Fela: Incorporating Flexible Parallelism and Elastic Tuning to Accelerate Large-Scale DML

Jinkun Geng, Dan Li and Shuai Wang
Department of Computer Science and Technology, Tsinghua University
Beijing, China 100084
steam1994@163.com, tolidan@tsinghua.edu.cn, s-wang17@tsinghua.edu.cn

***Abstract*—Distributed machine learning (DML) has become the common practice in industry, because of the explosive volume of training data and the growing complexity of training model. Traditional DML follows data parallelism but causes significant communication cost, due to the huge amount of parameter transmission. The recently emerging model-parallel solutions can reduce the communication workload, but leads to load imbalance and serious straggler problems. More importantly, the existing solutions, either data-parallel or model-parallel, ignore the nature of flexible parallelism for most DML tasks, thus failing to fully exploit the GPU computation power. Targeting at these existing drawbacks, we propose `Fela`, which incorporates both flexible parallelism and elastic tuning mechanism to accelerate DML. In order to fully leverage GPU power and reduce communication cost, `Fela` adopts hybrid parallelism and uses flexible parallel degrees to train different parts of the model. Meanwhile, `Fela` designs token-based scheduling policy to elastically tune the workload among different workers, thus mitigating the straggler effect and achieve better load balance. Our comparative experiments show that `Fela` can significantly improve the training throughput and outperforms the three main baselines (i.e. data-parallel, model-parallel, and hybrid-parallel) by up to $3.23\times$, $12.22\times$, and $1.85\times$ respectively.**

***Index Terms*—Distributed machine learning, flexible parallelism, token-based scheduling, straggler effect**

I. Introduction

Machine learning, especially deep learning, nowadays is confronted with large data sets and high dimensional models, which have gone beyond the storage and computation capability of a single machine. Facing such a situation, distributed machine learning (DML) becomes the common practice to handle those consuming training tasks. At a high level, DML solutions can be executed under data parallelism, model parallelism, or even hybrid parallelism.

Data parallelism requires each machine (i.e. worker) to hold a complete model instance, and the workers use different portions of training data to update the model parameters. At the end of each iteration, the workers synchronize their parameters and continue the next iteration. Despite its wide support [1]–[3], data-parallel solutions cannot execute DML efficiently in large scale, due to the huge amount of network transfer [4]–[7] and the common straggler problem [8]–[10]. Although the computation and communication capability have both been improved in the past decade, DML is developing towards the network-bounded direction [6], [7], [18]. A couple of factors combine to intensify the communication overheads for today's DML tasks, including the increasing model size and popularity of large-batch training (refer to Section II-A for more details). Besides, data-parallel solutions also suffer from serious straggler effect [9], [10]. Straggler effect is a common problem for DML tasks running under Bulk Synchronous Parallel (BSP) mode, which means all workers have to wait for the slowest one (i.e. straggler) in each iteration. To tolerate stragglers, Asynchronous Parallel (ASP) [1] and Stale Synchronous Parallel (SSP) [10] sacrifice iteration quality for speed, which is undesirable in most practical cases. Other solutions, such as FlexRR [11], mitigate straggler effect by offloading the workload to other workers, which can be very costly, considering the large volume of training data.

As an alternative to data-parallel ones, model-parallel solutions [12]–[15], as well as those hybrid-parallel ones incorporated with data parallelism [6], [16], are proposed in recent years to accelerate large-scale DML. Rather than maintaining a complete model instance on each worker, model-parallel solutions partition one model into different sub-models and each worker only trains one of them. Compared with data-parallel ones, model-parallel solutions allow each node to only transfer the boundary parameters between neighbor nodes, thus reducing the communication cost. However, model-parallel solutions also suffer from distinct drawbacks. First, the typical model-parallel solutions [12]–[15] follow the pipeline-based workflow, which means the input of some workers depends on the output of the other workers. During each iteration, the workers located in the backend need to wait until the front workers complete training and transfer the required parameters, thus causing the *bubble effect* [13] (i.e. bad work conservation) and wasting computation resource. Second, model partition can be hardly balanced [12], [14], and the straggler problem also exists in model-parallel solutions. The recent work, ElasticPipe [15], integrates auto-tuning mechanism to migrate the workload in the runtime. ElasticPipe can mitigate the straggler problem to some extent, but it relies on the head node to make new partitions periodically, with reference to the the profiling information collected in previous iterations. Therefore, it cannot timely respond to the real-time situation. When the straggler effect is switching rapidly among workers, the auto-tuning mechanism can even worsen the straggler problem, because it may migrate the workload from workers which have recovered from straggler effect, but

add the workload to those workers which are suffering from straggler effect.

Apart from the straggler problem, we also observe the necessity of flexible parallelism while training large and complex models. More specifically, the computation of different parts of the model leads to different utilization of GPU resource. In order to saturate the GPU power, it needs different batch sizes to train different parts of the model. Unfortunately, most existing works only consider the fixed batch size throughout the whole training process. FlexPS [17] considers flexible parallelism for different stages (i.e. a couple of iterations). However, it fails to consider the parallelism difference among different sub-models. Besides, since it mainly targets at data-parallel solutions under PS-based architecture, FlexPS inherits the drawbacks of data-parallel solutions and PS-based solutions, and it also suffers from the straggler problem, as well as centralized network bottleneck.

Noting that none of the existing works have fully considered the flexible parallelism and straggler mitigation in DML, we propose `Fela` to cover both dimensions. `Fela` adopts flexible parallelism to train different sub-models, so as to better utilize the GPU power. Meanwhile, `Fela` designs a *token-base scheduling mechanism* to elastically tune the workload for different workers, and further mitigate the straggler problem more efficiently and more timely. We conduct comparative experiments on a 8-node cluster and evaluate the training throughput for equal number of iterations, which shows that `Fela` outperforms the baselines by up to $3.23\times$,$12.22\times$ and $1.85\times$, respectively.

The remaining part of the paper is organized as follows. Section II introduces the background and motivation of our work. Section III and Section IV describe the details of design and implementation. Section V proves the outperformance of `Fela` with testbed experiments. Section VI includes some related work and Section VII concludes the paper.

II. Background and Motivation

A. Emerging Trends of DML

Reviewing the current situation of DML tasks and relevant hardware development, we notice that DML is growing network-intensive. Although DML is known as computation-hungry, the recent booming development of computation hardware (e.g. GPU, TPU, ASIC, etc) has mitigated the computation bottleneck to a large extent. Looking back in the past 5 years, the computation power has been increased by $35\times$ [7], [18]. By contrast, the communication capability, though has also made some progress (e.g. RDMA [19]), cannot match the development speed of computation power. 10 Gbps network still remains as the mainstream in public cloud and some private DML clusters [3], [6], [20] and 10$\sim$25 Gbps will continue to dominate the market in the near future [1]. The newly invented RDMA technology can effectively improve the network performance. But RDMA has its internal drawbacks (e.g. PFC Pause Frame Storm [5], [19], [21]) that constrain its scalability. Meanwhile, the upgrading cost is also a major concern for network operators to replace Ethernet NICs with RDMA NICs in large data center. Therefore, it is reasonable to believe that the network capability will fall behind the computation power in the near future, and such a mismatch will intensify the communication bottleneck for DML tasks.

Besides, large-batch training is becoming more popular in recent DML practice, in order to tackle the increasing volume of training data [22]–[25]. More than the batch size, DML models, especially the deep neural network models, are also growing wider and deeper for the sake of stronger learning capability [26]. Starting from the simple LeNet with only 5 layers, the typical neural network models have reached hundreds of, or even thousands of layers (refer to Table I). On the other hand, however, GPU memory is scarce and expensive in DML clusters, and one node cannot hold a large model with a large batch size. Therefore, the explosive growth of batch size and model size necessitate the involvement of lots of nodes in jointly training, which further increases the communication workload (especially for data-parallel solutions).

