Addressing Skewness in Iterative ML Jobs with Parameter Partition

Shaoqi Wang†, Wei Chen†, Xiaobo Zhou†, Sang-Yoon Chang†, and Mike Ji*
†Department of Computer Science, University of Colorado, Colorado Springs, USA
*School of Informatics, University of Edinburgh, UK
{swang,cwei,xzhou,schang2}@uccs.edu, mji@inf.ed.ac.uk

***Abstract*—Computational skewness is a significant challenge in multi-tenant data-parallel clusters that introduce dynamic heterogeneity of machine capacity in distributed data processing. Previous efforts to addressing skewness mostly focus on batch jobs based on the assumption that processing time is linearly dependent on the size of partitioned data. However, they are ill-suited for iterative machine learning (ML) jobs, which (1) exhibit a non-linear relationship between the size of partitioned parameters and processing time within each iteration, and (2) show an explicit binding relationship between input data and parameters for parameter update.**

In this paper, we present FlexPara, a parameter partition approach that leverages the non-linear relationship and provisions adaptive tasks to match the distinct machine capacity so as to address the skewness in iterative ML jobs on data-parallel clusters. FlexPara first predicts task processing time based on a capacity model designed for iterative ML jobs without the linear assumption. It then partitions parameters to parallel tasks through proactive parameter reassignment. Such reassignment can significantly reduce network transmission cost incurred by input data movement due to the binding relationship. We implement FlexPara in Spark and evaluate it with various ML jobs. Experimental results show that compared to hash partition, FlexPara speeds up the execution by up to 54% and 43% in private and NSF Chameleon clusters, respectively.

I. Introduction

Statistical machine learning (ML) plays an important role in modern applications and services. Recently, distributed implementation and execution of iterative ML algorithms on data-parallel clusters are increasingly common, e.g., Spark [1] and MLlib [2]. ML algorithms begin with an initial solution (called parameters) and then improve the solution to fit input data through sequential iterations [3], [4].

For cost-effectiveness and high utilization, data-parallel clusters are frequently multi-tenant [5], [6], [7]. However, it introduces dynamic heterogeneity and unpredictable performance variation of machine capacity in data processing due to resource contention and interference in the shared cloud infrastructure. Such heterogeneity yields computational skewness among parallel tasks and poses a significant challenge for iterative ML jobs. When skewness occurs, tasks on slower workers take longer to process and update their parameters than tasks on faster workers, slowing down the entire iteration execution and the job completion time.

Speculative task execution has been applied to address skewness in jobs that fit MapReduce framework [8], [9], [10]. For example, in reduce stage, the job duplicates a straggler task on a fast worker and shuffles intermediate data to that task. However, iterative ML jobs work differently. Intermediate parameters shuffled between adjacent iterations get bound with input data to ensure the convergence. The binding relationship results in that both parameters and the input are moved to the straggler task. Such data movement leads to significant network transmission cost since the input size is several orders of magnitude larger than the size of parameters.

Most efforts [11], [12], [13], [14] mitigate skewness based on the assumption that task processing time is linearly dependent on the size of partitioned data. While the assumption may hold true for map stage, it does not work well for reduce stage. Recent work [15] shows that the linear assumption can be a serious limitation when there is non-linear relationship between the processing time of reduce task and the size of partitioned intermediate data. It proposed a new task progress indicator that operates without the linear assumption. But it does not utilize the non-linear relationship to mitigate computational skewness.

As reduce stage, iterative ML jobs, such as PageRank and Topic Modeling, exhibit non-linear relationship between the size of partitioned parameters and the task processing time within each iteration. We recognize that the nature of iteration also provides the opportunity to leverage the non-linear relationship through adaptive parameter partition so as to mitigate skewness. However, leveraging the non-linear relationship in these ML Jobs should address several challenges. First, the capacity model developed in work [15] only accounts for reduce tasks in MapReduce Jobs. The model cannot be applied to iterative ML jobs since it ignores the parameter-input binding relationship. Second, parameter partition should be aware of the binding relationship so as to reduce the cost of network transmission incurred by input data movement. A detailed explanation of the binding and non-linear relationships is described in Section II-A.

In this paper, we propose FlexPara, a partition approach that embraces the non-linear relationship and accounts for parameter-input binding relationship. FlexPara includes two major components: capacity model and parameter partition. Capacity model is built to predict task processing time during the job execution. Specifically, each worker (i.e., machine) profiles the procedure of parameter processing and correlates the parameter processing time and the size of bound input. Afterward, for a task with partitioned parameters on the

worker, it calculates the processing time of each parameter and predicts the task processing time.

Based on the capacity model, parameter partition first obtains the predicted processing time of each task under hash partition, in which parameters are partitioned to the worker with the bound input (i.e., no additional input data movement). Then, parameters from slower tasks are continually and regularly reassigned to faster ones to mitigate the skewness. In contrast to previously partitions (e.g., PIKACHU [12]), this reassignment-based partition reduces the deviation from hash partition to reduce network transmission cost.

In a nutshell, we make the following contributions over previous approaches: (1) we provide a capacity model tailored for iterative ML jobs to predict task processing time without the linear assumption, and (2) we develop a novel parameter partition approach that accounts for the non-linear relationship and the parameter-input binding to efficiently address skewness in iterative ML jobs on multi-tenant clusters.

We implement FlexPara in Spark and evaluate it in clusters using various iterative ML jobs. Experimental results show that the execution time by FlexPara is faster than that by hash partition by up to 54% and 43% in private and NSF Chameleon clusters, respectively. FlexPara achieves even greater significant performance improvements when compared to PIKACHU, obtaining 1x and 72% faster execution time in private and Chameleon clusters, respectively.

The rest of this paper is organized as follows. Section II gives the analysis of iterative ML jobs and a case study to establish the relevance of FlexPara. Section III describes the design of FlexPara. Section IV gives the implementation details. Section V and Section VI present the experimental setup and evaluation results. Section VII reviews related work. Section VIII concludes the paper.

