Minimizing Training Time of Distributed Machine Learning by Reducing Data Communication

Yubin Duan, *Student Member, IEEE,* Ning Wang, *Member, IEEE,* Jie Wu, *Fellow, IEEE*

Abstract—Due to the additive property of most machine learning objective functions, the training can be distributed to multiple machines. *Distributed machine learning* is an efficient way to deal with the rapid growth of data volume at the cost of extra inter-machine communication. One common implementation is the *parameter server* system which contains two types of nodes: worker nodes, which are used for calculating updates, and server nodes, which are used for maintaining parameters. We observe that inefficient communication between workers and servers may slow down the system. Therefore, we propose a graph partition problem to partition data among workers and parameters among servers such that the total training time is minimized. Our problem is NP-Complete. We investigate a two-step heuristic approach that first partitions data, and then partitions parameters. We consider the trade-off between partition time and the saving in training time. Besides, we adopt a multilevel graph partition approach to fit the bipartite graph partitioning. We implement both approaches based on an open-source parameter server platform—PS-lite. Experiment results on synthetic and real-world datasets show that both approaches could significantly improve the communication efficiency up to 14 times compared with the random partition.

Index Terms—Data communication, data sparsity, distributed machine learning, graph partition, parameter server framework

1 Introduction

With the explosive growth of data volume, accelerating the distributed machine learning has attracted more and more attention. Machine learning algorithms usually iteratively use data to update parameters. The training data usually consists of texts, audios, or images. Parameters of machine learning models usually represent the feature weights. For example, in spam classification, word counts are usually used as features. Parameters are used to adjust the influence of different words. The volume of data and parameters in real-world applications is too large to be processed by a single machine. For example, ImageNet [1] contains more than 1 million labeled images. Although the computational demand increases rapidly, the improvement of the computational power of a CPU/GPU unit is almost stagnant. Driven by this dilemma, researchers focus on developing *distributed machine learning* frameworks [2], [3], [4], [5], [6] to efficiently distribute training workloads among multiple machines.

We focus on the *parameter server framework* [4], [5] which is an efficient deployment of distributed machine learning systems. It has been implemented or adopted in both academia [7], [8] and industry [9], [10], [11]. A typical parameter server contains server nodes (or simply *servers*) that store globally shared parameters and worker nodes (or simply *workers*) which are allocated with data. The division and allocation of parameters and data samples is done by the *scheduler*. From the learning-application point of view, the training of a large machine learning model can be split into sub-problems which are assigned to workers. Before training, the scheduler would assign a subset of data (model parameters) to each worker (server). All data samples and parameters would be allocated without duplication. The data and parameter allocation is known by all machines and would not change. During training, workers would iteratively: 1) *pull* parameters from servers, 2) *calculate* parameter updates locally, and 3) *push* new parameter values to servers. We denote these three steps as one round. Servers would aggregate updates from workers and maintain the globally shared parameters in either a synchronous or asynchronous way. Note that servers and workers could be co-located in the same *machine*, as shown in Fig. 1, to exploit the fast inner-machine communication. Steps 1 and 3 involve communication via either the memory bus (inner-machine) or Ethernet (inter-machine). Step 2 needs computational power from CPU/GPU.

We notice that the communication time between different machines in the parameter server accounts for a large proportion among the total training time. We aim to reduce the communication delay between server nodes and worker nodes.We consider the synchronous scheme for parameter updating, since asynchronous updating may diverge some machine learning algorithms [12]. Specifically, a machine learning server usually contains multiple physical machines. The computation units such as CPU cores are connected with high-speed I/O buses. However, different machines are usually connected via Ethernet cables, which are usually much slower than I/O buses. The inter-machine communication and synchronization usually causes the bottleneck of the parameter server system. Typical machine learning algorithms could lead to massive network traffic because of

- *Y. Duan and J. Wu are with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, 19122. E-mail: yubin.duan@temple.edu, jiewu@temple.edu.*
- *N. Wang is with the Department of Computer Science, Rowan University, Glassboro, New Jersey 08028. E-mail: wangn@rowan.edu.*
- *This research was supported in part by NSF grants CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, CNS 1651947, and CNS 1564128.*

Manuscript received January 18, 2021 and revised April 15, 2021

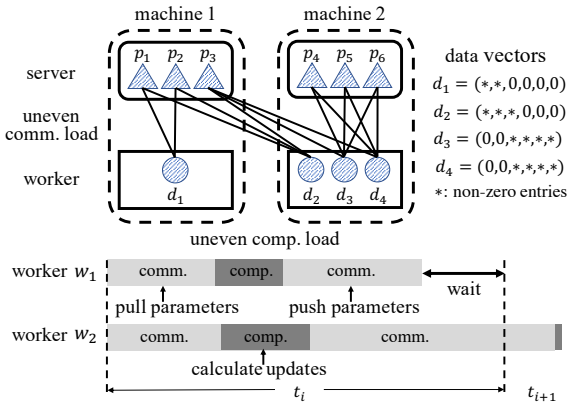


Fig. 1. Training on the parameter server framework.

the large number of iterative parameter updates. The communication volume could be 100 times larger than the data size [13]. For example, in a text classification application, the data size is 10.95 GB. If we randomly allocate the data samples to 16 machines, the network communication volume is about 0.9 TB. Even if we use high-speed networks of 40 Gbps bandwidth, the communication time is not negligible. These factors, especially inter-machine communication, lead to the communication bottleneck. The negative impact of the communication and synchronization has been shown in [12], [14], [15], and reducing the cost is challenging.

To deal with the communication bottleneck, existing approaches include compressing the communication volume [16], [17], investigating stale synchronous schemes [11], [12], etc. However, these approaches do not consider the inherent *sparsity* of data samples. The sparsity means that many data samples contain zero attributes, such as d_1 and d_2 shown in Fig. 1. Parameters related to these zero-valued attributes do not need to be sent among workers and servers [5], [18]. The data sparsity brings opportunities for reducing the uneven communication volume by data partition, while the balanced partition problem is usually NP-complete [19]. A good partition could migrate some of the inter-machine communication to inner-machine communication. For example, if we move the d_2 in Fig. 1 from machine 2 to machine 1, then there is no need to send p_1 and p_2 from machine 1 to machine 2. In this way, the inter-machine communication volume can be reduced.

Furthermore, in the synchronous scheme, data load balancing is critical and should also be considered in partitioning. When training a machine learning model, workers usually process data samples in batches. The number of rounds that can be executed at each worker node is controlled by the number of data samples allocated to it. Workers with more data need more rounds of synchronous to terminate, which also enlarges the overall training time.

An example in Fig. 1 shows the importance of the data and parameter partitions. First, the data partition affects the communication volume. We use a spam classification example to show the relationship between the data samples and model parameters. The data allocated to each worker is a word vector. An element in the vector represents the count of the corresponding word in a data sample. A zero-count word provides no additional information to the spam classification. As shown in Fig. 1, the last four elements in data d_1 are zero. It means that the last four words are not included in the sample email. If a word does not appear in a email, it makes no contribution to the model training. Therefore, the corresponding parameter that indicates the weight of the word would not be updated. If d_1 is allocated to worker w_1, then w_1 only needs to pull two parameters from servers, one for each non-zero entry. In contrast, the data allocated to w_2 has fewer zero entries, especially in d_3 and d_4. Consequently, w_2 needs to pull six parameters. This leads to uneven communication loads of different workers. Besides, the parameter partition impacts the inter-machine communication volume, which is 3 in the example, i.e., transferring p_1, p_2 and p_3 to the server in machine 1 and the worker in machine 2. Moving p_3 to the server on machine 2 could reduce the inter-machine communication volume to 2. Furthermore, the data partition also affects the computational load among workers. In this example, w_1 is allocated with one data sample while w_2 has three data samples. As a result, w_1 needs one round to finish while the other worker needs three rounds. This uneven computational load wastes the computational resources. Above all, the worker w_2 has a heavier communication burden and may need more rounds to finish. It becomes the bottleneck of the system.

To make full use of the data sparsity, we propose to efficiently partition data among workers and parameters among servers such that the training time of machine learning modes is minimized. We investigate a two-step heuristic that first partitions data and then partitions parameters. In data partition, we study the trade-off between time complexity and performance. Specifically, we use greedy heuristics for data partition, and progressively allocate unassigned data samples to server nodes. In each iteration of data allocation, one or more data samples can be considered. If more data samples are considered, the number of possible allocations becomes larger, while it is more likely to avoid the local optima. In an extreme case, if all unassigned data samples are investigated at the same iteration, the optimal assignment could be found, while the time complexity would be exponential. For example, considering two data samples in each iteration can bring up to 10% improvement compared with only adding one. Besides, we also adapt a multilevel graph partition approach to directly partition both data and parameters.