TABLE I: Growing Neural Network Layer Numbers

Model	Year	Layer Number
LeNet-5	1998	5
AlexNet	2012	8
ZF Net	2013	8
VGG16	2014	16
VGG19	2014	19
GoogleNet	2014	22
ResNet-152	2015	152
CUImage	2016	1207
SENet	2017	154

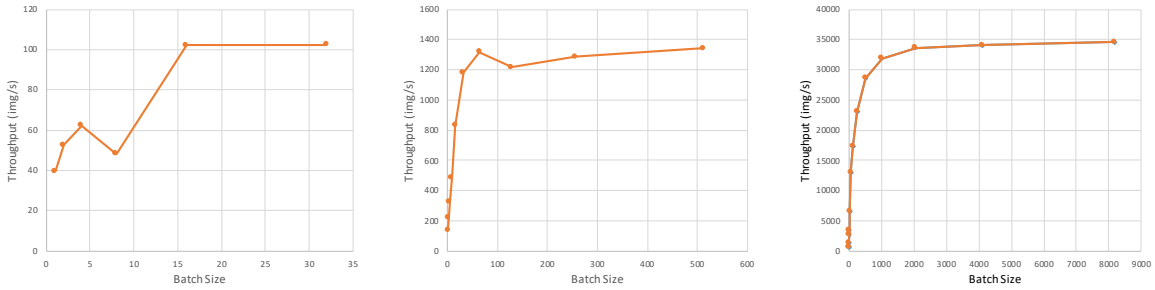
B. Flexible Parallelism in DML

Contrary to the stringent demand of GPU memory, we observe that the GPU computation power can hardly be saturated while training DML tasks. The reasons can be attributed to two main aspects. One aspect is due to the communication overhead of parameter synchronization, which causes idle time of GPU between iterations. The other aspect is because of the parallelism degree heterogeneity in training different model parts. Take VGG19 model as an example, the network layer becomes smaller when it goes deeper, and the optimal batch size for different layers also varies. To demonstrate this, we conduct a comparative experiment on one NVIDIA Tesla K40c 12 GB GPU. We choose different layers in VGG19 model and separately train the model layer with different batches, in order to find the optimal batch to saturate the GPU. The measurement results can be illustrated as Figure 1.

As shown in Figure 1, in order to saturate the GPU (i.e. to reach the maximum throughput), the required batch size is different for the layers. For the CONV layer located in the front of VGG19 (i.e. (64,64,224,224) [2]), the batch size of 16 has enabled it to saturate the GPU and reach the maximum throughput. By contrast, for the CONV layer located in the

[1]According to the recent survey from Mellanox [20], 10 Gbps NICs have occupied 69% of the total market in 2018, and 10$\sim$25 Gbps NICs are predicted to occupy 58% of the total market in 2022.

[2](64,64,224,224) describes the network layer in (C_{in}, C_{out}, H, W) format, where C_{in} is the number of input channels, C_{out} is the number of output channels, H is the height and W is the width of the network layer.



(a) Training CONV layer (64,64,224,224) (b) Training CONV layer (512,512,14,14) (c) Training FC layer (4096, 4096)

Fig. 1: Training throughput with different batch sizes

back of VGG19 (i.e. (512,512,14,14)), it requires the batch size of 64 to reach the maximum throughput. In other words, *the backend CONV layer needs higher parallelism degree (i.e. a larger batch size) than the front CONV layer*. What's more, the FC layer even requires much larger batch size (i.e. 2048) to saturate the GPU.

However, under data parallelism, single node can not hold a large batch due to the GPU memory constraint [3] , thus GPU computation power cannot be well utilized. Even with model parallelism, the fixed batch size also leads to serious waste of GPU memory and computation power. When the batch size is fixed for different sub-models, large batch size causes unnecessary GPU memory consumption for workers undertaking sub-models with low parallelism degree (i.e. the sub-model does not need large batch size to saturate GPU), whereas small batch size causes low GPU utilization for workers undertaking sub-models with high parallelism degree. More than deep neural networks, the heterogeneity of parallelism degree is also very common for other DML tasks, such as matrix factorization and PageRank, which incur distinct heterogeneity while workers are training different parts [17], [27], [28] and requires different parallelism degrees to better utilize the computation resource. Therefore, it can be concluded that flexible parallelism, rather than fixed batch size, is necessary to improve the resource efficiency and accelerate DML tasks.

C. Straggler Problem in DML

Straggler problem is a common concern in large-scale DML. While executing a DML task with many workers, the progress of workers varies during each iteration, due to the heterogeneity of computation/communication performance [8]–[11], as well as workload imbalance [15], [27], [28]. Under BSP mode, all the workers have to wait for the straggler before starting the new iteration. Therefore, the GPU resource of faster workers is wasted and the overall training time is prolonged.

To address the straggler problem, there are two lines of works proposed. One line of works change the original DML algorithm to tolerate stragglers. ASP and SSP slack the synchronization condition and allow faster workers to use stale parameters to continue next iteration. However, ASP spoils the iteration quality and may cause convergence failure [1], [29]. SSP makes some trade-off between iteration speed and iteration quality, but introduces staleness bound, which needs elaborate tuning and can worsen the convergence performance in some cases [10]. Besides, such algorithmic changes have damaged the reproducibility of DML tasks, and are not preferred in many cases. The other line of works migrate the workload to reduce or even avoid straggler effect. When some workers are suffering from a heavy workload, the scheduler will move some workload to other faster workers. Such works are also confronted with two main challenges. First, the migration cost is expensive, especially for data-parallel solutions, because it requires transferring a large volume of high-dimensional training samples among workers. Second, the delayed scheduling can even exacerbate the load imbalance and worsen straggler effect. The typical works, such as FlexRR [11] and ElasticPipe [15], need centralized schedulers to periodically detect the performance of workers and re-distribute the workload. However, when the straggler effect is time-varying and switches rapidly (i.e. *transient* stragglers [10]), the delayed workload migration may add more burden to the stragglers and further extend the training time.

D. Existing Drawbacks and Our Motivation

We compare the representative works in five dimensions and summarize the comparison results in Table II, from which we can see that none of the existing works have covered the five dimensions to accelerate large-scale DML. Data-parallel solutions, including FlexRR and FlexPS, suffer from serious communication cost due to huge amount of data transfer, as well as the centralized PS bottleneck. Model-parallel and hybrid-parallel solutions, including PipeDream, ElasticPipe and Stanza, suffer from bad work conservation because the input of some workers have to depend on the output of others, and remain idle at the beginning of every iteration [4]. The straggler-mitigated solutions, including LazyTable and FlexRR, either sacrifice iteration quality and damage algorithm reproducibility (e.g. LazyTable), or migrate the workload at an expensive cost and may fail to response to straggler effect timely (e.g. FlexRR). More importantly, most existing works, except FlexPS, ignore the the necessity of flexible parallelism in DML. FlexPS considers the difference of parallel degrees for training the whole model in different stages (i.e. a couple of

[3]Specifically, while training a complete VGG19 model with PyTorch on Tesla K40c GPU, the batch size larger than 32 has exceeded the GPU memory.

[4]PipeDream [12] improves the workload conservation by using SSP, however, as aforementioned, SSP sacrifices iteration quality and damages algorithm reproducibility.

TABLE II: Comparison of Representative DML Solutions

Solution	Parallel Mode	Flexible Parallelism	Straggler Mitigation	Communication Efficiency	Work Conservation	Algorithm Reproducibility
LazyTable [10]	Model-Parallel	×	√	√	√	×
FlexRR [11]	Data-Parallel	×	√	×[1]	√	×
FlexPS [17]	Data-Parallel	√	×	×[2]	√	√
PipeDream [12]	Model-Parallel	×	×	√	×	×
ElasticPipe [15]	Model-Parallel	×	√	√	×	√
Stanza [6]	Hybrid-Parallel	×	√	√	×	√
Fela	Hybrid-Parallel	√	√	√	√	√

[1] The communication inefficiency is due to the expensive migration cost for straggler mitigation.
[2] The communication inefficiency is due to the centralized bottleneck at PS and the huge network transfer of data-parallel solutions.

iterations), but fails to consider training different sub-models in one iteration.

Targeting at the drawbacks of existing DML solutions, we propose Fela, which fully considers flexible parallelism and elastic tuning in DML. Fela adopts hybrid parallelism to better utilize the computation and communication resource in the DML cluster. More than that, Fela makes three main contributions, which can be summarized as follows.

(1) Fela considers the flexible parallelism in DML. While training different sub-models, Fela chooses different batch sizes to feed to the sub-model, so as to saturate the GPU power and accelerate training DML tasks.

(2) Fela designs a role of Token Server(s) and leverages token-based scheduling to *reactively* mitigate straggler problems. Tokens represent workload. When workers are free, they request more tokens by themselves, rather than wait for the periodically polling and scheduling from the schedulers.

(3) Fela integrates elaborate policies (refer to Section III-D∼III-E) in its token-based scheduling, which can effectively reduce the communication cost and speedup the training process of DML.

III. Design of Fela

A. Architecture and Workflow

The architecture of Fela is shown as Figure 2. There are two roles designed, namely, Token Server (TS) and worker.

We design TS to manage the progress of workers. Note that although TS is logically separated from workers, it can be co-located with workers, because TS is not computation-intensive and CPU resource on the server can suffice its demand. Different from the well-known Parameter Servers (PS), TS does not hold the massive model parameters, instead, it only produces tokens for workers to consume. The network workload is at most hundreds of bytes during each transfer, which causes no centralized bottleneck.