II. Motivations

A. ML Analysis

In MapReduce jobs, intermediate data generated from map tasks is partitioned and shuffled to reduce tasks. However, in iterative ML jobs, model parameters are partitioned from the previous iteration. The task first receives partitioned parameters. It then fetches the input data bound to the parameters. Finally, the task starts computation.

The definition of parameter and bound input varies in different ML jobs. Thus, we conducted analysis of Topic Modeling and PageRank jobs theoretically and experimentally.

Topic Modeling uses LDA [16] to iteratively learn the parameters of topic-word distribution until they can best explain the corpus of documents. For the entire corpus, each document has its own *parameters: Dirichlet parameter γ and multinomial parameter ϕ*. These two parameters are regarded as one parameter in the following description since they are updated together. In each iteration, the parameter is shuffled to the corresponding document (i.e., the bound input) and updated based on variational inference algorithm [16].

In study [16], the authors proposed that the total number of computations to process (i.e., update) one parameter within one iteration is about N^2K. *N* is the number of words in the document and *K* is the number of total topics. To validate this, we conducted the experiment that uses the dataset from Wikipedia [17] in Spark local mode. We ran the LDA job on two workers with different processing capacities, separately. The result is represented as a set of data points in Figure 1(a), where we used polynomial regression to plot two curves that fit these points. Moreover, since every document has a different number of words, the processing time for every parameter within the task is different. We rank the processing time of parameters in one task and set each parameter a serial number. The result is shown in Figure 1(b).

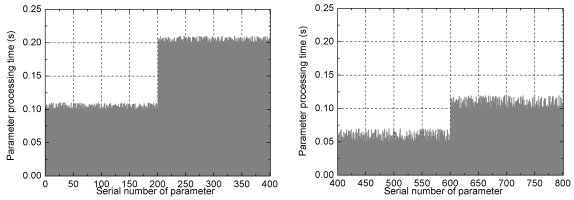
For the distributed implementation of Topic Modeling, the input data is identically distributed in clusters before the iteration begins. All parameters are then partitioned to the input in order to conduct parallel task processing during each iteration. Since the processing time for each parameter varies, there is no linear relationship between the task processing time and the number of parameters within the task. Figure 1(c) shows the cumulative processing time in order of serial numbers and the non-linear relationship.

PageRank assigns a weighted score (i.e., parameter) to every page, and the score of a page measures its importance in the web. Let *u* be a web page. Let $|N_u|$ be the number of pages *u* points to and B_u be the set of pages that point to *u*. In each iteration, its *parameter* P_u is shuffled to the set B_u *(i.e., the bound input)* and updated as $P_u = c\sum_{w \in B_u} P_w/N_w$. Variable *c* is the normalization factor, *w* is the page in B_u. We can see that the processing time for one parameter depends on the number of pages within its B_u. Since every page has different size of B_u, the processing time for every parameter is different, leading to the non-linear relationship between the processing time and the number of parameters.

B. Case Study

TABLE I
PARTITIONED INPUT PAGES.

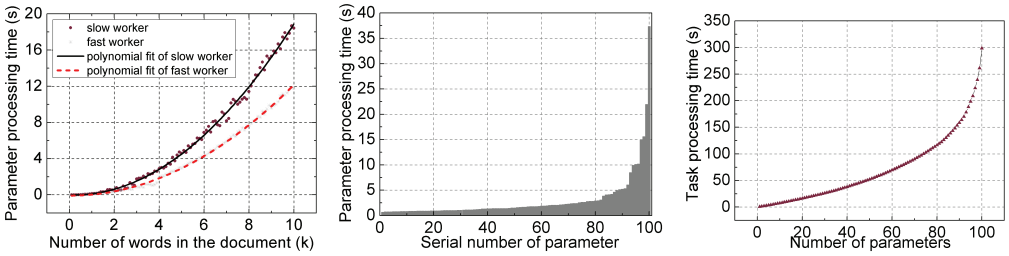
Serial number of parameters	The bound input size of each parameter	Total input size	Input location
1 - 200	2.5MB	500MB	slow worker
201 - 400	5MB	1000MB	slow worker
401 - 600	2.5MB	500MB	fast worker
601 - 800	5MB	1000MB	fast worker



(a) Slow worker. (b) Fast worker.

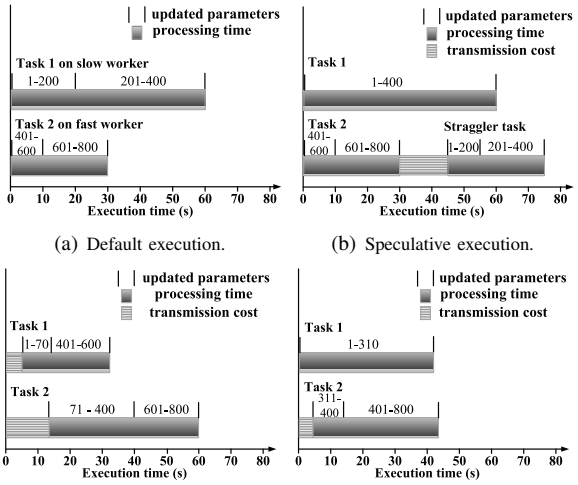
Fig. 2. Parameters processed on two workers.

We created a small 3-node Spark cluster with 800Mbps network bandwidth just for demonstration of dynamic heterogeneity and computational skewness. The 3-node cluster was configured with one master and two slave nodes (i.e., worker).



(a) Relationship between parameter processing time and size of bound input. (b) Ranked parameter processing time within one task. (c) Non-linear relationship between task processing time and number of parameters.

Fig. 1. Experimental analysis of Topic Modeling for the non-linear relationship.



(a) Default execution. (b) Speculative execution.

(c) Linear-based partition. (d) Proposed partition.

Fig. 3. Task executions.

The slave nodes are heterogeneous with one fast worker and one slow worker. The CPU processing capacity of the fast worker is twice as fast as that of the slow one.

We ran PageRank job under different skewness mitigation scenarios. Before the iteration begins, the input data bounded to each parameter is partitioned among two nodes based on hash partition as shown in Table I. Within each iteration, 800 parameters are partitioned to two workers to be updated.

