA streams the input data from disk, so the execution time includes both data loading time and computation time.

Regarding computation time, MA-LDA is 15.1 × faster than Gensim-LDA on `PubMed` and is 5.6 × faster on `NYTimes` when using 8 workers. For each data point in a batch, MA-LDA calculates a gradient and updates the local model. MA-LDA utilizes the sparse property so that the updating cost is correlated with the number of nonzero features in the data point. Therefore, MA-LDA is more efficient when the dataset is sparse, e.g., `PubMed`. In contrast, Gensim-LDA updates the model only once in each iteration, and the *DirichletExpectation* step is the most expensive part of the update process. Gensim-LDA amortizes this cost over the gradient computing of data points in a batch, regardless of sparseness.

Summary. The above study reveals a trade-off between single-node and distributed machine learning systems facing different workloads. Briefly speaking, the single-node machine learning library is a better option when a single machine is able to hold the dataset, whereas the distributed counterpart is favorable when the dataset cannot fit in the storage space of a single machine.

6 Related work

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

Distributed stochastic gradient descent. Robbins and Monro [35] proposed SGD, which has been one of the most popular optimization algorithms for training ML models. In practice, one usually uses mini-batch SGD [9,14], which has been implemented in popular libraries such as TensorFlow [2], XGBoost [11], and PyTorch [33].

Model averaging (MA) was initially studied by Zinkevich et al. [51], in which they proposed one-shot SGD with model averaging for convex problems and proved its convergence. For non-convex problems, Zhang et al. [48] evaluated the performance of MA on deep models.

Much work has also been done on other aspects, such as data compression [26], data sketch [21], and gradient quantization [23], to improve the performance of MGD. Moreover, there is also recent work devoted to fault tolerance in distributed MGD [5,39].

Computation and communication trade-off. To balance the trade-off between computation and communication, there are several 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 [30] argues that distributed computing should at least beat the single machine implementation. Huang [19] uses the small number of machines as much as possible to ensure the performance and efficiency.

Second, there are many proposals for reducing communication costs by performing local computation as much as possible. For example, Grape [15] 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 [18] is a geo-distributed machine learning system using parameter servers. It tries to reduce communication costs by favoring communications within local-area networks over wide-area networks. The parallel SGD and model averaging techniques in MA-SGD fall into the same ballpark—it performs as many local model updates as possible within every 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 [32,38,41].

Parameter server versus 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 synchronously or asynchronously 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. For instance, Li et al. [27] proposed executing MGD on parameter servers. Petuum [40] and Angel [22] are two general-purpose ML systems using the parameter-server architecture. There is also previous work on using parameter servers to implement LDA. For

example, YahooLDA [4] partitions and distributes the data-level matrix across parameter servers. LightLDA [42] uses parameter servers to store the topic assignment matrix. Both systems implement LDA based on MCMC.

Another popular architecture for distributed machine learning is `AllReduce` [37]. 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 few systems use `AllReduce` for training linear models. Agarwal et al. [3] combined two MPI primitives, `TreeAggregation` and `Broadcast`, to implement the `AllReduce` primitive, which is indeed the implementation already used by MLlib. As we have noted in Sect. 3, after we introduce model averaging, the master of MLlib becomes a communication bottleneck. As a result, we have to re-implement the `AllReduce` mechanism, using the model-partitioning techniques. The optimization strategy used by [3] is also different from ours. Agarwal et al. [3] proposed a *hybrid* optimization method for SGD and L-BFGS, where L-BFGS is the core technique and the results from executing (a single iteration of) model averaging of SGD are merely used as an initial starting point for L-BFGS. We have indeed compared our approach with this hybrid approach (Fig. 5).

Other machine learning systems on Spark. Kaoudi [24] built a cost-based optimizer to choose the best gradient descent plan for a given workload. In our work, we use the grid search to find the best parameters for each workload and thus do not need the optimizer. Anderson [6] integrated MPI into Spark and offloads the workload to an MPI environment. They transfer the data from Spark to MPI environment, use high-performance MPI binaries for computation, and finally copy the result back to the distributed file system for further usage. It is definitely interesting and worthwhile to compare Spark's RPC-based communication cost with that of using a native MPI implementation of `AllReduce`. We believe that using native MPI implementation can further reduce Spark's communication cost, though it is challenging to integrate existing MPI libraries into Spark.

7 Conclusion

In this paper, we have focused on the Spark ecosystem and studied how to run ML workloads more efficiently on top of Spark. With a careful study over implementations of Spark MLlib, we identified its performance bottlenecks. Utilizing *model averaging* (MA) and other optimization techniques enabled by MA (e.g., `AllReduce`-style communication primitives for exchange of local models), we can significantly improve MLlib without altering Spark's architecture. We present two specific case studies by implementing MA-based techniques on top of MLlib's implementations for SGD and LDA. Throughout an extensive experimental evaluation over both public and industrial workloads, we have demonstrated that, the two MA-based variants, MA-SGD and MA-LDA, not only outperform their counterparts in MLlib, but also perform similarly to or better than state-of-the-art distributed ML systems that are based on parameter servers, such as Petuum and Angel.