Our contributions are summarized as follows:

- We propose a data and parameter partition problem to reduce the inter-machine communication and minimize the training time for the parameter server framework under the synchronous parallel scheme.
- We investigate a two-step heuristic and the trade-off between partitioning complexity and its performance. We also analyze a theoretical bound for the data partition step based on the submodularity.
- We further adapt a multilevel partition approach of general graphs. Unlike the two-step heuristic, there is no priority for data and parameter partitions.
- Experiments on an open-source parameter server framework show algorithm performances on synthetic and real-world datasets with different sparsity.

2 Related Work

Based on classical distributed system design [20], there are several salable frameworks for the distributed machine learning [5], [9], [10], [11], [13], [21], [22], including parameter server [5], [9], [10], [11], [13] for machine learning algorithms, and MXNet [22] for deep learning algorithms. Our paper is based on the parameter server framework. Open source parameter server frameworks include PS-lite [5], Petuum [11], YahooLDA [9], etc. The parameter server has also been widely implemented in industry by Google [10], Microsoft, etc. The parameter server is first proposed for specific applications, such as YahooLDA [9] and DistBelief [10]. For example, YahooLDA is optimized for Latent Dirichlet Allocation [23]. Petuum [11] takes the first step to build a general platform. PS-lite [5] further optimizes the communication and improves the system performance.

However, the communication cost still brings inefficiency to these systems. To improve system performance, researchers have attempted several approaches to reduce the communication cost. [16], [17] propose to compress the communication volume. [11], [12] investigate stale synchronous schemes. They focus on the trade-off between the model convergence speed and synchronous time consumption. However, these approaches did not make full use of the data sparsity. [24] investigates the communication volume optimization problem for geo-distributed machine learning jobs in the parameter server architecture. The authors focus on the online resource placement problem for multiple jobs. In this paper, we consider reducing the communication time cost for a job by wisely partitioning training data and model parameters over workers and servers based on data sparsity.

The graph partition schemes have been studied in [18], [19], [25], [26], [27]. These schemes can be grouped as two categories: the vertex partition and the edge partition. The vertex partition problem usually refers to equally partitioning a set of vertices into k parts such that the number of edges spanning different partitions is minimized [19]. The edge partition is defined in an analogous way. [28] introduces a Kernighan-Lin heuristic approach that has been deployed in a widely used software called METIS [25]. These methods did not consider the possible aggregation of the partition cost. The edge partition problem was originally proposed in [27]. [19] further proposes an approximation algorithm and considers the cost aggregation of each partition. However, these methods did not focus on bipartite graphs. The Parsa proposed in [18] considers the bipartite graph partition. However, the optimization problem formulated in the Parsa is not to directly minimize the overall training time. In our paper, we model the training time consumption. Besides, we investigate a direct graph partition approach.

Besides machine learning systems deployed in data centers, the distributed learning frameworks for wireless networks attract more and more attentions because of the quick development of wireless communication technologies. Chen *et. al.* proposed a distributed learning framework which optimizes the computation offloading for resource orchestration in multi-access edge computing [29]. More generally, federated learning [30], [31] enables multiple mobile devices to train a machine learning model locally without sharing data. One statistical challenge of federated learning is that local data on mobile devices are non-IID. Zhao *et. al.* [32] proposed to overcome the issue by sharing a small set of global data between all devices. Wang *et. al.* [33] proposed to intelligently select different mobile devices to participate in each round of training with reinforcement learning. Another challenge is the security concerns caused by untrasted participants. Zhang *et. al.* [34] proposed a commitment-based scheme to select specific data for the assurance of the high confidence of participates. Besides, resource-constrained devices restrict the scale of federated learning models. To make large CNN training affordable, He *et. al.* [35] investigated a group knowledge transfer training approach that reduces local computation workload while maintaining model accuracies.

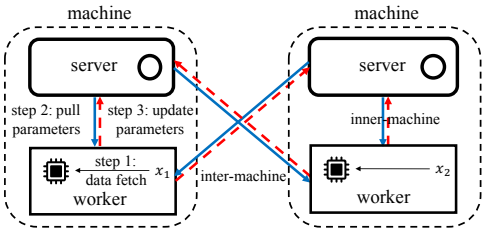


Fig. 2. A training round in the parameter server framework.

3 Model

3.1 Distributed machine learning

Typical machine learning algorithms can be viewed as minimizing an objective function. The additive property of its objective function makes distributed machine learning possible. The objective function of supervised learning usually corresponds to a measure of the prediction error, such as the inaccuracy of classifying spam emails. Let $D = \{d_1, d_2, \ldots, d_n\}$ denote the set of data, and d_i be the i-th data which is usually a m-dimensional vector containing m feature values. Let $P = \{p_1, p_2, \ldots, p_m\}$ denote the set of parameters in the machine learning algorithms. These parameters form a weight vector $W = [\omega_1, \omega_2 \ldots, \omega_m] \in \mathbb{R}^m$, which is used to adjust the weights of m features in data. The prediction error of learning algorithms with parameter W can be measured by a loss function $l(d_i, W)$, for example, l_1 loss - least absolute deviations, or l_2 loss - least square errors, etc. Learning algorithms iteratively update the parameter vector W and try to minimize the risk function:

$$Loss(W) = \sum_{i=1}^{n} l(d_i, W) + \Omega(W), \tag{1}$$

where $\Omega(W)$ is the regularization term. It is usually used to penalize the learning model complexity and to avoid overfitting. The risk function value is aggregated from the losses of all data. The aggregation could be allocated to multiple machines if the data is allocated correspondingly.

To minimize the risk function, one common optimization algorithm is the gradient descent. For example, when training a logistic regression model, the gradient descent algorithm is used to minimize the cross-entropy loss. In gradient descent, we need to iteratively calculate the gradient of the loss function w.r.t. model parameters and update the

parameters based on the gradient. Formally, let $\nabla Loss(W_t)$ denote the gradient of the loss function at iteration t. Then, we need to update parameters for iteration $t+1$ as $W_{t+1} = W_t - \eta \nabla Loss(W_t)$, where η is the training rate. Because of the additive property of the loss function shown in Eq. (1), the calculation of the gradient $\nabla Loss(W_t)$ can be distributed to multiple workers. Specifically, $\nabla Loss(W_t) = [\partial Loss/\partial w_1, \partial Loss/\partial w_2, \ldots, \partial Loss/\partial w_m]$, where w_i is the i-th element of vector W_t. The calculation of $\partial Loss/\partial w_i$ can be reorganized such that multiple workers can run in parallel:

$$\begin{aligned}\frac{\partial Loss}{\partial w_i} &= \frac{\partial\Big[\sum_{j=1}^{n} l(d_j, W) + \Omega(W)\Big]}{\partial w_i} \\ &= \sum\nolimits_{j=1}^{n} \frac{\partial l(d_j, W)}{\partial w_i} + \frac{\partial \Omega(W)}{\partial w_i} \\ &= \frac{\partial\Big[\sum_{j=1}^{n_1} l(d_j, W)\Big]}{\partial w_i} + \frac{\partial\Big[\sum_{j=n_1+1}^{n_2} l(d_j, W)\Big]}{\partial w_i} + \cdots \\ &\quad + \frac{\partial\Big[\sum_{j=n_{k-1}+1}^{n} l(d_j, W) + \Omega(W)\Big]}{\partial w_i}, \end{aligned} \tag{2}$$

where $n_1, n_2, \ldots n_{k-1}$ are the indexes that partition the data samples. The calculation of partial derivatives of $\frac{\partial[\sum_{j=1}^{n_1} l(d_j,W)]}{\partial w_i}, \frac{\partial[\sum_{j=n_1+1}^{n_2} l(d_j,W)]}{\partial w_i}, \ldots, \frac{\partial[\sum_{j=n_{k-1}+1}^{n} l(d_j,W)+\Omega(W)]}{\partial w_i}$ can be assigned to worker $w_1, w_2, \ldots, w_k$, respectively. Similarly, the calculation of other gradient elements can be reorganized for multiple workers. In this way, the training process is distributed to different machines.