In the default task execution, 800 parameters are partitioned based on hash partition as the input data. Figure 2 shows the processing time of each parameter. For parameters with 2.5MB bounded input data, the slow worker spends roughly 0.1s on one parameter among serial numbers from 1 to 200 and the fast worker spends roughly 0.05s on one parameter among serial numbers from 401 to 600. For parameters with 5MB bounded input data, each worker takes twice as much time. The default execution time of two workers is shown in Figure 3(a). In detail, tasks 1 and 2 receive 400 parameters from the previous iteration and update them on the 1,500MB size input pages separately. Due to the distinct computational capabilities, task 1 takes approximately 60s to finish while task 2 takes 30s.

Figures 3(b) to 3(d) show the performance of three skewness mitigation scenarios. Figure 3(b) uses the reactive speculative task execution method. The cluster starts a straggler task of task 1 on the fast worker after task 2 is finished. To guarantee the convergence speed, the input bounded to parameters from serial number 1 to 400 is moved from the slow worker to the fast worker, incurring network transmission cost of 15s. Although the straggler spends less time on the fast worker, the overall task execution time is longer than the default execution.

In a linear partition strategy, the processing capacities of two workers are calculated as 0.15s per parameter and 0.075s per parameter respectively, based on the statistics from the default execution. As a result, it partitions 270 parameters to the slow worker and 530 to the fast one, expecting processing time to be 40.5s and 39.75s on the two workers separately. Moreover, previous linear-based approaches (e.g., PIKACHU) are not aware of input data transmission in ML jobs. Thus, one partition scenario is shown as Figure 3(c). Such a partition results in 27s processing time on slow worker and 46.5s on fast one. Also, the slow worker spends extra 5s to transfer the input bounded to parameters 401 to 600, while the fast worker spends extra 13.25s to transfer the input bounded to parameters 71 to 400.

Figure 3(d) shows the result of the proposed partition by FlexPara in this paper. Based on the processing time of each parameter, the slow worker spends 42s in processing 310 parameters, and the fast worker spends 39s in processing 490 parameters and 4.5s in transferring the input bounded to parameters 311 to 400. The overall execution time is 37.9% faster than that of the default execution.

III. FlexPara Design

A. Optimality Problem Formulation

The goal of skewness mitigation is to minimize the overall execution time of parallel tasks. That becomes equivalent to minimizing the time of the slowest task (the one with the largest execution time):

$$\underset{\vec{V}}{argmin}\ \max_{i}\ T_{i,Exe}(V_i) \tag{1}$$

where the vector $\vec{V}$ refers to the entire set of parameters. Each of the element V_i refers to a subset of parameters partitioned to task *i*. $T_{i,Exe}(V_i)$ represents the execution time. For iterative ML jobs, it is comprised of those components from processing and networking:

$$T_{i,Exe}(V_i) = T_{i,Pro} + T_{i,Net^{pa}} + T_{i,Net^{in}} \tag{2}$$

where $T_{i,Pro}$ is processing time that is non-linearly dependent on the size of partitioned V_i; $T_{i,Net^{pa}}$ corresponds to the transmission cost (time) for parameter shuffling; and $T_{i,Net^{in}}$ is the cost for the input movement. These two network costs

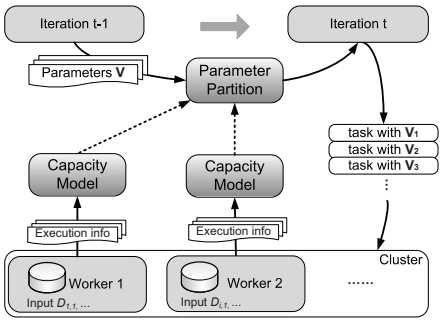


Fig. 4. The architecture of FlexPara.

are non-negligible, especially in multi-tenant clusters with unstable network bandwidth. Note that, $T_{i,Pro}$ refers to the processing time on CPU since a previous study [18] has shown that most ML jobs are CPU-intensive jobs.

FlexPara studies each performance variable in Equation 2 and solves Equation 1. Specifically, it first approximates the execution time to $T_{i,Exe}(V_i) = T_{i,Pro}$ by assuming that the network bandwidth is unlimited. Based on this assumption, FlexPara models the heterogeneous worker capacities to predict each $T_{i,Pro}$ and partitions $\overrightarrow{V}$ to balance various $T_{i,Pro}$s. It then improves the partition to further balance execution time ($T_{i,Exe}(V_i)$) by considering network transmission costs $T_{i,Net^{pa}}$ and $T_{i,Net^{in}}$.

Figure 4 shows the architecture of FlexPara. We describe their functionality as follows:

- **Capacity Model** correlates the parameter processing time to the size of bound input on the worker based on polynomial regression. It then uses the regression function to estimate the processing time of each parameter within the task so as to predict the task processing time.
- **Parameter Partition** uses capacity models from parallel workers to predict processing time of each task in the next iteration. It continually and regularly reassigns parameters from slower tasks to faster ones to mitigate the skewness.

B. Capacity Model

1) Model overview: Iterative ML jobs, such as PageRank, Matrix Factorization, and Topic Modeling, fit within one model that searches a set of parameter ***V*** to best explain or fit input data ***D***. Such jobs are usually solved by iterative algorithms that can be expressed as the following form:

$$V^{(t)} = \Delta(V^{(t-1)}, D) \tag{3}$$

where, $V^{(t)}$ is the parameters at iteration *t*, update function Δ trains the parameters from previous iteration *t*-1 on the bound input *D*. This operation repeats itself until parameters in *V* converge or meet certain requirements specified by users.

Running these jobs on data-parallel clusters often distributes the input *D* over multiple parallel workers. Thus, *V* and *D* become vectors $\overrightarrow{V}$ and $\overrightarrow{D}$. Specifically, in each iteration, the subset of data $D_{i,j}$ located on worker *j* is used for updating V_i by running task *i* based on function $\Delta()$. At the end of each iteration, all updated V_i are synchronized and partitioned to the next iteration. Note that, $D_{i,j}$ and $D_{i,k}$ refer to the same data and the difference is the data location.