Acknowledgements This work is funded by the National Natural Science Foundation of China (NSFC) Grant No. 61832003 and U1811461, and Chinese Scholarship Council.

References

1. Amazon EC2 on-Demand Instance Pricing. https://aws.amazon.com/ec2/pricing/on-demand/
2. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: a system for large-scale machine learning. In: OSDI, pp. 265–283 (2016)
3. Agarwal, A., Chapelle, O., Dudík, M., Langford, J.: A reliable effective terascale linear learning system. J. Mach. Learn. Res. **15**(1), 1111–1133 (2014)
4. Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., Smola, A.J.: Scalable inference in latent variable models. In: WSDM, pp. 123–132. ACM (2012)
5. Alistarh, D., Allen-Zhu, Z., Li, J.: Byzantine stochastic gradient descent. In: Advances in Neural Information Processing Systems, pp. 4613–4623 (2018)
6. Anderson, M., Smith, S., Sundaram, N., Capotă, M., Zhao, Z., Dulloor, S., Satish, N., Willke, T.L.: Bridging the gap between HPC and big data frameworks. Proc. VLDB Endow. **10**(8), 901–912 (2017)
7. Bernardo, J.M., et al.: Psi (digamma) function. Appl. Stat. **25**(3), 315–317 (1976)
8. Boden, C., Spina, A., Rabl, T., Markl, V.: Benchmarking data flow systems for scalable machine learning. In: Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, pp. 1–10 (2017)
9. Bottou, L.: Large-scale machine learning with stochastic gradient descent, pp. 177–186 (2010)
10. Bottou, L.: Stochastic gradient descent tricks. In: Neural Networks: Tricks of the Trade, pp. 421–436. Springer (2012)
11. Chen, T., Guestrin, C.: Xgboost: a scalable tree boosting system. In: SIGKDD, pp. 785–794 (2016)
12. Chen, W., Wang, Z., Zhou, J.: Large-scale l-BFGS using mapreduce. In: Advances in Neural Information Processing Systems, pp. 1332–1340 (2014)
13. Dai, J., Wang, Y., Qiu, X., Ding, D., Zhang, Y., Wang, Y., Jia, X., Zhang, C., Wan, Y., Li, Z., et al.: Bigdl: a distributed deep learning framework for big data. Preprint arXiv:1804.05839 (2018)
14. Dekel, O., Gilad-Bachrach, R., Shamir, O., Xiao, L.: Optimal distributed online prediction using mini-batches. Journal of Machine Learning Research **13**(Jan.), 165–202 (2012)
15. Fan, W., Xu, J., Wu, Y., Yu, W., Jiang, J., Zheng, Z., Zhang, B., Cao, Y., Tian, C.: Parallelizing sequential graph computations. In: SIGMOD, pp. 495–510 (2017)
16. Foulds, J., Boyles, L., DuBois, C., Smyth, P., Welling, M.: Stochastic collapsed variational Bayesian inference for latent Dirichlet allocation. In: SIGKDD, pp. 446–454. ACM (2013)
17. Hoffman, M., Bach, F.R., Blei, D.M.: Online learning for latent Dirichlet allocation. In: NIPS, pp. 856–864 (2010)