3.2 Parameter server framework

We integrate the idea of data and parameter partition to the parameter server framework [4]. The parameter server framework consists of servers, workers, and a scheduler. Servers are in charge of maintaining globally shared parameters of machine learning models. Workers use subsets of data to solve sub-problems and update parameters. Each actual machine contains a server and a worker. All data samples and parameters of the learning model would be divided without duplication into workers and servers, respectively. The assignment is done by the scheduler before the actual training starts and it is known to all machines. Besides, the assignment is fixed during the training process.

The communication is usually the bottleneck of the parameter server. Let k denote the number of machines. Let D_i and P_i denote the data allocated to the worker and server in the i-th machine, respectively. Before training, the scheduler allocates data and parameters among available machines. Procedures of a training round are illustrated in Fig. 2. Workers would fetch their data, pull the globally shared parameters from servers (solid lines), calculate the update of parameters, and push the updated parameters back to servers (dashed lines). The allocation of data samples and parameters assigned by the scheduler is fixed during training. The worker and server located in the same machine communicate via memory buses, which is the *inner-machine communication*. Workers and servers in different machines communicate via network infrastructures, such as Ethernet, which is the *inter-machine communication*. Compared with the inter-machine communication, the time cost of inner-machine communication is usually negligible. Therefore, we focus on the inter-machine communication volume in further analysis.

TABLE 1
Table of Notations

Notations	Description
D, P	the data set and the parameter set
$G(V, E)$	the correlation graph G, $V = D \cup P$,$E \subseteq D \times P$
$N(D')$	the neighbor set of data set $D' \subseteq D$
$\lvert \cdot \rvert$	the cardinality of a set
$f(D')$	the cardinality of neighbor set of D'
k	the number of machines
n, m	the number of training samples and attributes

3.3 Data sparsity in inference problems

Inference problems benefit from data sparsity when they are solved in a distributed manner. Specifically, when updating parameter values with data d_i, only non-zero attributes in the data change the parameter value [18], since zero elements bring no information for the interface. For example, in spam prediction, if a word does not exist in a spam e-mail, we cannot determine whether this word is more frequent in spam or non-spam e-mails, and cannot update the prediction model accordingly. If a worker finds that some attributes of its d_i are zero, then there is no need to pull parameters corresponding to these attributes from servers. The communication volume of this worker is reduced correspondingly. Therefore, if major attributes in input data samples are zero, the communication volume of training inference learning models could be significantly reduced. In practice, the parameter server framework supports to push or pull part of the parameters during training [5]. Workers can skip parameters that correspond to zero data attributes during training. The allocation of data samples and parameters is critical for balancing the communication volume among machines, and significantly affects the training time.

3.4 Network model

We construct a bipartite graph $G(V, E)$ to model the communication correlation. $V = D \cup P$ is the vertex set and $E \subseteq D \times P$ is the edge set. The scheduler in the parameter server framework would build the graph by scanning the non-zero entries of data $d \in D$. Specifically, we can use a hash map to store the graph. Each key represents a data sample and the corresponding value stores a list of correlated parameters. During data prepossessing, we scan and count the non-zero entries of each data vector and use indices of those entries to update the hash map. Using the hash map, we can finish the scan in linear time, and the scanning time is a part of the completion time. The scheduler also needs to decide which portion of data/parameters should be stored in each machine. The partition would be broadcast to all workers, and workers would know where to pull/push parameters during training. We aim to optimize the partition such that the training time of the model is minimized. Fig. 3 shows the correlation graph G. There is an edge between d_i and p_j only if the j-th entry of d_i is non-zero. Let $N : 2^D \to 2^P$ denote the *neighbor set* function

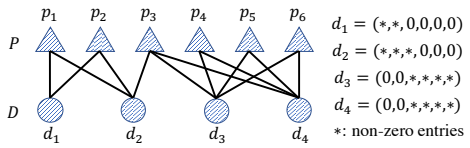


Fig. 3. The correlation between data and parameters.

which is used to find the set of parameters that are co-related with data. Formally, $N(D') = \{p \in P | (d,p) \in E, d \in D'\}$, where $D' \subseteq D$. For example, $N(\{d_1\}) = \{p_1, p_2\}$ and $N(\{d_1, d_2\}) = \{p_1, p_2, p_3\}$. Let $f : 2^D \rightarrow \mathbb{N}$ denote the neighbor set cardinality, i.e. $f(D') = |N(D')|$.

The inter-machine communication volume contains two parts: the *worker communication* volume and the *server communication* volume. The worker communication volume of the i-th machine is $|N(D_i) \setminus P_i|$. It represents the volume of parameters pulled/pushed by the worker at machine i. Specifically, $N(D_i)$ is the set of parameters needed by the worker. Among them, some may be stored in the server on the i-th machine. The difference set of $N(D_i)$ and P_i is the set of parameters that are transmitted between machines. The server communication volume of the i-th machine is $\sum_{j \neq i} |P_i \cap N(D_j)|$. It means the volume of parameters pulled/pushed by workers in other machines. Specifically, the worker in the j-th machine needs to access the value of parameters in $N(D_j)$, and parameters in set $P_i \cap N(D_j)$ are stored in the i-th machine. Therefore, the communication volume between P_i and D_j is $|P_i \cap N(D_j)|$. Accumulating over all machines other than i, we have the inter-machine server communication volume. Above all, the inter-machine communication volume of the i-th machine is $|N(D_i) \setminus P_i| + \sum_{j \neq i} |P_i \cap N(D_j)|$.

3.5 Problem formulation

In this paper, we propose to minimize the training time on the parameter server by taking advantage of data sparsity. We consider that the parameter update is synchronous among all machines, given that some learning algorithms may diverge in the asynchronous way [12].

We quantify the training time based on the number of rounds of synchronization and the inter-machine communication volume. Our formulation does not contain the computational time in each round since it is hardly changed by data partitions. We assume the time consumption of each round is proportional to the largest communication volume among machines. As for the number of rounds, typical machine learning algorithms would terminate when the loss function is sufficiently small, i.e. less than a given threshold, or when all the training data is used. For the worst case analysis, we assume that each worker terminates when all of the data assigned to it is used. Then, the number of rounds is determined by the worker who has the largest amount of data samples. Formally, it is $\max_i |D_i|$. An unbalanced data partition would slow down the training process. Solely optimizing inter-machine communication or the number of rounds cannot guarantee the minimal training time. For example, if all data samples and parameters are assigned to the same machine, there is no inter-machine communication. However, it is obviously not optimal since all other machines are idle. Therefore, our objective function can be formulated as $\max_i |D_i| \cdot \max_i (|N(D_i) \setminus P_i| + \sum_{j \neq i} |P_i \cap N(D_j)|)$. The first term $\max_i |D_i|$ shows the upper bound of the number of iterations. The other term is the upper bound of the communication volume in an iteration. Their product represents the upper bound of the training time in the worst case.

Admittedly, the time consumption of computing the parameter updates in each round also contributed to the total training time. However, for each training round, the computation time of a machine learning algorithm is mainly determined by the input batch size. The training batch size is usually a hyper-parameter determined by users. Once the batch size is given, it will be consistently applied to all worker machines. Therefore, the partition policy of data samples and model parameters can hardly affect the computation time in a training round. Therefore, we assume the computation time is a constant value related to training algorithms, and omit the term in the problem formulation.

The problem can be formulated as follows:

$$\min \max_i |D_i| \max_i (|N(D_i) \setminus P_i| + \sum\nolimits_{j \neq i} |P_i \cap N(D_j)|), \quad (3)$$

$$s.t. |N(D_i)| \leq \theta, \forall 1 \leq i \leq k, |\bigcup_i D_i| = n, |\bigcup_i P_i| = m. \quad (4)$$

The objective is to minimize the upper bound of the overall training time. It is modeled by the upper bounds of iteration amounts and the communication volume in an iteration. $|N(D_i)| \leq \theta$ is the RAM memory constraint for each machine. We assume that the problem is feasible, i.e. the total RAM memory among machines is large enough to store the machine learning model used in training. $|\bigcup_i D_i| = n$ and $|\bigcup_i P_i| = m$ represent the allocation constraints meaning that all data and parameters should be allocated.

4 Two-Step Heuristic Partition

The objective function of our problem is a function of both set D_i and P_i. Optimizing the function is NP-complete even when assuming that the sets of P_i are given [26]. Therefore, we optimize the original problem via a two-step heuristic. In the first step, we partition the data set D. In the second step, we partition the parameter set P. Besides, the submodular property of function f could help the first step optimization. In this section, we first explain the submodular property, and then introduce the two steps of optimization.