TS Logic. On the TS side, there are four main components, namely, Token Generator, Token Distributor, Token Bucket and Info Mapping. Token Bucket stores the tokens generated by Token Generator whereas Info Mapping stores the mapping information between workers and tokens. Besides, to reduce network transfer and scheduling overheads, we partition the whole Token Bucket into multiple sub-Token Buckets (STBs), corresponding to each worker (refer to Section III-E). Token Generator generates tokens and responds to the token requests from workers. One token represents training one sub-model

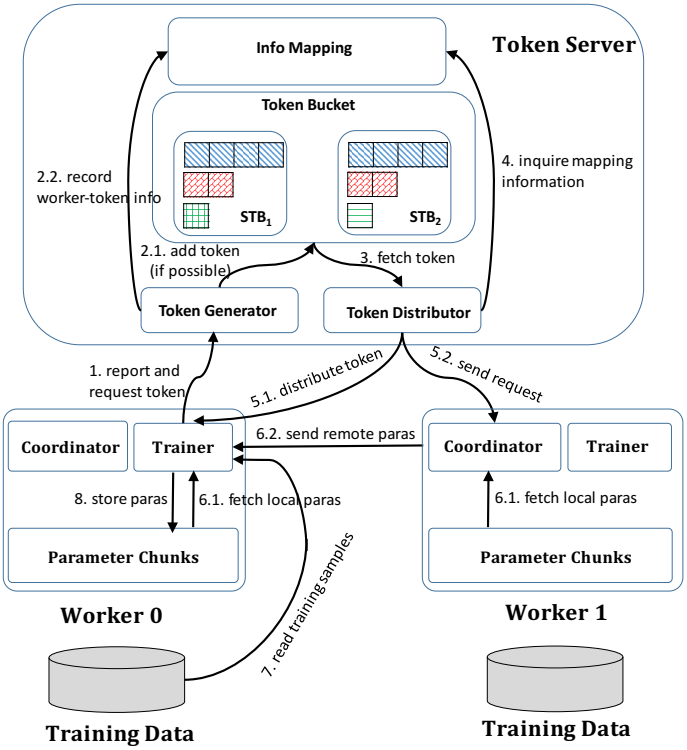


Fig. 2: Architecture of Fela

with a certain batch size. The worker, which obtains the token, is responsible to train the sub-model according to its configured parallelism degree (i.e. batch size). In order to do so, the worker may need to fetch the dependent model parameters (or the training samples) from its local storage, or the remote storage of other workers.

The sub-models are correlated with each other, and the input of some sub-models depends on the output of the others, so the tokens cannot be generated at one time. Instead, when there are tokens reported to have been completed by workers, Token Generator first records the (worker, token) information[5] in Info Mapping, then generates fresh tokens based on the completed ones, and adds them to Token Bucket.

The token generation policy considers flexible parallelism and generates different number of tokens for different sub-

[5]We have assigned a unique ID for each worker and each token during the start-up, then we insert (wid, tid) as an entry into Info Mapping, to indicate that $Worker_{wid}$ has completed $Token_{tid}$, and the output parameters of that sub-model is held by $Worker_{wid}$.

models. Afterwards, Token Distributor distributes these tokens to workers which have sent token requests. Before distributing the token, the (worker, token) information is registered in Info Mapping for future reference [6]. The distribution policy considers data locality and inquires Info Mapping, so as to maximize the chances for workers to fetch the dependent input parameters from its local storage rather than remote workers. While distributing the token to one worker, Token Distributor also notifies the other workers, which hold the dependent model parameters to train the sub-model represented by the token. These relevant workers thus will be prepared for the incoming requests from the worker and send the dependent model parameters to the worker, so that it can start training the sub-model represented by the token.

Worker Logic. On the worker side, there are three main components, namely, Trainer, Coordinator and Parameter Chunks. Parameter Chunks store the model parameters generated by Trainer or fetched by Coordinator from the remote. At the beginning of every iteration, the workers send requests to TS and ask for new tokens for training. After the worker obtains a new token, Trainer starts to train the corresponding sub-model represented by the token. It reads the training samples *mostly* from its local storage and fetches the dependent model parameters from local Parameter Chunks. If the dependent parameters are not stored in local Parameter Chunks, it will communicate with Coordinator of other workers and acquire the dependent parameters from them. After training the sub-model, it stores the output model parameters in local Parameter Chunks, which may be sent to other workers by Coordinator. Then it reports the completed token to TS and requests for another new token for training. Specially, when the tokens for one iteration are reported to have been completed by workers, Token Distributor will notify the worker to synchronize the sub-model parameters with others, and update its local model parameters after synchronization. While the worker is synchronizing and updating parameters for one sub-model, its Trainer is not blocked and can continue to request tokens of other sub-models (if any) for training.

B. Illustration of Token-Based Scheduling

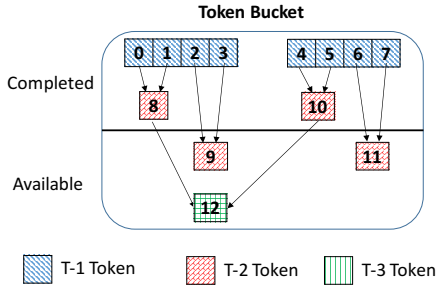


Fig. 3: Illustration of Token-Based Scheduling

Suppose we are training a DML model which has heterogeneous parallel degrees for different parts. Considering flexible parallelism, we first need to partition the model into several sub-models. The partition scheme can be user-defined. Intuitively, the finer-grained partition earns more flexible parallelism and better computation/communication overlap. However, too many sub-models also add more communication workload [7] and scheduling complexity for TS, thus degrading the training performance. In our current implementation, we use a simple but effective *bin-partitioned* method to partition the model offline (refer to Section IV-A).

For simplicity, we assume that the model is partitioned into 3 sub-models (denoted as SM-1, SM-2, and SM-3). Correspondingly, there will be 3 types of tokens generated, denoted as T-1 Tokens, T-2 Tokens, and T-3 Tokens (as shown in Figure 3[8]). SM-1, SM-2 and SM-3 have different parallel degrees. Training SM-1 needs the batch size of 16 to saturate GPU, training SM-2 needs the batch size of 32, and training SM-3 needs 64. The input of SM-2 depends on the output of SM-1, whereas the input of SM-3 depends on the output of SM-2, so T-2 Tokens and T-3 Tokens cannot be generated at the beginning. Further, to saturate GPU while training different sub-models, only when $32 \div 16 = 2$ T-1 Tokens have been completed, can 1 T-2 Token be generated and added to Token Bucket. Only when $64 \div 32 = 2$ T-2 Tokens have been completed, can 1 T-3 Token be generated and added to Token Bucket. In the case shown in Figure 3, $Token_8$ is generated and added to Token Bucket after $Token_0$ and $Token_1$ have been completed by workers. $Token_{12}$ is generated and added to Token Bucket after $Token_8$ and $Token_{10}$ have been completed.

To start the training, Token Generator first generates a certain number of T-1 Tokens, with reference to the total batch size in one iteration. For example, if the task needs to train with 128 samples (i.e. the total batch size is 128) in one iteration, then Token Generator will generate and add $128 \div 16 = 8$ T-1 Tokens to Token Bucket at the beginning of every iteration, and one T-1 Token represents training SM-1 with the 16 samples (i.e. the batch size of 16).

When the worker request for tokens, Token Distributor chooses a token from Token Bucket, which has *the most* dependent model parameters on its local Parameter Chunks (We explain more details in Section III-D). Since T-1 Tokens do not need any dependent model parameters from other workers, Token Distributor simply choose one T-1 Token randomly (or sequentially) for the worker. The worker then trains SM-1 with 16 training samples read from its local storage (if the local training samples have not been consumed). After training, the worker stores the output parameters of SM-1 in local Parameter Chunks. Then it reports to TS that the T-1 Token has been completed, and requests for new tokens. Token

[6]We register the entry (wid, tid) in Info Mapping to indicate that $Worker_{wid}$ is currently training the sub-model represented by $Token_{tid}$. Then we search in Info Mapping and find which workers are holding the dependent parameters, so that we can notify their Coordinator of sending the parameters to $Worker_{wid}$.

[7]Take VGG19 as an example, if we partition the model layer by layer, then we can get 19 CONV sub-models and 3 FC sub-models. Different workers need to transfer the boundary parameters to each other. Even with a small batch size (e.g. 16∼64), the transfer workload can be comparable to, or even larger than the data-parallel solutions.