2) Task processing time: The processing time of task *i* depends on the various parameters within V_i and the worker on which it runs. For running it on worker *j*, we define a function to estimate the time:

$$T_{i,Pro} = \sum_{v_i \in V_i, d_{i,j} \in D_{i,j}} f_j(v_i, d_{i,j}) \tag{4}$$

where v_i is the parameter within V_i and $d_{i,j}$ refers to the bound input within $D_{i,j}$. v_i and $d_{i,j}$ are connected by sharing the same hash key. $f_j(v_i, d_{i,j})$ represents the processing time of each parameter.

3) Parameter processing time: As illustrated in Section II, the processing time of each parameter depends on the size of the bound input. To estimate f_j, we first collect historical parameter processing results from the tasks running on the worker. To adapt to the dynamic heterogeneity, we use the information from tasks (one slave worker could run multiple tasks) in the previous iteration to estimate f_j in the current iteration. Thus, we collect a set of data points that relate the processing time of each parameter to the size of bound input as shown in Figure 1(a). For each worker, the number of collected data points equals the number of parameters partitioned to the worker.

We then use polynomial regression to construct a mathematical function that has the best fit for these points. The function can be presented as the fitting curves in Figure 1(a). After the regression, we obtain all coefficients to construct the function:

$$f_j(v_i, d_{i,j}) = a_j^1 * |d_{i,j}|^0 + a_j^2 * |d_{i,j}|^1 + ... + a_j^{n+1} * |d_{i,j}|^n \tag{5}$$

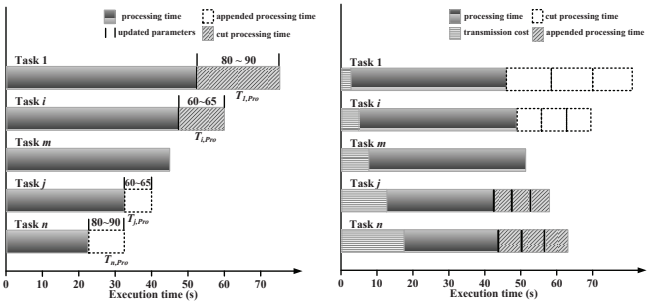
where a_j^1 to a_j^{n+1} are coefficients and *n* represents the highest order in the regression.

Note that a previous work [15] uses an exponential function with fixed index to construct the mathematical function. However, such a method cannot be adaptive to various ML jobs with different time complexities in f_j. In contrast, polynomial regression can effectively characterize linear, super-linear and $n\log n$ computations. Thus, we use polynomial regression to construct the mathematical function.

C. Parameter Partition

The partition is comprised of three components as follows. We first predict computational skewness based on the default hash partition. Afterwards, we mitigate the skewness through two steps: balance processing time via parameter reassignment to shift loads from slower tasks to faster ones, and consider network transmission cost and improve the partition via balancing execution time. The executed partition accounts for the input-parameter binding relationship so that it does not hamper the convergence of ML model.

1) Skewness prediction: With the capacity model, one can directly partition parameters to balance the processing time. However, such a method ignores the parameter-input binding relationship, leading to non-trivial network transmission cost. To reduce such cost, FlexPara partitions parameters on the basis of the default hash partition that causes no network transmission. We first calculate the parameter distribution under hash partition. We then use the capacity model to predict



(a) Skewness mitigation though parameter reassignment.

(b) Unbalanced execution time due to heterogeneous transmission costs.

Fig. 5. Example of Parameter Partition.

the processing times of the tasks running on parallel workers and examine the potential computational skewness. If the skewness exists under the hash partition, FlexPara performs parameter reassignment.

2) Skewness mitigation - parameter reassignment: We use a heuristic load balancing algorithm to reassign parameters from slower tasks to faster ones in order to mitigate the skewness. Algorithm 1 describes the process of parameter reassignment. Specifically, it calculates the difference of processing time between the fastest task and the slowest task. If the difference exceeds a threshold *p*, the skewness exists and the reassignment is initiated (lines 4 to 13). During the reassignment, tasks are ranked according to the processing time. Let $T_{m,Pro}$ be the median value. The tasks with processing time larger than $T_{m,Pro}$ are regarded as slower tasks. Faster tasks are defined in a similar way. Then, we pair a faster task *j* with a slower task *i* based on their distance to *m*. Afterwards, parameters from *i* are reassigned to *j*. Note that the number of reassigned parameters is in proportion to the time difference between tasks *i* and *j* in order to accelerate the skewness mitigation (lines 7 to 9). $|V_i|$ refers to the number of parameters in partitioned V_i. Finally, we predict processing time based on the new partition until the difference (*Diff*) is smaller than the threshold *p*.

Figure 5(a) shows an example with the predicted processing time for five tasks. After detecting faster tasks (i.e., *j* and *n*) and slower tasks (i.e., *i* and 1), FlexPara pairs task 1 with *n* and task *i* with *j*. It then reassigns parameters from task *1* to *n* and from *i* to *j*. Figure 5(a) shows the execution time ($T_{1,Pro}$ and $T_{i,Pro}$) that are cut from tasks *1* and *i*, and the execution time ($T_{j,Pro}$ and $T_{n,Pro}$) that tasks *j* and *n* get appended. Indeed, before the reassignment, task 1 spends $T_{1,Pro}$ in processing parameters from serial number 80 to 90. After reassignment, these parameters are processed by task *n* with execution time of $T_{n,Pro}$. The number of reassigned parameters from 1 to *n* is larger than that from *i* to *j*. Tasks 1 and *i* reassign parameters to tasks *n* and *j* separately during the next reassignment circle until the threshold *p* is met.

3) Skewness mitigation - balancing execution time: After reassignment, the uneven partitioned parameters result in heterogeneous $T_{i,Net^{pa}}$ as well as extra input data movement $T_{i,Net^{in}}$. For example, when the five tasks in Figure 5(a) are located on different workers, Figure 5(b) shows the execution time after the reassignment. Although the processing times are balanced, the network transmission cost of tasks with more shuffled parameters (i.e., tasks *j* and *n*) is more than that of others, leading to unbalanced execution time. To evaluate the cost, FlexPara estimates the network transmission rate of each worker. For worker *j*, the rate is defined as:

$$R_{j,Net} = \sum S_k / \sum (T_{k,Net^{pa}} + T_{k,Net^{in}}) \qquad (6)$$

where *k* refers to task *k* that previously runs on this worker. S_k is the size of transmissive data including the shuffled parameters and transmitted input data. Both numerator and denominator are collected from the previous iteration to estimate $R_{j,Net}$ in the current iteration.