4.1 Submodular property

Before presenting our solution to minimize the training time, we first show the submodular property of the neighbor set cardinality function, which is useful for introducing our algorithms.

Theorem 1. The neighbor set cardinality $f(\cdot)$ over the data set D is non-negative, monotone, and submodular.

Proof: It is proved by definition. The set function f is defined as the cardinality of a neighbor set. By definition, the function is non-negative. For all $D' \subseteq D'' \subseteq D$, we have $N(D') = \bigcup_{d \in D'} N(\{d\}) \subseteq \bigcup_{d \in D'} N(\{d\}) \cup \bigcup_{d \in D' \setminus D''} N(\{d\}) = N(D'')$. Therefore, we have $f(D') = |N(D')| \leq |N(D'')| = f(D'')$ implying that the function f is monotone.

Algorithm 1 Data partition

Input: Bipartite graph $G(D, P, E)$, number of machines k, subset size limitation α, subset size weight β
Output: Data partition $D_i, i = 1, 2, \ldots, k$
1: initialize $D_i \leftarrow \emptyset$ for all $i = 1, 2, \ldots, k$
2: Unassigned data set $D' \leftarrow D$
3: **while** D' is not empty **do**
4: find the data partition $D_i \leftarrow \arg\min_i |D_i|$
5: $S \leftarrow \arg\min_{S \subseteq D', |S| \leq \alpha} f(D_i \cup S) - \beta \cdot |D_i \cup S|$.
6: **if** $|D_i \cup S| > \lceil n/k \rceil$ **then**
7: $S \leftarrow \arg\min_{d \in S} f(D_i \cup d)$
8: assign all $d \in S$ to partition D_i
9: $D' \leftarrow D' \setminus S$
10: $N(D_i) \leftarrow N(D_i) \cup N(S)$
11: **return** $D_i, i = 1, 2, \ldots, k$ as the data partition

To prove the submodularity, by definition, we need to show $f(D' \cup \{d\}) - f(D') \geq f(D'' \cup \{d\}) - f(D'')$ for every $D' \subseteq D'' \subseteq D$. It is also called the property of diminishing returns. Let $g(D, d) \triangleq f(D \cup \{d\}) - f(D)$. Then, we are going to show that $g(D', d) \geq g(D'', d)$.

By definition of $N(\cdot)$, we have $f(D' \cup \{d\}) = |N(D' \cup \{d\})| = |\bigcup_{d' \in D'} N(d') \cup N(d)| = |N(D') \cup N(\{d\})|$. According to the inclusion-exclusion principle, we further have that $|N(D') \cup N(\{d\})| = |N(D')| + |N(\{d\})| - |N(D') \cap N(\{d\})|$. Then, we have

$$\begin{aligned} g(D', d) &= |N(D') \cup N(\{d\})| - |N(D')| \\ &= |N(D')| + |N(\{d\})| - |N(D') \cap N(\{d\})| - |N(D')| \\ &= |N(\{d\})| - |N(D') \cap N(\{d\})|. \end{aligned}$$

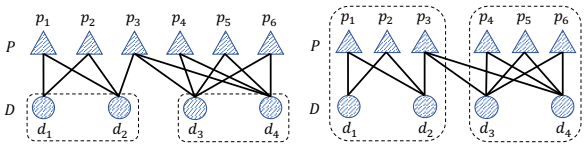
Similarly, $g(D'', d) = |N(\{d\})| - |N(D'') \cap N(\{d\})|$.

To show that $g(D', d) \geq g(D'', d)$, we now prove that $|N(D') \cap N(\{d\})| \leq |N(D'') \cap N(\{d\})|$. For all $d' \in N(D') \cap N(\{d\})$, we know that $d' \in N(D')$ and $d' \in N(\{d\})$. We have shown that $N(D') \subseteq N(D'')$. Therefore, $d' \in N(D'')$. Hence, $d' \in N(D'') \cap N(\{d\})$. It means that any element in set $N(D') \cap N(\{d\})$ is also an element of set $N(D'') \cap N(\{d\})$. Therefore, $|N(D') \cap N(\{d\})| \leq |N(D'') \cap N(\{d\})|$. Consequently, $g(D', d) \geq g(D'', d)$ holds, and f is submodular. ■

Although the submodular property of the objective function is useful for optimization, optimizing the objective function directly is complex. To minimize the training time, we propose to first partition dataset which is already NP-complete [18] and then partition the model parameters.

4.2 Data partition

In the data partition, we focus on minimizing the maximum neighbor set size in different machines. Formally, our objective of data partition is $\min \max_i |N(D_i)|$. The insight is that we hope to balance the number of parameters needed by each worker without considering the difference between inner-machine and inter-machine communication. Also, we need to balance the dataset size allocated to each machine. This is because the overall training time is related with the $\max_i |D_i|$ as illustrated in Eq. (1). Since $\sum_i |D_i|$ is a constant, $\max_i |D_i|$ could be minimized when the data samples are evenly allocated to all workers. Therefore, when balancing the neighbor set $N(D_i)$ among all machines, we don't want to break the balance among sizes of data sets D_i. To ensure the balance of D_i, we set a constraint that $|D_i| \leq \lceil n/k \rceil, i = 1, 2, \ldots, k$ for the data partition. The min-max submodular partition is NP-complete [36]. The submodular approximation algorithm [37] shows an $O(\sqrt{n/\log n})$-approximation for the problem. Their algorithm initializes all data as unassigned. Then, they iteratively choose the worker with the smallest partition, and the worker is assigned to the subset of unassigned data that introduces the smallest increase on its neighbor size. The subset of data is searched by sampling from all unassigned data. Different from their approach, we consider limiting the size of the set of data added in each iteration. We investigate the trade-off between the algorithm's time complexity and its performance. In addition, we show a theoretical bound for our algorithm from a different aspect as [37].



(a) The data partition (b) The parameter partition

Fig. 4. Illustration of the two-step heuristic.

The procedure of our algorithm is illustrated in Alg. 1. First we initialize each data partition as an empty set in line 1, i.e. every $d \in D$ is unassigned. Let D' store the set of unassigned data, and it is initialized as D in line 2. Then, we iteratively assign data to the machine with the smallest partition until all data is assigned. Specifically, in each iteration, the algorithm chooses the machine i with the smallest $|D_i|$ as shown in line 4. In line 5, a subset of unassigned data is found such that the data assignment cost $f(D_i \cup S) - \beta |D_i \cup S|$ is minimized. In the cost function, $\beta \in \mathbb{R}^+$ is used to adjust the weight of data size. If β is large, then the algorithm prefers to choose a large subset. For subsets with the same size, the term $f(D_i \cup S)$ decides which is chosen. It is the one that induces the smallest increase to f, i.e. the neighbor set size. To control the time complexity of finding such a subset, the size of S is limited by α. $\alpha = 1, 2, \ldots, \lceil n/k \rceil$. For example, if $\alpha = 2$ then, the size of S is limited to 2. Note that the size of S could be 1 when $\alpha = 2$. If we strictly fix $|S| = \alpha$, then we cannot guarantee that $|D_i| \leq \lceil n/k \rceil$. The value of α can be used to adjust the time complexity and performance of the partition algorithm. Specifically, with a large α, more possible combinations will be investigated in each iteration. Hence, the algorithm is less likely to be trapped in a local optimum point. However, the time complexity increases exponentially with α. If the α is too large, the partition algorithm is no longer feasible for a large input graph. In the experiment, we compare the $\alpha = 1$ case with the $\alpha = 2$ case. Eventually, we hope the data samples could be evenly allocated among all machines, i.e. each machine should have $\lceil n/k \rceil$ number of data samples. In an effort to achieve the even allocation, if adding the chosen subset of data to D_i would exceed the number $\lceil n/k \rceil$, then we greedily choose one data sample $d \in S$ which induces the smallest increase on f and set $S = \{d\}$ in line 7. Then, the data in S would be added to D_i and $N(D_i)$ would be updated correspondingly in lines 8-10.