[8]Here the tokens are numbered for better illustration. In real execution, the tokens are distributed to workers based on the sequential order of their requests, and there is no predefined correlation between tokens and workers.

Generator receives the report and records it in Info Mapping. Since there is only one T-1 Token completed, Token Generator is unable to generate T-2 Token. However, after workers have reported 2 T-1 Tokens, Token Generator is able to generate 1 T-2 Token and adds it to Token Bucket. Similarly, when 2 T-2 Tokens have been completed, Token Generator is able to add 1 T-3 Token to Token Bucket. The whole training process works in this way: Tokens are consumed by workers and generated by TS. Besides, the generation of fresh tokens depends on the completed tokens reported by workers.

C. Reactive Straggler Mitigation

Compared with previous straggler-mitigation solutions (e.g. FlexRR [11] and ElasticPipe [15]), the token-based scheduling of `Fela` avoids the arbitrary workload re-distribution from schedulers. Instead, `Fela` allows workers *to work according to their capabilities*. Faster workers with high performance request tokens more frequently from TS and earn more workload to compute, whereas slower workers request less. TS only serves as a *reactive* token producer and distributor. It does not intervene the workload distribution, but let workers determine by themselves. Therefore, `Fela` can achieve more reasonable workload distribution in time-varying environment. On the contrary, the periodic and proactive re-distribution from schedulers may fail to match the real-time situation of workers, and the length of profiling period is hard to tune. When transient stragglers [10] exist, the states of workers switch rapidly between normal workers and stragglers. If the profiling period of schedulers is too large, it can probably mistake normal workers for stragglers, reducing workload from normal workers but adding more burden to stragglers. On the other hand, if the profiling period of schedulers is too small, the schedulers need to frequently interact with workers and compute new distribution for workers, which adds more communication complexity and computation overheads to the schedulers. Besides, the small profiling period also makes schedulers more sensitive to performance fluctuation, and incur unnecessary workload migration among workers.

D. Aggressive Depth-First Scheduling (ADS) Policy

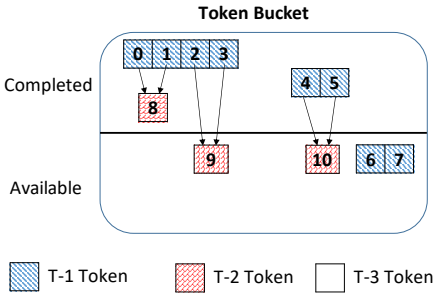


Fig. 4: Aggressive Depth-First Scheduling (ADS) Policy

When there are multiple tokens available in Token Bucket, we follow *aggressive depth-first scheduling policy* to distribute them. Generally speaking, there are two principles considered by Token Distributor for token distribution.

Principle 1: When there are tokens of different levels coexisting, Token Distributor will first distribute the token in the highest level, and there is no need to wait for the completion of all tokens in previous levels (i.e. aggressive distribution). As shown in Figure 4, there are T-1 Tokens (e.g. $Token_6$ and $Token_7$) and T-2 Tokens (e.g. $Token_9$ and $Token_{10}$) coexisting in Token Bucket. Token Distributor can distribute either T-1 Token or T-2 Token to workers. Under such situations, `Fela` prefers to distribute T-2 Token first.

The reasons for Principle 1 are due to two aspects. First, if T-2 Tokens are distributed and completed earlier, then T-3 Tokens can be generated earlier, and there can be more tokens generated in Token Bucket to avoid the *locking problem* [9]. By contrast, if T-2 Tokens are always distributed after T-1 Tokens, then when new requests come, all the completed T-1 tokens are still on the fly and have not been reported by workers, so Token Generator adds no fresh tokens to Token Bucket. New requests thus encounter the *locking* situation and have to wait until fresh tokens are available. Second, Principle 1 benefits data locality. `Fela` combines report and request for workers. In other words, when workers report the completed tokens to TS, it subsequently requests fresh tokens from TS. As shown in Figure 4, the worker has just reported the completion of T-1 tokens, and the number of T-1 tokens reaches 2. Token Generator then generates 1 T-2 Token, whose input depends on the output of the previously completed 2 T-1 Tokens. Under such a situation, it can be implied that the worker is holding at least 50% of the dependent model parameters for the T-2 Token [10]. Therefore, it is economic to distribute the T-2 Token to the worker, because it does not require much network transfer from other workers. Meanwhile, the remained T-1 Tokens can be distributed to other workers, which does not require them to fetch dependent model parameters from remote. On the contrary, if we do not follow Principle 1, and distribute another T-1 Token to the worker. Surely it does not need to fetch dependent parameters for the token, but another worker, which finally gets the T-2 Token, needs to fetch at least 50%, and probably 100%, of the dependent parameters from other workers.

Principle 2: When there are multiple tokens available in Token Bucket and they belong to the same level, `Fela` prefers to distribute the one with the most dependent parameters stored on the request worker.

As shown in Figure 4, there are 2 T-2 Tokens ($Token_9$ and $Token_{10}$) available in Token Bucket. In order to decide which T-2 token to distribute first, Token Distributor measures the data locality of the candidate tokens towards the worker, and the measurement can be formulated as follows.

$$locality\_score(wid, tid) = \frac{\parallel \mathbb{H}_{wid} \cap \mathbb{D}_{tid} \parallel}{\parallel \mathbb{D}_{tid} \parallel} \quad (1)$$

where $\mathbb{H}_{wid}$ denotes the set of tokens completed by $Worker_{wid}$, and $\mathbb{D}_{tid}$ denotes the set of dependent tokens for $Token_{tid}$.

[9] *Locking problem* means that all the tokens are consumed at this time and workers have to wait until new tokens are generated.

[10] The worker may even hold the complete dependent model parameters for the T-2 Token, if the two T-1 Tokens are both completed by itself.

Recall that we have recorded the (worker, token) information in Info Mapping, therefore, we know which worker(s) hold the dependent parameters for the candidate tokens that will be distributed. The locality score is determined by the number of tokens in $\mathbb{D}_{tid}$ that are held by $Worker_{wid}$. By inquiring Info Mapping, Token Distributor knows which tokens have been completed by the worker, so it can calculate the locality score based on the mapping information between workers and tokens. Based on that, it distributes the token with the largest locality score to the worker. Specially, when there are multiple tokens with the same locality score, Token Distributor randomly (or sequentially) choose one and distribute it to the worker.

Taking Figure 4 as an example, we have depicted the dependency relationship among tokens, i.e. $Token_9$ depends on $Token_2$ and $Token_3$, and $Token_{10}$ depends on $Token_4$ and $Token_5$. Based on the dependency relationship, we have $\mathbb{D}_9 = \{Token_2, Token_3\}$, $\mathbb{D}_{10} = \{Token_4, Token_5\}$. If $Worker_0$ comes to requests for tokens and it currently holds $Token_2$ and $Token_3$, i.e. $\mathbb{H}_{wid} = \{Token_2, Token_3\}$, then we can calculate that $locality\_score(0, 9) = \frac{2}{2} = 1$, $locality\_score(0, 10) = \frac{0}{2} = 0$, so Token Distributor will distribute $Token_9$, rather than $Token_{10}$, to $Worker_0$. On the other hand, if $Worker_0$ holds $Token_3$ and $Token_4$, i.e. $\mathbb{H}_{wid} = \{Token_3, Token_4\}$, then $locality\_score(0, 9) = locality\_score(0, 10) = \frac{1}{2} = 0.5$. In that case, either $Token_9$ or $Token_{10}$ can be distributed to $Worker_0$. In our implementation, we choose the one with the smallest token ID (i.e. $Token_9$) and distribute it to the worker.

E. Hierarchical Fetching (HF) Policy

Although ADS Policy can benefit data locality in `Fela`, we still observe a considerable amount of data transfer among workers during each iteration. Besides, when workers perform at a similar speed, multiple workers may send requests for new tokens at the same time, and cause *fetching conflicts*. In order to prevent one token from being distributed to more than one worker, Token Distributor needs to use locking mechanism to avoid conflicts, and at least one worker will encounter fetching failure and need to be re-distributed with a token, thus incurring additional scheduling overheads. We illustrate this by using Figure 4 as an example.