In Section III-C2, Algorithm 1 only predicts the processing time at the end of each reassignment (line 15). To balance the execution time, FlexPara replaces it with execution time prediction. Algorithm 2 describes the procedure.

Algorithm 1 Parameter reassignment

1: **Variable:** Skewness threshold *p*, reassignment unit *b*;
2: Rank tasks based on processing times;
3: *Diff* = difference of the time between the fastest task and the slowest one;
4: **while** *Diff* is larger than *p*
5: set $T_{m,Pro}$ as the median processing time of tasks;
6: **for** task *i* **in** tasks with larger time than $T_{m,Pro}$
7: *d* = the distance between task *i* and *m* in the rank;
8: Obtain task *j* with same distance but smaller time;
9: Calculate the number of reassigned parameters $|V_r| = d * b$;
10: Parameter reassignment from task *i* to task *j*:
11: $|V_i| = |V_i| - |V_r|$; $|V_j| = |V_j| + |V_r|$;
12: **end for**
13: Re-predict processing times under current partition;
14: Re-rank tasks; Re-calculate the *Diff*;
15: **do while**

Algorithm 2 Execution time prediction of task *i*

1: Get shuffled parameters V_i;
2: Estimate the rate $R_{j,Net}$ of the worker *j* running task *i*;
3: Obtain the binding input $D_{i,*}$ based on V_i and shared keys;
4: Get local input data $D_{i,j}$ on worker *k*;
5: Calculate the moved input $D_{i,move} = D_{i,*} - D_{i,j}$;
6: Transmissive data size S_i = size of V_i and $D_{i,move}$;
7: Predict $T_{i,Pro}$ based on capacity model and V_i;
8: Predict transmission cost $T_{i,Net^{pa}} + T_{i,Net^{in}} = S_i / R_{k,Net}$;
9: Execution time $T_{i,Exe} = T_{i,Pro} + T_{i,Net^{pa}} + T_{i,Net^{in}}$;
10: Return $T_{i,Exe}$;

IV. Implementation

We implement FlexPara in Spark version 1.6.3 by modifying `spark.core` code and `spark.mllib` library. The driver in Spark collects capacity models from parallel workers and conducts the parameter partition.

A. Capacity Model

To store the capacity model, we add an object `executor.coefficient` in each worker. The object collects the data points during task execution in Spark executor by rewriting function `rdd.iterator()` in `ShuffleMapTask.scala` file. The information of shuffled parameters comes from the class `shuffleReadMetrics` in `TaskMetrics.scala` file and `MLlib` library. We use *Flanagan's Java Scientific Library* [19] to calculate the coefficients in polynomial regression.

B. Parameter Partition

Skewness prediction: Both the parameter and the input are stored in key-value form, where the key is the *id* (i.e., hash key) and the value is the parameter or the bound input. We add an object `executor.localid` in each worker to collect *id* of the stored input and send them to the driver. The driver connects *id* with the keys in `executor.globalvalues` that stores the information of global parameters to obtain the hash partition.

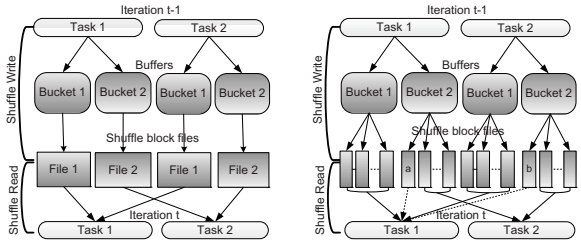
Parameter reassignment: Figure 6(a) shows the implementation of the default hash partition that consists of two phases. In the shuffle write phase, the task from the previous iteration first partitions parameters based on their hash values and then stores them as buckets in the buffer space. Each task has R buckets and R equals to the number of tasks in the next iteration. Afterward, parameters in these buckets are written continuously to the local disk and form files called ShuffleBlockFile. Each bucket corresponds to one file. In the shuffle read phase, tasks in the next iteration fetch these files.

Parameter reassignment in FlexPara reassigns a subset of parameters from one task to another. However, the default implementation does not support such reassignment. The reason is that for each task t in the next iteration, a previous task only stores the parameters partitioned to t in one file. For example, in Figure 6(a), if the parameters partitioned to task 2 in iteration t are reassigned to task 1, task 1 has to fetch two File-2 files from iteration t-1. In other words, the entire set of parameters in task 2 are reassigned to task 1.

Figure 6(b) demonstrates our implementation. We change the storage of ShuffleBlockFile in the shuffle write phase so that each bucket can be written to multiple local disk files. Note that, the number of these files equals to the reciprocal of the reassignment unit in Algorithm 1. For example, when the reassignment unit is set to 5%, each bucket is be written to 20 files and one reassignment reassigns at least 5% parameters from a fast task to a slow one. To support multi-file shuffle read, we establish more TCP communication channels to transfer the files. Thus, in the shuffle read phase, the reassignment is realized by reading a subset of files corresponding to one bucket according to Algorithm 1. For example, reassigning a subset of parameters from task 2 to task 1 can be implemented by additionally transferring files a and b to task 1.

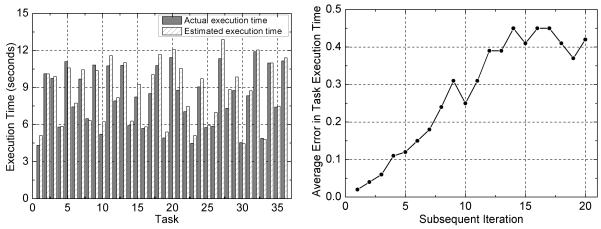
Parameters in each task are stored in variable `records` in `ShuffleMapTask.scala`. Class `SortShuffleWriter` writes the parameters to the local disk. To write one bucket to multiple files, we revised the function `insertAll()` in the class `ExternalSorter` used by `SortShuffleWriter` and we modified the *buffer* related class `PartitionedPairBuffer` in `spark.util.collection`.