TABLE 2
The cost of adding data in each iteration

	d_1,d_2	d_1,d_3	d_1,d_4	d_2,d_3	d_2,d_4	d_3,d_4
Iteration 1 ($D_1 = \emptyset$)	3	6	6	6	6	4
Iteration 2 ($D_2 = \emptyset$)	-	-	-	-	-	4

We could use random sampling to reduce the running time of the Alg. 1. Considering that finding the best subset from all unassigned data is time consuming, we could sample a subset of unassigned data $T \subseteq D'$, and find S from T. The sample rate, i.e. the possibility of each element being picked up, is set to $n/(k|D'|)$ to guarantee that there are enough samples.

The submodular property brings a bound to the performance of the data partition. In most real-world applications, $k \ll n$, i.e. the number of machines used for distributed machine learning is far less than the number of data. Therefore, compared with n, k can be treated as constant. Besides, the approximate ratio provided in [37] is based on the assumption that $k = \Theta(n/\log n)$. We propose to provide an approximation analysis with respect to k.

Theorem 2. For number of machines $k = \Theta(\sqrt{n/\log n})$, the data partition result of Alg. 1 could guarantee that $\max_i f(D_i) \leq \Theta(\sqrt{n/\log n})OPT$, where OPT is $\max_i f(D_i^*)$, i.e. the worst cost in the optimal partition.

Proof: Let D_i denote the data partition found by our algorithm. We have shown that the set function f is monotone. This property remains for $\alpha = 1, 2, \ldots, \lceil n/k \rceil, \beta \in \mathbb{R}^+$. Hence, after data partition with Alg. 1, $D_i \subseteq D$ for $i = 1, \ldots, k$. Consequently, we have that $f(D) \geq f(D_i)$ for $i = 1, \ldots, k$. Therefore, $f(D) \geq \max_i f(D_i)$.

From Theorem 1, we know that the set function f is submodular. By definition, we have that for any $D', D'' \subseteq D$, $f(D') + f(D'') \geq f(D' \cup D'') + f(D' \cap D'')$. Assume there exists a data partition $D_i^*, i = 1, \ldots, k$, such that the data is equally partitioned and $\max_i f(D_i^*)$ is minimized among all possible partitions. For any $D_i^*, D_j^* \subseteq D$, we have that $D_i^* \cap D_j^* = \emptyset$. By definition, $f(\emptyset) = 0$. Therefore, we have that $f(D_i^*) + f(D_j^*) \geq f(D_i^* \cup D_j^*) + f(\emptyset) = f(D_i^* \cup D_j^*)$. Based on this inequation, we have

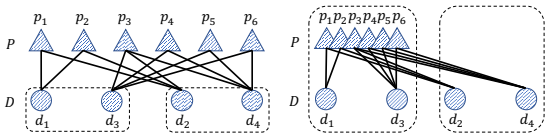
$$\begin{aligned}\sum_{i=1}^{k} f(D_i^*) &= f(D_1^*) + f(D_2^*) + \cdots + f(D_k^*) \\ &\geq f(D_1^* \cup D_2^* \cup \cdots \cup D_k^*) = f(D). \end{aligned} \quad (5)$$

Combining it with $f(D) \geq \max_i f(D_i)$, we have that

$$\sum_{i=1}^{k} f(D_i^*) \geq \max_i f(D_i). \quad (6)$$

Furthermore, we have that $\max_i f(D_i^*) \geq f(D_i^*), \forall i = 1, \ldots, k$. Correspondingly, $k \cdot \max_i f(D_i^*) \geq \sum_{i=1}^{k} f(D_i^*)$. Combining it with the inequality shown above, we have that $\max_i f(D_i) \leq k \cdot \max_i f(D_i^*)$. If $k = \Theta(\sqrt{n/\log n})$, then we have $\max_i f(D_i) \leq \Theta(\sqrt{n/\log n})OPT$. ■

We use an example in Fig. 4(a) to show the data partition with $\alpha = 2$. We allocate at most two data samples to a machine in each iteration. For simplicity, we assume the sample rate is 1, meaning that all data are considered. In addition, the value of β is large enough, say $\beta = n$, that adding two data in each iteration always has lower cost than adding one data. Therefore, we always add two data in each iteration unless the number of data samples on the machine exceeds $\lceil n/k \rceil = 2$. The cost of adding data could be simply evaluated by the neighbor set size f. Table 2 shows the cost of adding data in each iteration. Initially, all data are unassigned. In each iteration, a machine with the smallest data size is chosen and the pair of data that induces the smallest increase on neighbor set size, or simply the cost, is chosen. In the first iteration, both $|D_1|$ and $|D_2|$ are zero. The tie is broken by machine labels, and machine 1 is chosen. Then adding data d_1, d_2 to D_1 has the smallest cost. Therefore, they are assigned to machine 1. Then in the second iteration, machine 2 is chosen since $|D_2| = 0$. Then d_3 and d_4 are chosen and assigned to machine 2. In this assignment, $\max_i |N(D_i)| = 4$.



(a) The data partition (b) The parameter partition

Fig. 5. Adding one vertex leads to a worse performance.

TABLE 3
The cost of adding data in each iteration

	d_1	d_2	d_3	d_4
Iteration 1 ($D_1 = \emptyset$)	2	3	4	4
Iteration 2 ($D_2 = \emptyset$)	-	3	4	4
Iteration 3 ($D_1 = \{d_1\}$)	-	-	6	6
Iteration 4 ($D_2 = \{d_2\}$)	-	-	-	6

The effectiveness of setting multiple data can be shown by using the following example. Adding only one vertex in each iteration reduces the algorithm time complexity, but may also reduce its performance. An example is shown in Fig. 5(a). We set $\alpha = 1$ and only one data can be added in each iteration, which is equivalent to a data partition implementation in [18]. In the first iteration, machine 1 is chosen for $|D_1| = 0$. Following the cost given in Table 3 (We only need to consider $f(D_i \cup S)$ since $|D_i \cup S|$ could be constant.), d_1 is chosen and added to D_1. In the second iteration, machine 2 is chosen and d_2 is assigned to it. In the third iteration, $|D_1| = |D_2| = 1$. Again, machine 1 is chosen for the smaller label. The cost of adding d_3 and d_4 are the same. We still break the tie by the index, and d_3 is added. In the final round, d_4 is added to machine 2. In this assignment, $\max_i |N(D_i)| = 6$, which is worse than the previous assignment.

4.3 Parameter partition

After the partition of D is determined, we partition P among servers. The objective is to minimize the inter-machine communication volume since the value of $\max_i |P_i|$ has been determined, or formally, $\min \max_i (|N(D_i) \setminus P_i| + \sum_{j \neq i} |P_i \cap N(D_j)|)$ Besides, the memory constraint could be removed for the parameter partition. It should have been satisfied in the data partition whose objective is $\min_i |N(D_i)|$. Otherwise, the input scenario as infeasible.

Algorithm 2 Parameter partition

Input: Computation graph $G(D, P, E)$, number of machines k, and the data partition D_i
Output: Parameter partition $P_i, i = 1, 2, \ldots, k$
1: $P_i \leftarrow \emptyset$ for all $i = 1, 2, \ldots, k$
2: $P' \leftarrow P$
3: **for** $i = 1, 2, \ldots, k$ **do**
4: initialize the partition cost, $cost(P_i) \leftarrow |N(D_i)|$
5: **while** P' is not empty **do**
6: choose the partition P_i with the smallest cost
7: $p^* \leftarrow \arg\min_{p \in N(D_i)} cost(P_i \cup p^*) - cost(P_i)$
8: assign p^* to partition P_i.
9: remove p^* from P'
10: **return** $P_i, i = 1, 2, \ldots, k$ as the data partition

Also, the parameter partition would not change the value of $|N(D_i)|$. Therefore, the memory constraint can be omitted.

We solve the parameter partition with a greedy approach. Initially, servers are given empty parameter sets. Then, in each iteration, we choose the server on the machine that has the smallest cost which is the inter-machine communication volume $|N(D_i) \setminus P_i| + \sum_{j \neq i} |P_i \cap N(D_j)|$, and then give it an unassigned parameter that would induce the smallest increase on the cost. The intuition is to balance the inter-machine communication while keeping the increase as low as possible. The procedures are shown in Alg. 2. Specifically, lines 1-4 initialize the parameter partition and the cost. P' stores unassigned parameters. The initial cost is $|N(D_i)|$ since $P_i \cap N(D_i) = \emptyset$, i.e. all communication is inter-machine. In line 6, the algorithm chooses the machine with the smallest cost. In line 7, an unassigned parameter p^* is greedily chosen. To reduce search space, we only consider $p \in N(D_i)$, i.e., parameters that have correlation with data in the machine. Lines 8-9 update P_i, $cost(P_i)$, and P'. The allocation repeats until all parameters are assigned.