Suppose there are two workers, namely, $Worker_0$ and $Worker_1$. If $Worker_1$ is much slower than $Worker_0$, then $Worker_0$ may obtain $Token_0$ and $Token_1$ prior to $Worker_1$. After $Worker_0$ reports the completeness of $Token_0$ and $Token_1$, $Token_8$ is generated in Token Bucket, and $Worker_0$ can obtain $Token_8$ according to ADS Policy, and continue its training without any network transfer from the other worker. However, when their speeds are similar, $Worker_0$ and $Worker_1$ may request for $Token_0$ simultaneously. Confronted with such a situation of conflict, Token Distributor has to use locking mechanism to serialize the requests and guarantee that $Token_0$ is only assigned to one worker (e.g. $Worker_0$). The other worker (e.g. $Worker_1$) encounters a fetching failure and Token Distributor needs to roll back and re-distribute another token (e.g. $Token_1$) for it. Afterwards, $Worker_0$ and $Worker_1$ report the completeness of $Token_0$ and $Token_1$ at nearly the same time, and they request for new Tokens (i.e. $Token_8$) concurrently, so Token Distributor again resorts to locking. Besides, either $Worker_0$ or $Worker_1$ obtains $Token_8$, it needs to fetch the output data (of $Token_0$ or $Token_1$) from the other worker.

In order to improve data locality, as well as reduce fetching conflicts and scheduling overheads, we initially partition the whole Token Bucket into several sub-Token Buckets (STBs). Each worker corresponds to one sub-Token Bucket (STB). The worker first consumes the tokens in its STB. Only after the STB is empty, will the worker fetch the tokens from other STBs. During the whole process, ADS Policy is still followed. While the worker is consuming the tokens in its own STB, the data locality is maximized and there is no network transfer. After its STB is empty, the worker becomes a *helper* and turns to fetch tokens from the STB of stragglers, following ADS Policy. New helpers will be prioritized to assist the straggler with the least helpers and the slowest progress.

Based on the partitioned Token Buckets and HF Policy, we have achieved two targets: 1) While all workers are working on their own STBs, there is no *fetching conflict* or *fetching failure* among workers, and locking mechanism is unnecessary during this stage. Therefore, the scheduling overheads are reduced. 2) Only when some workers have emptied their own STBs, will locking mechanism be needed for serialization. Meanwhile, the helper prioritization also mitigates *fetching conflicts* after some workers have emptied their own STBs and become *helpers*, because different *helpers* are less likely to help the same straggler when there are still many stragglers. Besides, the performance gap between *helpers* and stragglers are relatively large, which also reduces the possibility for them to contend for the same token.

F. Conditional Token Distribution (CTD) Policy

Previous works [3], [6] have also observed that different sub-models show a distinct difference in computation and communication cost. Take the typical convolutional neural networks (CNNs) as an example, CONV layers can consume $>$90% of the computation power but only generate $<$10% synchronization cost, and FC layers go the opposite. Therefore, if we treat each sub-model equally in DML training, it will cause an imbalanced utilization of computation and communication resources. To address this problem, `Fela` supports *conditional token distribution policy* to balance the resource utilization. As shown in Figure 3, we suppose SM-1 and SM-3 are computation-intensive (e.g. CONV layers) whereas SM-2 are communication-intensive (e.g. FC layers). Then, it is not cost-effective to employ the whole cluster to train the SM-2 represented by T-2 Tokens, because they do not require so much computation power but the large-scale synchronization will cause heavy communication workload in the cluster. Targeting at this, `Fela` pre-defines a subset of workers (denoted as $\mathbb{S}$) to undertake training of T-2 Tokens.

(1) $\forall i \in \mathbb{S}$, Token Distributor prioritize T-2 Token distribution for $Worker_i$. As for the other tokens than T-2 Tokens,

ADS Policy is still followed. In other words, the scheduling priority becomes T-2>T-3>T-1 for $Worker_i$.

(2) $\forall j \notin \mathbb{S}$, Token Distributor will not distribute T-2 Tokens for $Worker_j$, and the distribution of other tokens still follows ADS Policy. In other words, the scheduling priority becomes T-3>T-1 for $Worker_j$.

With CTD Policy, `Fela` only needs to synchronize the parameters of communication-intensive sub-models within a small set of workers (i.e. $\mathbb{S}$), so the communication cost can be much saved.

IV. Implementation

In order to implement `Fela` and achieve high performance, there are two main issues to consider. First, the training model needs to be properly partitioned. Second, the hyper-parameters (i.e. the parallelism degrees for each sub-model and the conditional subset size) need to be tuned. By considering both the effectiveness and efficiency, we design the configuration tuning mechanism for `Fela`, which combines both offline model partition and runtime configuration tuning to identify the *near-optimal* configurations for high performance.

A. Offline Model Partition

Model partition is completed offline and does not change during runtime. The aim is to partition the model into several sub-models, with each enjoying *approximately* equal parallelism degree. In order to earn the maximum flexibility, finer-grained model partition is preferred. For example, in CNN, we can partition each layer as a sub-model. However, finer-grained partition can generate more sub-models and `Fela` needs to find the optimal or near-optimal parallelism degree for each, thus leading to an exhaustive search in a huge space. Therefore, to reduce the searching complexity, we need to control the number of sub-models. Targeting at this, we design a *bin-partitioned method*, which can efficiently partition the model in a coarse-grained manner.

Although the CNN model can be composed of many layers, there are only a limited number of shapes for these layers. For example, VGG19 contains 16 CONV layers and 3 FC layers, but these layers can be categorized into 6 types according to their shapes (i.e. 5 types of CONV layers and 1 type of FC layer). For each type, we measure the *threshold batch size* to reach the maximum throughput (refer to Figure 1) [11], and we can further find that some types of layers have very close threshold batch sizes [12], whereas others have very different threshold batch sizes [13]. We arrange the *threshold batch sizes* according to the location order in VGG19 (refer to Figure 5).

Then we design a series of bins to partition the whole model into sub-models. Note that different bin sizes are achievable based on the desired partition granularity (i.e. different number of sub-models), and further affect the complexity of configuration tuning. In our experiments, We choose 16 as the bin size [14], then the bins are $[0, 16), [16, 32), [32, 48), [48, 64), \cdots$. In this way, the 19-layer VGG model is partitioned into 3 sub-models (shown in Figure 5), i.e. Layer 1∼8 (CONV), Layer 9∼16 (CONV), and Layer 17∼19 (FC). For GoogLeNet, the network structure is even simpler and is partitioned into 3 models, i.e. Layer 1∼4 (CONV), Layer 5∼9 (CONV), and Layer 10∼12 (CONV+FC).

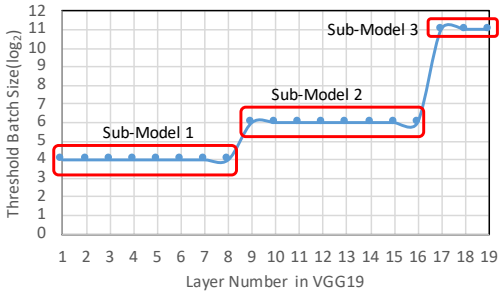


Fig. 5: Threshold Batch Sizes of Different Layers in VGG19

B. Runtime Configuration Tuning

The configuration tuning is completed during runtime, and it is executed in two phases. First, we profile *per-iteration time* for different configurations of parallelism degrees, without considering *CTD Policy*. Second, we target at the sub-models containing FC layers (communication-intensive) and search for the optimal (or near-optimal) size of conditional subset.

Phase 1: Parallelism Degree Tuning. It is time-consuming to find the optimal parallelism degrees in a huge continuous search space. To reduce the tuning complexity, we only consider discreet candidate values to configure the parallelism degrees for sub-models.

Suppose there are M sub-models partitioned, denoted as SM-1, SM-2, $\cdots$, SM-M. We assign each of them a weight w_i. We let $w_1 = 1$, and choose it as the base. The number of T-1 Tokens is determined by the total batch size and its *threshold batch size*, and can be calculated as

$$n_1 = \max(\frac{total\_batch\_size}{threshold\_batch\_size}, N) \quad (2)$$

N is the worker number, because we guarantee that there are at least 1 T-1 Token on each worker, in order to reduce the idle time and skewed consumption of training samples at the beginning. Following that, the number of T-i tokens is calculated as $n_i = \frac{w_i}{w_1} \times n_1$.

The aim of parallelism degree tuning is to find such a sequence $\{w_1, w_2, \cdots, w_M\}$ to achieve the lowest *per-iteration time*. Instead of searching in continuous space, we only consider the discrete candidate values, i.e. $\{1, 2, 4, \cdots, 2^{\lfloor log_2 N \rfloor}\}$. Given the structural characteristics of CNN model, when it goes deeper, the parallelism degree becomes larger. Based on this prior knowledge, we can further reduce the searching space by constraining $w_{i+1} \geq w_i$. Since our test bed contains 8 nodes ($N = 8$), we only consider the candidate values $\{1, 2, 4, 8\}$. Besides, since there are 3 sub-models partitioned ($M = 3$), and we have $w_{i+1} \geq w_i$, the searching space is reduced to $4 + 3 + 2 + 1 = 10$ cases.