To obtain the network transmission rate, we used Linux command `netstat` to monitor the network status of each worker and implemented a distributed monitoring tool based on `RPC`. The tool estimates the transmission rate at the end of each iteration.



(a) Default implementation. (b) Reassignment implementation.

Fig. 6. Partition implementation comparison.



(a) Task execution time. (b) Dynamic skewness.

Fig. 7. (a) the execution time in the physical cluster, (b) the average error of execution time in subsequent iterations compared to the reference.

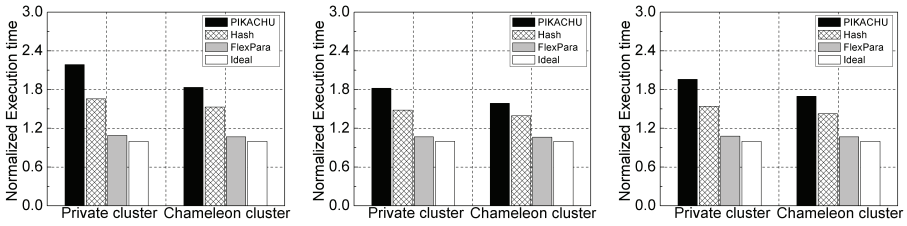
V. Evaluation Setup

A. Testbeds

We built two multi-tenant Spark clusters to evaluate the performance of FlexPara. A private cluster is deployed in a university private cloud with 37 virtual machines (VMs), i.e., one master node and 36 worker nodes. Each node is assigned with one virtual CPU core. A Chameleon cluster is deployed in NSF Chameleon OpenStack KVM cloud [20]. The cluster contains one master node and 9 worker nodes (Chameleon allows total 10 nodes). Each node is assigned with four virtual CPU cores. The two clusters are configured with 10Gbps network bandwidth. Note that the private cloud is shared by all faculty, staffs, and students and Chameleon OpenStack KVM cloud is shared by different universities and institutions.

B. Workloads

To estimate the effectiveness of FlexPara, we choose three representative ML jobs. **PageRank** uses the dataset from HiBench, containing 10k *pages* (k = 1000) for the private cluster and 5k *pages* for Chameleon cluster. The number of parameters equals to the number of pages. **Matrix Factorization** uses alternating least squares approach for model training. The experiments use the dataset from SparkBench [21], containing a 20k*3k matrix and 23k parameters for the private cluster, and a 10k*3k matrix and 13k parameters for Chameleon cluster. The *rank* is configured as 300 for the two clusters. **Topic Modeling** with LDA uses the dataset from Nytimes corpus [22], containing 40k articles with 256 topics for the private cluster, and 20k articles with 256 topics for Chameleon cluster. The number of parameters equals to the number of articles. The data size in the Chameleon cluster is smaller since only 10 VMs can be applied. We run these jobs until they reach default convergence thresholds in Spark.



(a) PageRank. (b) Matrix Factorization. (c) Topic Modeling.

Fig. 8. Execution time comparison among three approaches in private and Chameleon clusters.

C. Compared Approaches and Metrics

The ideal execution environment for ML jobs is a homogeneous cluster and input data are pre-partitioned to achieve homogeneous loads on parallel tasks. Hence, we run the jobs in such an environment and consider it as the *baseline*.

We evaluate the performance of three partition approaches: *FlexPara* (our scheme), *default hash partition* in Spark, and *PIKACHU* (recent proactive partition approach). In default hash partition, parameters with the same hash key are partitioned to the same task (parameter is stored in key-value form). PIKACHU estimates machine capacity based on the linear relationship between map task execution time and task input size. It then partitions intermediate data to reduce stage [12]. Note that PIKACHU was implemented in MapReduce Hadoop environment. In our evaluation, we implement PIKACHU in Spark environment so as to compare it with FlexPara on the same computing framework.

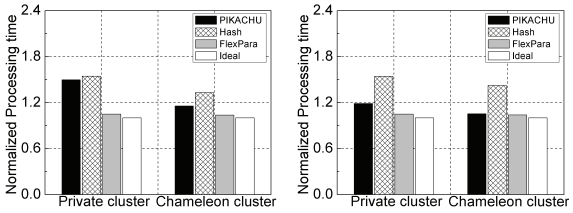
The performance measurements include execution time, processing time, and network transmission cost spent on one iteration on average. The measurements are normalized to the baseline performance.

VI. Experimental Evaluation

FlexPara features several run-time hyperparameters. In the default settings, the highest order n in Equation 5 is set to be 9 in the two clusters. The threshold p and the reassignment unit b in Algorithm 1 are set to be 9% and 5% in the private cluster. These two values are set to be 6% and 8% in Chameleon cluster. Note that these default values are the best settings empirically obtained by experiments in Section VI-D.

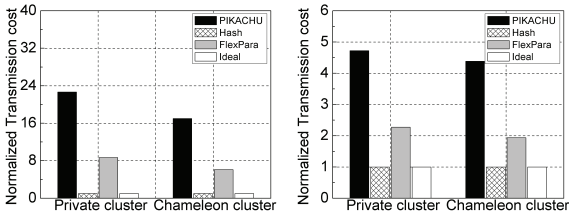
A. Dynamic Heterogeneity and Capacity Model Accuracy

To illustrate the computational skewness in multi-tenant clusters, we run PageRank job with hash partition in the private cluster. We present the actual task execution time in a randomly selected iteration in Figure 7(a). The result shows that the fastest task can be 1.5x faster than the slowest task in the private cluster. Figure 7(a) also shows the estimated execution time using the capacity model. The results show that the capacity model achieves high accuracy in execution time estimation, with an average error of 7.5% (average error in work [15] is 7.05%). Note the average error is 6.95% in Chameleon cluster. To verify the dynamics of heterogeneity, we run PageRank job with the hash partition in the private cluster. We randomly select one iteration as the reference. We measure the average error of the time in the subsequent iterations and compare to the reference as shown in Figure 7(b).