Different data partitions would impact the parameter partition. Fig. 4(b) and Fig. 5(b) illustrate the parameter partition results based on the different data partitions. Clearly, the example in Fig. 4(b) has less inter-machine communication, which is only one parameter p_3 transferring between two machines. In contrast, all six parameters need to be pulled/pushed between two machines.

Theorem 3. The worst-case time complexity of the parameter partition is $O(m(n + m))$, where $n = |D|$ and $m = |P|$.

Proof: The initialization in lines 1-2 takes linear time. For the while loop, we first analyze the time cost in each iteration. Choosing P_i costs at most $O(k)$. Choosing p^* in line 7 costs at most $O(n + km)$. Specifically, calculating set D_i costs at most $O(n)$ and the cardinality of D_i is $O(m)$. The increase on cost function could be calculated by querying the correlation $D_i \cap p$ for $i = 1, \ldots, k$. Therefore, it costs $O(km)$ time to find p^* from $O(m)$ candidates. In lines 8-9, assigning and removing can be finished in $O(1)$ time. The while loop has $O(m)$ iterations. Consequently, the while loop costs $O(m(k+n+km+1)) = O(m(n+m))$. The worst case time complexity of the parameter partition is $O(m(n+m))$. ■

In real-world applications, the graph is usually sparse and the worst case is not likely to occur. The complexity analysis for data partition is analogous, except the time of finding the best subset S depends on α.

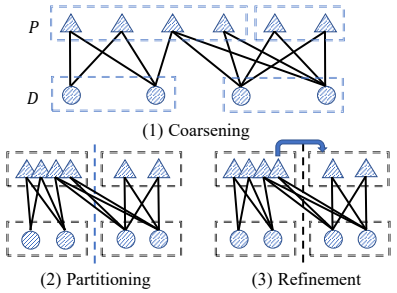


Fig. 6. Illustration of the multilevel approach.

5 Directly Partition the Graph

Besides the two-step heuristic, we also adapt a multilevel approach [38] to directly partition the graph, i.e., partition both data and parameters at the same time. The multilevel approach is designed to partition general graphs and we adapt the algorithm for bipartite graph partition.

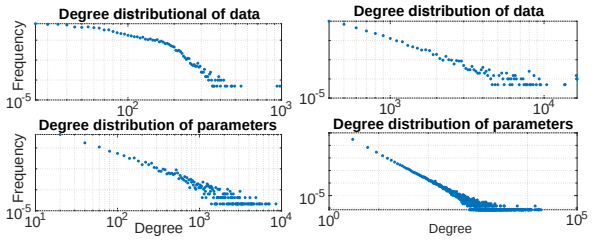
The multilevel approach contains three stages: coarsening, partitioning and refinement. An illustration is shown in Fig. 6. Unlike the two-step heuristic approach, there is no priority for data partition or parameter partition. In the coarsening stage, the graph nodes are iteratively merged and a coarse graph that has a small number of nodes is generated. In the partitioning stage, the coarse graph is partitioned. In the refinement stage, the algorithm would refine the partition and recover the coarse graph to the original graph.

In the coarsening stage, unlike the multilevel approach for general graph, we merge data set D and parameter set P respectively. The merging procedure for D is explained as follows. 1) Mark all $d \in D$ as available. 2) Randomly choose an available $d_i \in D$; if there is no available d_i, then the coarse is finished. 3) Find an available $d_j \in D$ such that d_j and d_i have at least one same neighbor. Formally, there exists a $p \in P$ such that $p \in N(\{d_i\})$ and $p \in N(\{d_j\})$. If there is no such a d_j, then d_i is marked as unavailable. 4) Merge d_i and d_j, and label both as unavailable. Then repeat steps 2-4 until there are no more available nodes. The merging procedure for P is similar. Finally, the coarse graph with a small number of vertices is generated.

On the coarse graph, any partition method can be applied. However, since the coarse graph has a small number of vertices, we use a random partition method to save partition time. Then the refinement stage starts. Our refinement algorithm is based on the FM-algorithm [39] whose main idea is local search. The FM-algorithm is designed for bisection partition in general graphs. We adapt the algorithm to k-way partition for bipartite graphs. The idea is to sweep over data and parameters and record the *gain* of moving each to other machines. The gain refers to the decrease in the inter-machine communication volume. We move data/parameter to the machine that brings the largest gain, , as long as the moving would not break the memory constraint in Eq. (2). The data or parameter node with larger gain has higher priority for moving, i.e. would be

TABLE 4
Dataset Statistic

Dataset	#Samples	#Attributes	Mean Degree
news20	1.5×10^4	6.2×10^4	18
rcv1.binary	2.0×10^4	4.7×10^4	30
CTRa	1.0×10^6	4.0×10^6	48
soc-liveJournal	4.8×10^6	4.8×10^6	14
synthetic-sparse	1.0×10^4	1.0×10^4	10
synthetic-medium	1.0×10^4	1.0×10^4	30
synthetic-dense	1.0×10^4	1.0×10^4	50



(a) The rcv1 dataset (b) The news20 dataset

Fig. 7. The node degree distribution (real-world datasets).

moved first. The gain is updated after each movement. Each data/parameter is moved at most once to reduce time complexity. The algorithm stops when either there is no improvement or each data and parameter node have been moved once.

Theorem 4. The worst-case time complexity of the refinement algorithm is $O((m+n)|E|)$.

Proof: To calculate the gain of moving data $d \in D_i$ to D_j, we need to go through its neighbor set $N(d)$ and find out the value of $|N(d) \setminus P_i| - |N(d) \setminus P_j|$. Consequently, calculating gains for all $d \in D$ needs to go through all edges in the graph for finding neighbor sets of all $d \in D$, and costs $O(|E|)$. It is similar for gain calculation of parameter nodes. Therefore, finding the max gain of moving the d among k machines needs $O(k|E|)$ time. Moving a data or parameter node costs $O(1)$ times. Hence, moving nodes and updating gains costs at most $O(|E|)$. The moving would repeat for at most $(m+n)$ rounds. Therefore, the worst case time complexity is $O((m+n)|E|)$. ■

6 Experiment

6.1 Dataset

The dataset we use is summarized in Table 4. The first four are real-world datasets. Specifically, The news20 and rcv1.binary are text datasets. The CTRa is a click-through dataset from anonymous Internet companies. These datasets are stored in libsvm format [40]. The number of data corresponds to the size of set D in our model and the number of features corresponds to the size of P. The soc-liveJournal is a social network dataset. The social network dataset is a general graph $G' = (V', E')$ instead of a bipartite graph, but it can be easily converted. Specifically, we treat the nodes in the dataset as both set D and P, i.e. $D = P = V'$. If there is an edge between two nodes in G', we connect the corresponding nodes in D and P. In this way, the original graph could be converted to a bipartite graph.

TABLE 5
The performance improvement with 8 partitions

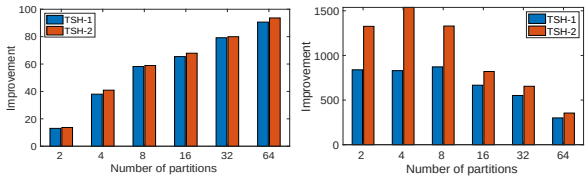
Dataset	Improvement over random partition (%)			
	TSH-1	TSH-2	MBP	PaToH
news20	184	193	175	106
rcv1.binary	109	111	103	42
CTRa	906	951	1371	607
soc-liveJournal	175	183	82	193

Besides the real-world dataset, we also build three sets of synthetic data with different data sparsity. Specifically, the number of correlations between data and parameters is represented by a matrix in the synthetic dataset, where rows represent data and columns represent parameters. The value of each element in the matrix could be either 0 or 1, showing whether there is a correlation between the corresponding data and parameter. The sparsity is controlled by a sparsity coefficient γ, which is defined as the percentage of zero elements in the matrix. The sparsities of sparse, medium, and dense synthetic datasets are 0.999, 0.995 and 0.991, respectively. The synthetic dataset may have no meaning from the machine learning point of view. It is because the data does not contain any real-world meanings. The purpose the of synthetic dataset is used to test the partition algorithm. During experiments, we ignore the result of the machine learning algorithms. Instead, we set a threshold on the the maximum number of iterations of training models, and want to compare the time used for training under different data or parameter partitions.