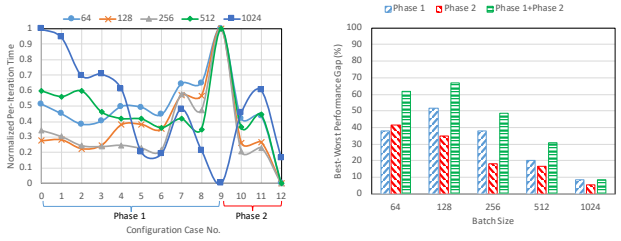
[11] The measurement is executed *once and for all*, and the profiled *threshold batch sizes* for different shapes can also be stored in repository and reused while running other DML tasks.

[12] For example, (64,64,224,224) and (128, 128,112,112) both reach the maximum throughput around the batch size of 16.

[13] For example, FC layer requires nearly 2048 batch size to reach the maximum throughput, which is distinctly different from all CONV layers

[14] The reason is because each layer needs at least 16 batch size to reach the maximum throughput (i.e. saturate the GPU) according to our profiling.

Phase 2: Conditional Subset Tuning. Parallelism degree tuning helps to better saturate GPU power whereas conditional subset tuning aims to better reduce the synchronization overheads for communication-intensive sub-model(s) (i.e. FC layers in CNN). To reduce the searching complexity, we halve the size of the conditional subset every time and measure the *per-iteration time*. For example, in our 8-node cluster, after the parallelism degree is fixed, we measure the *per-iteration time* with the subset size of 8,4,2,1, respectively. Therefore, there are $log_2 8 = 4$ cases to compare [15].



(a) Performance Tuning with Different Configuration Cases (b) Best-Worst Performance Gaps

Fig. 6: Illustration of Configuration Tuning

Figure 6 shows an example of the configuration tuning process, which measures the *per-iteration time* for training VGG19 with different batch sizes. During Phase 1, `Fela` searches for the best configuration of parallelism degrees. After Phase 1, the parallelism degrees for different sub-models are fixed to the best one found so far, and `Fela` continues to search for the best configuration of conditional subset size. Combing the two phases (i.e. parallelism degree tuning and conditional subset tuning), we get $10 + 4 - 1 = 13$ cases to search for each round of training. For each case, we measure the average of *per-iteration time* for 5 iterations, thus leading to 65 iterations of warm-up cost, which is trivial for training typical DML tasks (usually cost hundreds of thousands of iterations [30] [31]). After the warm-up stage, the near-optimal configuration is fixed and used for the following the subsequent training process.

Figure 6(a) illustrates the performance fluctuation for different configuration cases. Since the values of *per-iteration time* differ a lot while training with different batch sizes, in order to illustrate their fluctuation in one figure, we normalize the values to the scale of $[0, 1]$ [16]. From Figure 6(a) it can be implied that, the best performance (i.e. minimum *per-iteration time*) is obtained at different configurations while training with different batch sizes. For instance, while training with 64 batch size, it earns the smallest *per-iteration time* at Case 2 during Phase 1, and the parallelism degree configuration is $\{1, 1, 4\}$. Afterwards, the parallelism degree is fixed and it earns the smallest *per-iteration time* at Case 12 during Phase 2, and the conditional subset size is 1. On the other hand, while training with 1024 batch size, it earns the smallest *per-iteration time* at Case 9 during Phase 1, and the parallelism degree configuration is $\{1, 8, 8\}$. Afterwards, it searches for the optimal subset size during Phase 2, but finds Case 9 is still the best one, so `Fela` chooses 8 as the conditional subset size (i.e. no conditional token distribution) for its training process.

To further demonstrate the effectiveness of our configuration tuning mechanism, we calculate the performance gap between the best case and worst cases for both Phase 1 and Phase 2. As shown in Figure 6(b), among all the configuration cases in Phase 1 (i.e. Case 0$\sim$ Case 9), the best configuration case saves *per-iteration time* by 8.51%$\sim$51.69%, compared with the worst one. Among all the configuration cases in Phase 2 (Case 10$\sim$Case 12, together with the best case in Phase 1), the best configuration case saves *per-iteration time* by 5.31%$\sim$41.25%. Overall, among all the cases in Phase 1 and Phase 2, the best configuration case can outperform the worst one by 8.51%$\sim$66.78%. In other words, by paying the trivial cost of configuration tuning, `Fela` is able to avoid the possible performance degradation by 8.51%$\sim$66.78%, which strongly proves the necessity and effectiveness of the configuration tuning mechanism.

C. Other Implementation Details

We implement `Fela` and the baseline prototypes based on Pytorch due to its ease of model partition. Meanwhile, we adopt Gloo as the backend and TCP protocol is used for parameter transmission/synchronization. Noticing that the built-in hook mechanism of Pytorch cannot suffice the needs to control the computation workflow for `Fela`, we design an enhanced hook mechanism by adding *virtual layers* in the training model, which facilitates us to capture the input/output for each sub-model, but introduces trivial overheads. We have described the *virtual layer* implementation in our workshop paper [15] and omit the details here due to space limit.

V. Experimental Evaluation

A. Experiment Setting

We conduct both ablation study and comparative evaluation on a 8-node cluster, with all nodes directly connected to a 40GE switch. Each server is equipped with one Nvidia Tesla K40c GPU and Mellanox Connectx-3 NIC. The in-bound/outbound bandwidth is 10Gbps for each link.

As for the ablation study, we measure the performance improvement for each strategy/policy. As for the comparative evaluation, we choose three baselines for performance comparison, namely, data-parallel (DP) baseline, model-parallel (MP) baseline (refer to PipeDream [12] and ElasticPipe [15]), and hybrid-parallel (HP) baseline (refer to Stanza [6]). We choose VGG19 and GoogLeNet as the benchmarks [17] and execute all the cases under BSP. To demonstrate the outperformance of `Fela`, we compare the *average throughput* (AT) by training

[15]Besides reducing the searching space, there is another reason for us to abandon the case with subset size of 3,5,7, because these subset sizes are not divisible by the total number of nodes (i.e. 8), and they cannot evenly share the workload of the whole cluster. Therefore, the load imbalance can usually degrade the overall performance seriously.

[16]For example, while training with the batch size of 1024, we get 13 values of *per-iteration time*, denoted as $t_0, t_1, \cdots, t_{12}$. Then we normalize them as $Normalize(t_i) = \frac{t_i - \min\limits_{0 \le j \le 12} t_j}{\max\limits_{0 \le j \le 12} t_j - \min\limits_{0 \le j \le 12} t_j}$. The same normalization is also applied for other training rounds with different batch sizes

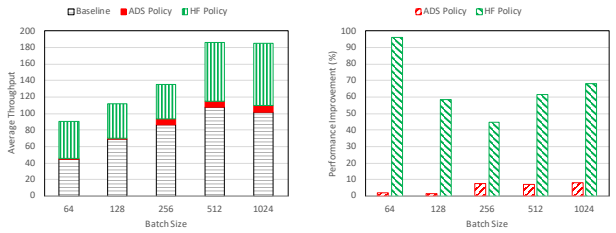
[17]The input size is ($batch\_size$, 3, 224, 224) for VGG19 and ($batch\_size$, 3, 32, 32) for GoogLeNet.

the benchmark for equal number of iterations (i.e. 100 iterations) [18], i.e.

$$AT = \frac{total\_batch\_size \times iter\_n}{total\_time} \quad (3)$$

where $iter\_n = 100$ is all the following experiments, and $total\_time$ is the total time to complete the 100 iterations.

B. Ablation Study



(a) Throughput Improvement (b) Percentage of Improvement

Fig. 7: Ablation Study (ADS Policy and HF Policy)

Since the configuration tuning mechanism has proved the effectiveness of flexible parallelism degree and CDT Policy, we turn back to validate the effectiveness of the other scheduling policies (i.e. ADS Policy and HF Policy). We apply the tuned configurations to the comparative cases with and without the policy, and then measure the performance gains for each policy. The results can be illustrated as Figure 7.