(a) PageRank. (b) Topic Modeling.

Fig. 9. Processing time comparison among three approaches.



(a) PageRank. (b) Topic Modeling.

Fig. 10. Transmission cost comparison among three approaches.

The cluster exhibits different skewness in the subsequent iterations. We observed similar results, though omitted due to space, for Chameleon cluster and other three jobs.

B. Execution Time Evaluation

Figure 8 shows the result of the execution time. Among the three approaches, FlexPara is the closest to the ideal and it achieves significant performance gain over the other two approaches. Compared to hash partition, FlexPara achieves significant performance improvements, with 54%, 39% and 43% faster execution time the three ML jobs respectively, in the private cluster. The experiment in Chameleon cluster shows similar results with 43%, 32% and 33% faster execution time respectively, for the three jobs. The private cluster has better performance improvement since the skewness in the private cluster is more severe than that in Chameleon cluster.

Compared to hash partition, PIKACHU results in obviously worse performance in execution time for each of the three ML jobs. The performance degradation mainly comes from a large amount of network transmission cost. Moreover, for iterative ML jobs, PIKACHU partitions parameters based on the inaccurate linearity assumption, leading to additional computational skewness. Overall, FlexPara achieves significant improvements over PIKACHU, by 1x, 70% and 82% faster execution time for the three ML jobs in the private cluster, and 72%, 47% and 59% faster execution time for the three ML jobs in Chameleon cluster.

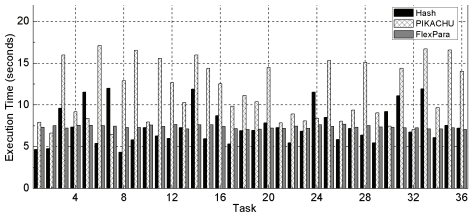


Fig. 11. Task execution time of PageRank job in the private cluster.

Figures 9 and 10 show the processing time and the network transmission cost of the three approaches in PageRank and Topic Modeling jobs. Figure 9 shows that processing time of both FlexPara and PIKACHU are smaller than that of hash partition, and FlexPara is closest to the ideal. FlexPara outperforms hash partition due to the parameter reassignment. Although PIKACHU models the capacity based on inaccurate linearity assumption, it can still mitigate the skewness to a certain extent. PIKACHU in Chameleon cluster performs better than that in the private cluster since the larger cluster increases its error in modeling capacity.

In Figure 10, the hash partition is close to the ideal in transmission cost, because they both incur little input data movement and the cost only comes from the parameter shuffling. The cost of FlexPara and PIKACHU is larger than that of the hash partition due to the extra input data movement. Specifically, PIKACHU results in the largest cost since it is not aware of the binding relationship. FlexPara, in contrast, reduces such cost through the parameter reassignment dependent on hash partition. Also, compared to PageRank, the cost of Topic modeling in hash partition is larger since it shuffles larger size of parameters.

Figure 11 plots the task execution time in a randomly selected iteration of PageRank job in the private cluster. Results show that the hash partition causes severe computational skewness. PIKACHU worsens the skewness due to its resulted network transmission cost. FlexPara can balance the task execution time. Figure 12 further plots the parameter reassignment procedure between tasks 7 and 8 before the iteration begins. The x-axis represents the reassignment times (i.e., *while* loop times in Algorithm 1). The left y-axis refers to the estimated execution time of two tasks after each reassignment. Before reassignment begins (reassignment times = 0), parameters size in each task is the same as that in hash partition. After each reassignment, estimated execution time in task 8 gets closer to the time in task 7 since more parameters are partitioned to task 8. The network transmission cost in task 7 is negligible since only parameters are shuffled to the task. In contrast, after first reassignment, the transmission cost in task 8 becomes non-negligible due to input data movement. The cost increases as more parameters are reassigned from task 7 to task 8.

TABLE II
OVERHEAD OF FLEXPARA IN THE PRIVATE CLUSTER.

Workload	regression	algorithms	storage	transmission	total
PageRank	0.83%	1.2%	2.5%	1.7%	6.23%
Matrix Factorization	0.79%	2.2%	2.9%	2.1%	7.99%
Topic Modeling	0.81%	1.8%	2.3%	1.6%	6.51%

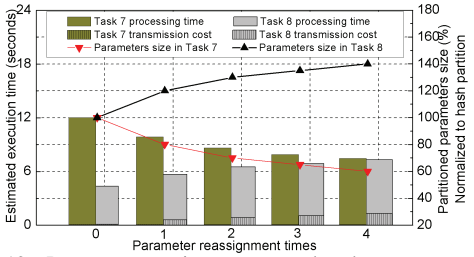


Fig. 12. Parameter reassignment procedure between two tasks.

C. Overhead

FlexPara has four overheads: the CPU overhead in polynomial regression, the CPU overhead in the algorithms, the I/O overhead in multiple parameters files storage, and the I/O overhead in multiple parameters files transmission. Table II lists their percentages in task execution time with default hyperparameters in the private cluster. The sum of four overheads is much smaller than the performance improvement. Similar results, omitted in this paper, hold for Chameleon cluster.

D. Hyperparameter Evaluation

We vary each hyperparameter and show the results of PageRank in the private cluster. Similar results hold for the other three jobs and also for the Chameleon cluster.

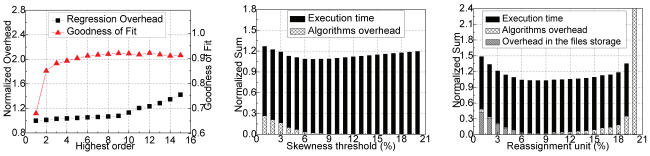
The highest order. Figure 13(a) shows the goodness of Fit and the overhead of polynomial regression when the highest order n varies from 1 to 15. For the Goodness of Fit, the smallest value occurs when the order is set to one. Once the order is set to 2 or higher, it becomes much better and remains on a slow-growth path. For the overhead (normalized to the overhead when the order is 1), it is increasing rapidly when the order is larger than 9. Thus, to maintain a good balance between performance and the overhead (high Goodness of Fit and low overhead), we choose 9 as the default highest order.