In both real-world and synthetic datasets, the scheduler of the parameter server would build the dependency graph by one round of traversal over data samples. The dependency graph represents the correlation between data and parameters. It starts from a bipartite graph with no edges. Two sets of vertices in the bipartite graph represent sets of data samples and parameters. During traversal, for each data sample, if there is a non-zero attribute, it adds one corresponding edge to the bipartite graph. After the traversal, the graph partition algorithms is called. After the partition, the scheduler would allocate the parameters and data samples to servers and workers, respectively, based on the partition result. The dependency graph and the partition result would be stored in a lookup table. In this way, the partition is also known to all workers. In each training round, workers can quickly acquire the graph and the partition. Therefore, they know where to pull/push the parameters of the learning model.

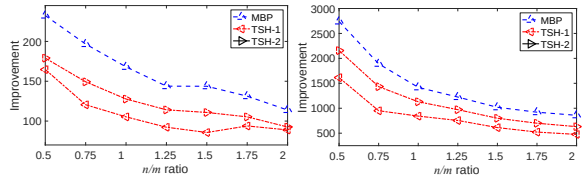
6.2 Experiment setup

In our experiment, we implement the graph partition on a local machine and then the learning algorithms on a cluster which is deployed in Amazon EC2. Our local machine has a 6-core i7-8700 CPU running at 3.2GHz with 32 GB RAM. The cluster on EC2 contains 16 t2.large instances. Each t2.large instance contains 8GB RAM and runs on Ubuntu 18.04. The parallel computing environment is set up in all instances. Specifically, each instance is installed with the OpenMPI library and its dependent packages. We deploy PS-lite, an open-source parameter server framework that is available at https://github.com/dmlc/ps-lite, on the cluster as the



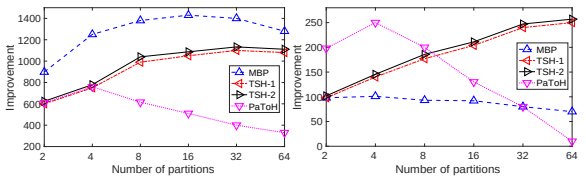
(a) Sparsity = 0.991 (b) Sparsity = 0.999

Fig. 8. Varying data sparsity on synthetic dataset.



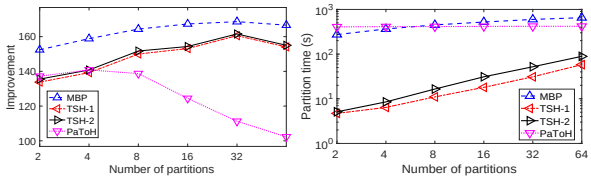
(a) Synthetic-medium dataset (b) Synthetic-sparse dataset

Fig. 9. Varying the number of data and parameters.



(a) CTRa dataset (b) soc-liveJournal dataset

Fig. 10. Improvement in communication volume over random partition.



(a) Training time (b) Partition time

Fig. 11. Comparisons in training and partition time on CTRa dataset.

platform to train machine learning algorithms. The instance where the partition algorithm is deployed is labeled as the scheduler. Each instance contains server and worker nodes.

On the local machine, we compare the performance of graph partition. The baseline is the random partition scheme. In this scheme, data and parameters are uniformly and randomly allocated to each machine. We implement the two-step heuristic and investigate the trade-off between its time complexity and performance. Specifically, in the data partition, we compare the difference between adding one vertex to a machine in each iteration (denoted as TSH-1) and adding two vertices (denoted as TSH-2). The multilevel bipartite partition approach is denoted as MBP. We also test a popular graph partition algorithm, PaToH [41], which is a well-optimized hypergraph partitioner and supports taking a bipartite graph as the input.

The inter-machine communication volume and the training time are impacted by the partition algorithm and the number of partitions. To clearly show the improvement caused by graph partition algorithms under different numbers of partitions, we use relative improvement over the random partition as the metrics to evaluate the performance of different algorithms. This metric helps cancel out the effect of the number of partitions. The relative improvement is defined as $(C_{\text{random}} - C_{\text{comparison}})/C_{\text{comparison}} \times 100\%$, where C_{random} represents the cost of random partition and $C_{\text{comparison}}$ represents the cost of partition found by comparison algorithms. The cost represents the inter-machine communication volume when we compare the performance of different partition algorithms in Figs. 8, 9, and 10. When we comparing the training time in Fig. 11, the cost represents the overall time consumption of both the graph partition phase and the model training phase. The partition result on the local machine is stored in recordio format.

6.3 Experiment results

Table 5 shows the performance of algorithms over real-world datasets. The performance is evaluated by the improvement on the inter-machine communication volume over random partitioning. The number of partitions is 8, i.e. k = 8. From the table, we can find that the improvement of TSH-2 is 1.83% – 4.97% higher than that of TSH-1. MBP outperforms TSH on the CTRa dataset, but underperforms TSH on the rcv1.binary and soc-liveJournal datasets. This is due to the different data and parameter degree distribution of these datasets. Moreover, we analyze the training time distribution on the rcv1.binary dataset. If we randomly assign data samples and parameters to 8 workers and using $\epsilon = 0.0001$ as the stopping criteria, the training time of rcv1.binary dataset is 181.1s. Among them, communication takes 170.5s. Using TSH-1, we can reduce the training time to 18.0s, which includes 1.2s of prepossessing and partition time. The model accuracy is not changed when using different graph partition algorithms. Accuracies on rcv1.binary and new20 datasets are 97.6% and 86.0%, respectively. In the following experiments, we investigate different factors that may impact these algorithms' performance.

We first test the impact of data sparsity based on our synthetic datasets. The experiment results are shown in Fig. 8. In the synthetic-dense dataset whose $\gamma = 0.991$, the improvement of TSH-2 compared with TSH-1 is not obvious, no matter what the number of partitions is. The reason for this is that $|N(D_i)|$ easily becomes large when the total number of edges in G is large. In an extreme case, if $\gamma = 0$, i.e. the bipartite graph is fully connected, there is no way to achieve any improvement by graph partition. The improvement of TSH-1 compared with the random partition is in the range of 13.0% – 90.6%, and it increases if the number of partitions k increases. In the synthetic-sparse dataset with $\gamma = 0.999$, the performance gap between TSH-2 and TSH-1 is obvious. When $k = 4$, TSH-2 outperforms TSH-1 by 85.4%. It shows that TSH-2 could make better use of the data sparsity than TSH-1. However, when the number of partitions becomes larger than 4, the performance of TSH-1 and TSH-2 decreases. This is because each machine would only be assigned very few data and parameters. The difference between partition algorithms becomes smaller. In the extreme case when $k = n$, each machine only contains one

data and there is no difference between partition algorithms.

In addition, we investigate whether the ratio between the number of data and parameters impacts the performance of partition algorithms in terms of the inter-machine communication volume. We extract a part of data from the synthetic datasets. The number of extracted data varies from 50% to 200% of the parameter amount. Fig. 9 shows the simulation results in communication volume. With larger n/m ratio, the algorithms' performances decrease. But the decrease rate is smaller when the ratio of n/m increases. It shows that when the number of data is smaller, the partition algorithm plays a more important role. This is because the $|N(D_i)|$ is smaller when the number of data is smaller. A good partition could avoid the overlap between $N(D_i)$ for $i = 1, \ldots, k$. Their performance seems to become stable when the ratio n/m is larger than 2, and they still could achieve about 100% improvement on synthetic-medium dataset and about 700% improvement on synthetic-sparse dataset compared with the random partition.

Then we evaluate the impact of k on real-world datasets. We choose to use the CTRa and the soc-LiveJournal datasets. Note that the soc-LiveJournal is originally a general graph. Experiment results are shown in Fig. 10. We find that these algorithms have better performance on the CTRa dataset. The reason for this is that the social network inherently contains many small densely connected clusters. When the number of partitions becomes larger, the performance of the MBP and the PaToH drops. The impact of k is stronger to the PaToH, whose performance begins to drop when $k > 4$. This shows the weakness of the PaToH's recursive approach that uses the bisection partition multiple times to achieve the k-way partition. Small errors in bisection partitions accumulate when k is large. The performance of TSH-1 and TSH-2 is more stable on both datasets and there is no obvious performance drop.