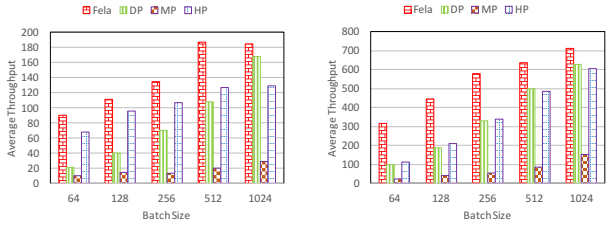
Combined with the measured results in Figure 6, we summarize the performance improvement for every strategy/policy as Table III. Specifically, While training with different batch sizes, different strategy/policy can affect the performance to different extent. For example, ADS Policy is more effective while training with large batches (as shown in Figure 7(b), whereas CDT Policy is more effective while training with small batch sizes (as shown in Figure 6(b)). Generally speaking, the effectiveness of different strategy/policy is related to the trade-off between computation overheads and communication overheads. Each strategy/policy may boost the performance to its greatest extent at different batch sizes.

TABLE III: Summary of Ablation Study

Strategy/Policy	Performance Improvement
Parallelism Degree Tuning	8.51%∼51.69%
ADS Policy	1.64%∼8.21%
HF Policy	44.80%∼96.30%
CDT Policy	5.31%∼41.25%

C. Comparative Experiment Results

1) Non-Straggler Scenario: We first compare the *average throughput* of different solutions in the non-straggler scenario. The experimental results in Figure 8 show that `Fela` distinctly outperforms all the three baselines (i.e. DP, MP and HP) in *average throughput*. More specifically, as for VGG19 benchmark, `Fela` outperforms DP by 9.98%∼3.23×, MP by 5.18×∼8.12×, HP by 15.77%∼49.65%. As for GoogLeNet benchmark, `Fela` outperforms DP by 13.25%∼2.15×, MP by 3.63×∼12.22×, HP by 19.01%∼1.85×.



(a) VGG19 (b) GoogLeNet

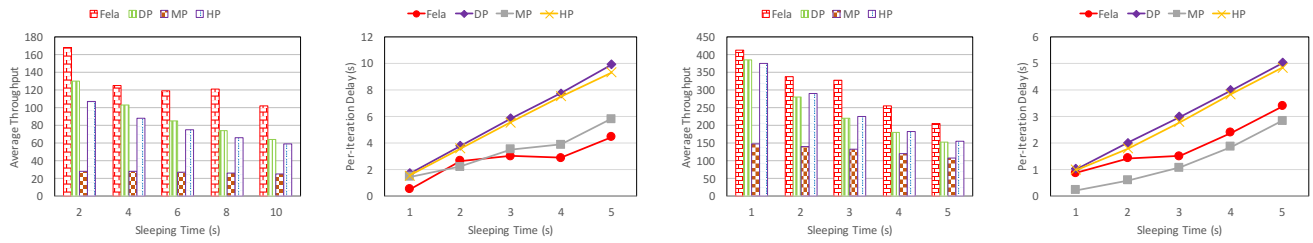
Fig. 8: AT Comparison in Non-Straggler Scenario

The comparative results indicate that MP gets the worst performance under BSP. The reasons can be attributed to two main aspects. First, when there are multiple workers, MP suffers from the worst work conservation due to its pipeline workflow. Considering the 8-worker setting in our experiment, the majority of workers remain idle during one iteration, waiting for the dependent data transferring from its neighbors. Second, MP uses small and fixed micro-batches [13] during each sub-iteration, in order to amortize the bubble time. However, the small and fixed micro-batch size leads to the under-utilization of GPU resources, which further prolongs the iteration time and degrades the throughput performance.

Besides, HP outperforms DP at the beginning, but as the batch size grows larger, it falls behind DP. The reasons can also be attributed to two main aspects. The first aspect lies in bad work conservation of HP, and the second aspect lies in the increasing network transfer when the batch size grows large. In our implementation of HP, we inherit the configuration in [6] (7 CONV workers and 1 FC worker). Therefore, the 1 FC worker has to remain idle at the beginning of each FP process (before it receives the output from the CONV workers) and at the end of each BP process (after it has sent out its output to CONV workers). However, HP avoids the significant network transfer of FC layers' synchronization because it only maintains the FC layers in one worker. By contrast, DP has to synchronize the parameters of FC layers during each iteration. Therefore, HP takes an advantage over DP at the beginning, and the synchronization benefit outweighs the overheads due to bad work conservation. However, the network transfer amount of HP is proportional to the batch size during each iteration. When the batch size grows large (e.g. 1024), HP also needs significant network transfer, especially for the FC worker, which needs to receive/send data from/to all the other 7 CONV workers. In this way, the FC worker can become a centralized bottleneck. By contrast, the amount of network transfer in DP does not change as the batch size grows. Therefore, as the batch size grows larger, the synchronization benefit of HP declines and falls behind DP eventually. Compared with DP and HP, `Fela` enjoys more flexibility based on the elaborate token-based scheduling policies and effective tuning, thus it can achieve better performance towards different workload.

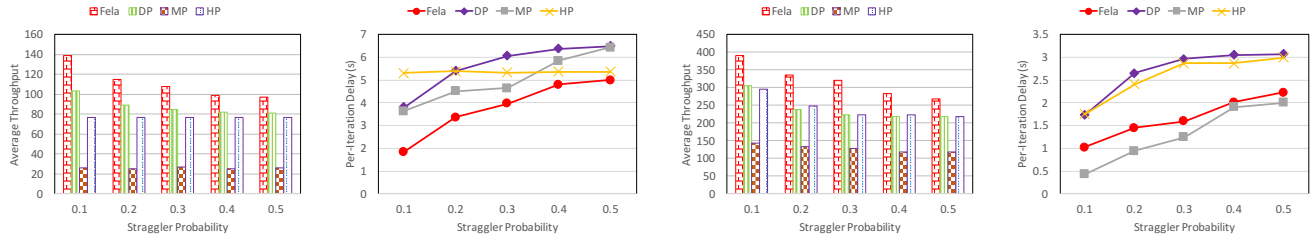
2) Straggler Scenario: We follow the method in [10], [11] to generate straggler effect and add sleeping delays to workers,

[18]Since `Fela` makes no changes to the training algorithm and does not affect the iteration quality, the convergence results of the solutions should be the same after equal number of iterations. Therefore, we compare the training throughput rather than the convergence speed.



(a) AT (VGG19) (b) PID (VGG19) (c) AT (GoogLeNet) (d) PID (GoogLeNet)

Fig. 9: AT and PID Comparison in Round-Robin Straggler Scenario



(a) AT (VGG19) (b) PID (VGG19) (c) AT (GoogLeNet) (d) PID (GoogLeNet)

Fig. 10: AT and PID Comparison in Probability-Based Straggler Scenario

so as to prolong their computation time. More specifically, we have designed two straggler scenarios, namely, *round-robin straggler scenario* and *probability-based straggler scenario*. Then, we compare the *average throughout* (defined as Equation 3) and *per-iteration delay* (PID) for different solutions. As for PID, it is defined as

$$PID = \frac{total\_time_s - total\_time_0}{iter\_n} \quad (4)$$

where $total\_time_s$ is the total time to complete the iterations in straggler environment, $total\_time_0$ is the total time to complete the iterations in non-straggler environment, and $iter\_n$ is the number of iterations (i.e. $iter\_n = 100$).

Round-Robin Straggler Scenario: We create the straggler scenario described in [10], which slows down each worker by d seconds in a round-robin way. In other words, $Worker_0$ is slowed down by d seconds in Iteration 0, $Worker_1$ is slowed down by d seconds in Iteration 1, and so on. Considering that VGG19 and GoogLeNet show distinct difference in iteration time, we use different ranges of d for them. As for VGG19, $d = 2, 4, 6, 8, 10s$. As for GoogLeNet, $d = 1, 2, 3, 4, 5s$.

Probability-Based Straggler Scenario: In probability-based straggler scenario. during every iteration, each worker becomes the straggler at the probability of p. As shown in Figure 10(b) and Figure 10(d), $p = 0.1, 0.2, 0.3, 0.4, 0.5$. As for VGG19, the straggler is slowed down by $d = 6s$. As for GoogLeNet, the straggler is slowed down by $d = 3s$.

Summary of Results: The comparison of average throughput and per-iteration delay is shown in Figure 9 and Figure 10.

(1) *Round-Robin Straggler Scenario*: As for VGG19 (refer to Figure 9(a)), `Fela` improves *average throughput* by 28.6%∼60.0% compared with DP, by 3.01×∼4.87× compared with MP, and by 41.61%∼84.16% compared with HP. As for GoogLeNet (refer to Figure 9(c)), `Fela` improves *average throughput* by 7.27%∼32.87% compared with DP, by 90.13%∼1.78× compared with MP, and by 9.72%∼45.19% compared with HP.