18. Hsieh, K., Harlap, A., Vijaykumar, N., Konomis, D., Ganger, G.R., Gibbons, P.B., Mutlu, O.: Gaia: Geo-distributed machine learning approaching {LAN} speeds. In: NSDI, pp. 629–647 (2017)
19. Huang, Y., Jin, T., Wu, Y., Cai, Z., Yan, X., Yang, F., Li, J., Guo, Y., Cheng, J.: Flexps: flexible parallelism control in parameter server architecture. Proc. VLDB Endow. **11**(5), 566–579 (2018)
20. Jiang, J., Cui, B., Zhang, C., Yu, L.: Heterogeneity-aware distributed parameter servers. In: SIGMOD, pp. 463–478 (2017)
21. Jiang, J., Fu, F., Yang, T., Cui, B.: Sketchml: accelerating distributed machine learning with data sketches. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1269–1284 (2018)
22. Jiang, J., Yu, L., Jiang, J., Liu, Y., Cui, B.: Angel: a new large-scale machine learning system. Natl. Sci. Rev. **5**(2), 216–236 (2017)
23. Jiang, P., Agrawal, G.: A linear speedup analysis of distributed deep learning with sparse and quantized communication. In: Advances in Neural Information Processing Systems, pp. 2525–2536 (2018)
24. Kaoudi, Z., Quiané-Ruiz, J.A., Thirumuruganathan, S., Chawla, S., Agrawal, D.: A cost-based optimizer for gradient descent optimization. In: SIGMOD, pp. 977–992. ACM (2017)
25. Kucukelbir, A., Ranganath, R., Gelman, A., Blei, D.: Automatic variational inference in Stan. In: NIPS, pp. 568–576 (2015)
26. Li, F., Chen, L., Zeng, Y., Kumar, A., Wu, X., Naughton, J.F., Patel, J.M.: Tuple-oriented compression for large-scale mini-batch stochastic gradient descent. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1517–1534 (2019)
27. Li, M., Anderson, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.Y.: Scaling distributed machine learning with the parameter server. In: OSDI, pp. 583–598 (2014)
28. Liu, D.C., Nocedal, J.: On the limited memory BFGS method for large scale optimization. Math. Program. **45**(1–3), 503–528 (1989)
29. Liu, X., Zeng, J., Yang, X., Yan, J., Yang, Q.: Scalable parallel EM algorithms for latent Dirichlet allocation in multi-core systems. In: WWW, pp. 669–679 (2015)
30. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what {COST}? In: HotOS (2015)
31. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al.: Mllib: machine learning in Apache Spark. J. Mach. Learn. Res. **17**(1), 1235–1241 (2016)
32. Onizuka, M., Fujimori, T., Shiokawa, H.: Graph partitioning for distributed graph processing. Data Sci. Eng. **2**(1), 94–105 (2017)
33. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, pp. 8024–8035 (2019)
34. Řehůřek, R., Sojka, P.: Software framework for topic modelling with large corpora. In: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, pp. 45–50. ELRA, Valletta, Malta (2010). http://is.muni.cz/publication/884893/en
35. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat. **22**, 400–407 (1951)
36. Stich, S.U.: Local SGD converges fast and communicates little. In: ICLR 2019 International Conference on Learning Representations, CONF (2019)
37. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. Int. J. High Perform. Comput. Appl. **19**(1), 49–66 (2005)
38. Ueno, K., Suzumura, T., Maruyama, N., Fujisawa, K., Matsuoka, S.: Efficient breadth-first search on massively parallel and distributed-memory machines. Data Sci. Eng. **2**(1), 22–35 (2017)
39. Xie, C., Koyejo, S., Gupta, I.: Zeno: Distributed stochastic gradient descent with suspicion-based fault-tolerance. In: International Conference on Machine Learning, pp. 6893–6901 (2019)
40. Xing, E.P., Ho, Q., Dai, W., Kim, J.K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., Yu, Y.: Petuum: a new platform for distributed machine learning on big data. IEEE Trans. Big Data **1**(2), 49–67 (2015)
41. Xu, N., Chen, L., Cui, B.: LogGP: a log-based dynamic graph partitioning method. Proc. VLDB Endow. **7**(14), 1917–1928 (2014)
42. Yuan, J., Gao, F., Ho, Q., Dai, W., Wei, J., Zheng, X., Xing, E.P., Liu, T.Y., Ma, W.Y.: Lightlda: big topic models on modest computer clusters. In: World Wide Web, pp. 1351–1361 (2015)
43. Yut, L., Zhang, C., Shao, Y., Cui, B.: LDA*: a robust and large-scale topic modeling system. Proc. VLDB Endow. **10**(11), 1406–1417 (2017)
44. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**(10–10), 95 (2010)
45. Zaheer, M., Wick, M., Tristan, J.B., Smola, A., Steele, G.: Exponential stochastic cellular automata for massively parallel inference. In: Artificial Intelligence and Statistics, pp. 966–975 (2016)
46. Zhang, C., Ré, C.: Dimmwitted: a study of main-memory statistical analytics. Proc. VLDB Endow. **7**(12), 11 (2014)
47. Zhang, H., Zeng, L., Wu, W., Zhang, C.: How good are machine learning clouds for binary classification with good features? In: SoCC, p. 649 (2017)
48. Zhang, J., De Sa, C., Mitliagkas, I., Ré, C.: Parallel SGD: When does averaging help? Preprint arXiv:1606.07365 (2016)
49. Zhang, K., Alqahtani, S., Demirbas, M.: A comparison of distributed machine learning platforms. In: ICCCN, pp. 1–9 (2017)
50. Zhang, Y., Jordan, M.I.: Splash: User-friendly programming interface for parallelizing stochastic algorithms. Preprint arXiv:1506.07552 (2015)
51. Zinkevich, M., Weimer, M., Li, L., Smola, A.J.: Parallelized stochastic gradient descent. In: NIPS, pp. 2595–2603 (2010)