Besides the communication volume, we also compare the training and partition time. Fig. 11(a) shows the improvement on training time. The variation of training time is similar to that of communication volume. This is because the communication is the bottleneck and takes more time than computation. When the inter-machine communication volume is reduced, the training time is also reduced. The improvement achieved by MBP, TSH-1, and TSH-2 is between about 135% – 170% on the CTRa dataset compared with the random partition. Fig. 11(b) shows the time consumption of partition algorithms. The time consumption of TSH-1 or TSH-2 is less than 100s even for 64 partitions. The MBP or PaToH takes a longer time. Especially, MBP takes 653s to generate 64 partitions. Nevertheless, the training usually takes hours in our setting. Running our TSH algorithms with small overhead (in minutes) could significantly reduce the training time. It is worthy to run our graph partition algorithms before training.

7 Conclusion

In this paper, we investigate the potential speed-up of parameter server frameworks brought by the data sparsity, especially when training inference models. By partitioning the data among workers and parameters among servers, the inter-machine communication volume could be reduced. We formulate a graph partition problem based on this observation. The partition problem is modeled by a bipartite graph. We investigate two approaches to solve the problem. In the two-step heuristic approach, we first solve the data partition by using the submodular property of the neighbor set cardinality function, and then partition the parameters among machines. In addition, we adapt the multilevel graph partition approach for general graphs to fit the optimization over the bipartite graph. Both approaches are tested with synthetic and real-world datasets. Experiment results show that the two-step heuristic approach provides a trade-off between computational complexity and algorithm performance, which helps the algorithm adapt to different datasets. The multilevel partition approach has better performance except in a social network dataset.

Our scheme cannot be applied to all machine learning models. For deep learning models with millions or billions of parameters, the data-parameter correlation is complex. Build the correlation graph is not feasible. Instead, using hashing techniques to compress those models might be a better approach. For reinforcement learning models, data samples may not be available in advance. Reinforcement learning can explore the solution space during training and learns from the experience. For this case, our data partition scheme cannot partition the model parameters before training. It is an interesting topic to discuss the model placement problems to optimize the training time for reinforcement learning models. Additionally, our partition scheme can be extended by integrating some ideas from federated learning. For example, we can discuss the trade-off between communication latencies and learning accuracy level as indicated in [42]. Moreover, the asynchronous update can be investigated in future work.

References

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *IEEE CVPR*, 2009.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *The Very Large Data Bases Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[4] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Proceedings of ACM NIPS*, 2014, pp. 19–27.

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

[6] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1235–1241, 2016.

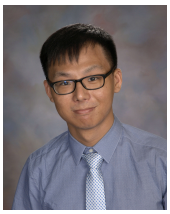
[7] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li *et al.*, "Bigdl: A distributed deep learning framework for big data," in *ACM SoCC*, 2019, pp. 50–60.

[8] G. Neglia, G. Calbi, D. Towsley, and G. Vardoyan, "The role of network topology for distributed machine learning," in *IEEE INFOCOM'19*. IEEE, 2019, pp. 2350–2358.

[9] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *ACM WSDM*, 2012, pp. 123–132.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Proceedings of NIPS*, 2012, pp. 1223–1231.
[11] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proceedings of ACM NIPS*, 2013, pp. 1223–1231.
[12] X. Zhao, A. An, J. Liu, and B. X. Chen, "Dynamic stale synchronous parallel distributed training for deep learning," in *IEEE ICDCS*, 2019, pp. 1507–1517.
[13] M. Li, "Scaling distributed machine learning with system and algorithm co-design," Ph.D. dissertation, PhD thesis, CMU, 2017.
[14] S. Shi, X. Chu, and B. Li, "Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms," in *IEEE INFOCOM*, 2019, pp. 172–180.
[15] C. Chen, W. Wang, and B. Li, "Round-robin synchronization: Mitigating communication bottlenecks in parameter servers," in *IEEE INFOCOM*, 2019, pp. 532–540.
[16] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.
[17] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Proceedings of the NIPS*, 2017, pp. 1509–1519.
[18] M. Li, D. G. Andersen, and A. J. Smola, "Graph partitioning via parallel submodular approximation to accelerate distributed machine learning," *arXiv preprint arXiv:1505.04636*, 2015.
[19] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of the ACM SIGKDD*, 2014, pp. 1456–1465.
[20] J. Wu, *Distributed System Design*. CRC Press, 1998.
[21] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging AI applications," in *USENIX OSDI*, 2018, pp. 561–577.
[22] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.
[23] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
[24] X. Li, R. Zhou, L. Jiao, C. Wu, Y. Deng, and Z. Li, "Online placement and scaling of geo-distributed machine learning jobs via volume-discounting brokerage," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 948–966, 2020.
[25] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.
[26] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.
[27] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *USENIX OSDI*, 2012, pp. 17–30.
[28] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.
[29] X. Chen, C. Wu, Z. Liu, N. Zhang, and Y. Ji, "Computation offloading in beyond 5g networks: A distributed learning framework and applications," *arXiv preprint arXiv:2007.08001*, 2020.
[30] J. Konečnỳ, B. McMahan, and D. Ramage, "Federated optimization: Distributed optimization beyond the datacenter," *arXiv preprint arXiv:1511.03575*, 2015.
[31] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM TIST*, vol. 10, no. 2, pp. 1–19, 2019.
[32] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv preprint arXiv:1806.00582*, 2018.
[33] H. Wang, Z. Kaplan, D. Niu, and B. Li, "Optimizing federated learning on non-iid data with reinforcement learning," in *IEEE INFOCOM*. IEEE, 2020, pp. 1698–1707.
[34] X. Zhang, F. Li, Z. Zhang, Q. Li, C. Wang, and J. Wu, "Enabling execution assurance of federated learning at untrusted participants," in *IEEE INFOCOM*. IEEE, 2020, pp. 1877–1886.
[35] C. He, M. Annavaram, and S. Avestimehr, "Group knowledge transfer: Federated learning of large cnns at the edge," *Proceedings of NeurIPS*, vol. 33, 2020.
[36] K. Wei, R. K. Iyer, S. Wang, W. Bai, and J. A. Bilmes, "Mixed robust/average submodular partitioning: Fast algorithms, guarantees, and applications," in *Proceedings of NIPS*, 2015, pp. 2233–2241.
[37] Z. Svitkina and L. Fleischer, "Submodular approximation: Sampling-based algorithms and lower bounds," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1715–1737, 2011.
[38] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs." *SC*, vol. 95, no. 28, pp. 1–14, 1995.
[39] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of IEEE DAC*, 1982, pp. 175–181.
[40] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.
[41] Ü. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)," *Encyclopedia of Parallel Computing*, pp. 1479–1487, 2011.
[42] N. H. Tran, W. Bao, A. Zomaya, M. N. Nguyen, and C. S. Hong, "Federated learning over wireless networks: Optimization model design and analysis," in *IEEE INFOCOM*, 2019, pp. 1387–1395.

Yubin Duan received his B.S. degree in Mathematics and Physics from University of Electronic Science and Technology of China, Chengdu, China, in 2017. He is currently a Ph.D. student in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. His current research focuses on scheduling algorithms in distributed and parallel computing.



Ning Wang is currently an assistant professor in the Department of Computer Science at Rowan University. He received his Ph.D. degree at Temple University in 2018. He obtained his B.E. degree at University of Electronic Science and Technology of China in 2013. He currently focuses on optimization problems in Internet-of-Things systems and Smart Cities applications. He has published nearly thirty papers in high-impact conferences and journals, such as, IEEE ICDCS, INFOCOM, IWQoS, IEEE TBD, JPDC, etc. He has served as a program committee member for top international conferences such as IEEE ICDCS, WCNC, etc., and reviewers for premier journals such as IEEE TPDS, TWC, TMC, TITS, TOIT, TSC, etc.

Jie Wu is the Director of the Center for Networked Computing and Laura H. Carnell professor at Temple University. His current research interests include mobile computing and wireless networks, cloud computing, and network trust and security. Dr. Wu regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Mobile Computing and IEEE Transactions on Service Computing. He is/was general chair/co-chair for IEEE MASS'06, IEEE IPDPS'08, IEEE DOCSS'09, IEEE ICDCS'13, ACM MobiHoc'14, ICPP'16, IEEE CNS'16, and WiOps'21, as well as program chair/co-chair for IEEE MASS'04, IEEE INFOCOM'11, CCF CNCC'13, and ICCCN'20. Dr. Wu is a Fellow of the AAAS and a Fellow of the IEEE.