(2) *Probability-Based Straggler Scenario*: As for VGG19 (refer to Figure 10(a)), `Fela` improves *average throughput* by 19.58%∼33.91% compared with DP, by 2.70×∼4.25× compared wIth MP, and by 27.13%∼80.29% compared with HP. As for GoogLeNet (refer to Figure 10(c)), `Fela` improves *average throughput* by 22.94%∼43.73% compared with DP, by 1.27×∼1.71× compared with MP, and by 23.28%∼43.46% compared with HP.

Besides, we depict *per-iteration delay* under different straggler settings (shown in Figure 9(b), Figure 9(d), Figure 10(b), and Figure 10(d)). It can be found that `Fela` achieves much smaller *per-iteration delay* compared with DP and HP.

(1) *Round-Robin Straggler Scenario*: As for VGG19 (refer to Figure 9(b)), `Fela` reduces *per-iteration delay* by 30.35%∼68.19% compared with DP, and by 26.00%∼64.86% compared with HP. As for GoogLeNet (refer to Figure 9(d)), `Fela` reduces *per-iteration delay* by 30.35%∼68.19% compared with DP, and by 12.56%∼46.01% compared with HP.

(2) *Probability-Based Straggler Scenario*: As for VGG19 benchmark (refer to Figure 10(b)), `Fela` reduces *per-iteration delay* by 23.23%∼51.36% compared with DP, and by 6.97%∼65.12% compared with HP. As for GoogLeNet benchmark (refer to Figure 10(d)), `Fela` reduces *per-iteration delay* by 27.62%∼46.22% compared with DP, and by 25.79%∼44.39% compared with HP.

It can also be observed that *per-iteration delay* of `Fela` is larger than MP in some cases. The reason is mainly due to the worst work conservation of MP solution. As aforementioned, even in non-straggler scenario, MP cannot well utilize the GPU of each worker. The majority of workers remain idle for considerable time during each iteration. Therefore, even the idle worker becomes a straggler (i.e. we delay the worker to start its computation), it imposes little effect compared with

the non-straggler scenario. The sleeping delay just overlaps with the original idle time, and the worker does not contribute to the overall training process for either straggler scenario or non-straggler scenario. Besides, we can also note that the *average throughput* performance of MP is the lowest among all the solutions in both non-straggler scenario and straggler scenario, which cannot compete against Fela in large-scale and large-batch training.

VI. Related Work

Data-Parallel, Model-Parallel and Hybrid-Parallel DML Solutions. Data-parallel solutions are most widely applied in previous DML practice due to its easy implementation. FlexPS [17] is a recent representative of data-parallel solutions, which strengthens the performance by considering flexible parallelism during different training stages. However, the increasing training data and large model size cause data-parallel solutions unable to tackle the DML tasks efficiently. PipeDream [12] and its subsequent variants (e.g. GPipe [13] and Dual Pipe [14]) partition the large model and train them in a pipe-based manner, but suffer from parameter staleness and load imbalance. ElasticPipe [15] introduces elastic tuning mechanism in model-parallel solutions and improves straggler-tolerance in time-varying environment. However, all these works earn the communication benefit but fail to fully utilize the computation power of nodes. Stanza [6] incorporates hybrid parallelism and better utilizes computation power than pure model-parallel solutions. But Stanza causes idle time for some workers and cannot achieve good work conservation. In contrast, Fela achieves *finer-grained flexible parallelism* and *reactive straggler mitigation*, which have been considered by none of existing works. Generally speaking, all the aforementioned works (e.g. PipeDream, ElasticPipe, FlexPS, Stanza, etc) can be considered as a special case of Fela.

BSP, ASP and SSP. BSP [33] is the typical mode for parameter synchronization in large-scale DML. ASP [1] and SSP [8]–[10] also earn much attention in recent years due to the emerging demand of overcoming communication bottleneck. Fela leverages BSP for DML to avoid algorithmic uncertainties and damage to reproducibility. However, BSP and ASP can be regarded as special cases of SSP. and it is worth noting that Fela can be easily extended to SSP by adding the age attribute to each token. By considering the age of token, Fela can distribute the tokens according to the predefined staleness bound, thus executing DML under SSP.

VII. Conclusion

We present Fela, which incorporates flexible parallelism and elastic tuning mechanism to accelerate large-scale DML. Fela adopts token-based scheduling and integrates elaborate scheduling policies to effectively accelerate DML training. Our comparative experiments on two benchmarks prove the outperformance of Fela over both data-parallel, model-parallel and hybrid-parallel baselines. Specifically, compared with the data-parallel baseline, Fela can improve the training throughput by 7.27%$\sim$3.23$\times$; compared with the model-parallel baseline, Fela can improve the training throughput by 90.13%$\sim$8.12$\times$; compared with the hybrid-parallel baseline, Fela can improve the training throughput by 9.72%$\sim$1.85$\times$.

References

[1] L. Mu *et al.*, "Scaling distributed machine learning with the parameter server," in *Proceedings of OSDI'14*.

[2] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of OSDI'16*.

[3] H. Zhang *et al.*, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *Proceedings of ATC '17*.

[4] S. Wang *et al.*, "BML: A high-performance, low-cost gradient synchronization algorithm for dml training," in *Proceedings of NeurIPS'18*.

[5] J. Geng *et al.*, "HiPS: Hierarchical parameter synchronization in large-scale distributed machine learning," in *Proceedings of ACM SIGCOMM Workshop on NetAI'18*.

[6] X. Wu *et al.*, "Stanza: Layer separation for distributed training in deep learning," *arXiv preprint arXiv:1903.06701v1*, 2018.

[7] A. Sapio *et al.*, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1812.10624*, 2019.

[8] J. Cipar *et al.*, "Solving the straggler problem with bounded staleness," in *Proceedings of HotOS'13*.

[9] Q. Ho *et al.*, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proceedings of NeurIPS'13*.

[10] H. Cui *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *Proceedings of USENIX ATC'14*.

[11] A. Harlap *et al.*, "Addressing the straggler problem for iterative convergent parallel ml," in *Proceedings of ACM SoCC'16*.

[12] H. Aaron *et al.*, "Pipedream: Pipeline parallelism for dnn training," in *Proceedings of SysML'18*, 2018.

[13] Y. Huang *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *arXiv preprint arXiv:1811.06965*, 2018.

[14] C.-C. Chen *et al.*, "Efficient and robust parallel dnn training through model parallelism on multi-gpu platform," *arXiv:1809.02839*, 2018.

[15] J. Geng *et al.*, "ElasticPipe: An efficient and dynamic model-parallel solution to DNN training," in *Proceedings of HPDC Workshop on ScienceCloud '19*.

[16] Z. Jia *et al.*, "Beyond data and model parallelism for deep neural networks," in *Proceedings of SOSP'19*.

[17] Y. Huang *et al.*, "FlexPS: Flexible parallelism control in parameter server architecture," in *Proceedings of VLDB'18*.

[18] L. Luo *et al.*, "Parameter Hub: A rack-scale parameter server for distributed deep neural network training," in *Proceedings of SoCC'16*.

[19] C. Guo *et al.*, "RDMA over commodity ethernet at scale," in *Proceedings of ACM SIGCOMM '16*.

[20] Mellanox Corp., "Mellanox corporate update: Unleashing the power of data," 2019. [Online]. Available: http://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf

[21] R. Mittal *et al.*, "Revisiting network support for RDMA," in *Proceedings of ACM SIGCOMM'18*.

[22] Y. You *et al.*, "Imagenet training in minutes," in *Proceedings of ICPP'18*.

[23] S. L. Smith *et al.*, "Don't decay the learning rate, increase the batch size," in *Proceedings of ICLR'18*.

[24] P. Goyal *et al.*, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv:1706.02677*, 2017.

[25] A. Takuya *et al.*, "Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes," in *Proceedings of NeurIPS 2017 Workshop: Deep Learning At Supercomputer Scale*.

[26] X. Zhang *et al.*, "Polynet: A pursuit of structural diversity in very deep networks," in *Proceedings of CVPR'17*.

[27] J. Oh *et al.*, "Fast and robust parallel SGD matrix factorization," in *Proceedings of KDD'15*.

[28] J. Geng *et al.*, "Rima:an RDMA-accelerated model-parallelized solution to large-scale matrix factorization," in *Proceedings of ICDE'19*.

[29] J. Geng *et al.*, "Accelerating distributed machine learning by smart parameter server," in *Proceedings of SIGCOMM Workshop on APNet'19*.

[30] S. Karen *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556v6*, 2015.

[31] S. Karen *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556v6*, 2015.

[32] G. Andrew. (2017) Bringing hpc techniques to deep learning. [Online]. Available: http://research.baidu.com/bringing-hpc-techniques-deep-learning

[33] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, Aug. 1990.