The threshold. Figure 13(b) presents task execution time and the algorithms overhead (both of them are normalized to the execution time when threshold is 1%) when the threshold p varies from 1% to 20%. The sum of the two achieves the lowest value when the threshold is set between 6% and 9%. A lower threshold brings smaller execution time but higher overhead, and vice versa.

The reassignment unit. Figure 13(c) plots task execution time, the algorithms overhead, and the overhead in multiple parameters files storage when the reassignment unit b varies from 1% to 20%. Once the unit is set between 5% and 8%, FlexPara performs the lowest sum. For a smaller unit, it spends longer time to calculate the reassignment in Algorithm 1. It also needs more files to store parameters and more TCP channels to shuffle parameters. A higher unit has lower overhead in the files storage, but it causes redundant reassignments.

VII. RELATED WORK

Speculative execution and task cloning: Speculative execution is used to mitigate skewness in data processing frameworks like MapReduce and Spark [8], [9], [10]. The concept is to run slow tasks redundantly on multiple machines.



(a) The highest order. (b) The threshold. (c) The reassignment unit.

Fig. 13. Hyperparameter evaluation of PageRank in the private cluster.

However, for ML jobs with binding parameter and input data, speculative execution on other machines results in extra input data movement, leading to certain network transmission cost.

Work stealing, work shedding: The concept of work stealing or work shedding is to move task loads from a busy worker to an idle worker. Previous approach [11] waits for a worker to idle before looking to steal work, incurring additional delays until work is found. Recent effort FlexRR [14] identifies slow workers and reassigns the load before fast workers finished. However, these stealing approaches estimate the stolen task load based linear assumption. In contrast, FlexPara proactively steals (reassigns) parameters without linear assumption.

Adaptive task size adjustment: Task size adjustment solves the performance skewness by matching the amount of data processed at heterogeneous machines to their respective capabilities. FlexMap [13] proposes a new map execution engine for MapReduce to create elastic map tasks with different input block sizes. PIKACHU [12] proposes new intermediate data partition algorithms to adaptively adjust the reduce task size for MapReduce jobs. Recent work [15] proposes a new task progress indicator for MapReduce jobs. But it does not use the indicator to mitigate computational skewness.

Less strict synchronization: Synchronization in Spark follows the strict BSP model and thus their performances can be impaired by slower tasks. Recent efforts propose alternative synchronization models to mitigate the skewness. A-BSP [23] is a BSP-based aggressive synchronization model that uses updates from partial input data for synchronization. SSP [3], [24] uses flexible synchronization and allows any worker to be up to a bounded number of iterations ahead of the slowest worker. SSP-based frameworks (e.g., Petuum [3]) mitigate the communication overhead problem with parameter server architecture [24], [25], [26].

VIII. Conclusion

In this paper, we tackle the difficult skewness problem in distributed iterative ML jobs in multi-tenant clusters. We propose FlexPara, a parameter partition approach that provisions adaptive tasks to match the distinct machine capacity. FlexPara first models machine capacity in task processing for iterative ML jobs without linear assumption. Then, it leverages the nature of iterative jobs to adaptively partition parameters in each iteration through proactive parameter reassignment in order to adjust task size. We have implemented FlexPara in Spark and performed evaluations with various ML jobs. Experimental results show that the execution time with FlexPara is faster than that with hash partition by up to 54% and 43% in private and Chameleon clusters, respectively.

IX. Acknowledgment

This research was supported in part by U.S. NSF grants CNS-1422119 and SHF-1816850. Results presented in this paper were obtained partially using the Chameleon testbed supported by NSF.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *Proc. of USENIX HotCloud*, 2010.

[2] 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, pp. 1235–1241, 2016.

[3] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson *et al.*, "Exploiting iterativeness for parallel ml computations," in *Proc. of ACM SoCC*, 2014.

[4] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *Proc. of IEEE INFOCOM*, 2018.

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proc. of USENIX ATC*, 2014.

[6] S. Wang, W. Chen, X. Zhou, L. Zhang, and Y. Wang, "Dependency-aware network adaptive scheduling of data-intensive parallel jobs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[7] C. Chen, W. Wang, and B. Li, "Performance-aware fair scheduling: Exploiting demand elasticity of data analytics jobs," in *Proc. of IEEE INFOCOM*, 2018.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.

[9] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *Proc. of OSDI*, 2008.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *Proc. of USENIX NSDI*, 2013.

[11] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proc. of ACM SIGMOD*, 2012.

[12] R. Gandhi, D. Xie, and Y. C. Hu, "Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters." in *Proc. of USENIX ATC*, 2013.

[13] W. Chen, J. Rao, and X. Zhou, "Addressing performance heterogeneity in mapreduce clusters with elastic tasks." in *Proc. of IEEE IPDPS*, 2017.

[14] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml," in *Proc. of ACM SoCC*, 2016.

[15] E. Coppa and I. Finocchi, "On data skewness, stragglers, and mapreduce progress indicators," in *Proc. of ACM SoCC*, 2015.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[17] "Wikipedia corpus," https://corpus.byu.edu/wiki/.

[18] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks." in *Proc. of USENIX NSDI*, 2015.

[19] M. T. Flanagan, "Michael thomas flanagans java scientific library," 2007.

[20] "Chameleon cloud," https://www.chameleoncloud.org/.

[21] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. of ACM CF*, 2015.

[22] "New york times corpus," https://catalog.ldc.upenn.edu/LDC2008T19.

[23] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proc. of ACM/USENIX/IFIP Middleware*, 2018.

[24] 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 *Proc. of OSDI*, 2014.

[25] Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, Y. Wan, Z. Li, J. Wang, S. Huang *et al.*, "Bigdl: A distributed deep learning framework for big data," *arXiv preprint arXiv:1804.05839*, 2018.

[26] S. Wang, A. Pi, and X. Zhou, "Scalable distributed dl training: Batching communication and computation," in *Proc. of AAAI*, 2019.