2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) | 978-1-7281-7002-2/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICDCS47774.2020.00018

Elan: Towards Generic and Efficient Elastic Training for Deep Learning

Lei Xie*, Jidong Zhai*, Baodong Wu†, Yuanbo Wang†, Xingcheng Zhang†, Peng Sun†, Shengen Yan†
*Tsinghua University, †Sensetime
Beijing, China
Email: *xie-l18@mails.tsinghua.edu.cn, zhaijidong@tsinghua.edu.cn,
†{wubaodong, wangyuanbo, zhangxingcheng, sunpeng1, yanshengen}@sensetime.com

***Abstract*—Showing a promising future in improving resource utilization and accelerating training, elastic deep learning training has been attracting more and more attention recently. Nevertheless, existing approaches to provide elasticity have certain limitations. They either fail to fully explore the parallelism of deep learning training when scaling out or lack an efficient mechanism to replicate training states among different devices.**

To address these limitations, we design Elan, a generic and efficient elastic training system for deep learning. In Elan, we propose a novel *hybrid scaling* mechanism to make a good trade-off between training efficiency and model performance when exploring more parallelism. We exploit the topology of underlying devices to perform *concurrent* and *IO-free* training state replication. To avoid the high overhead of start and initialization, we further propose an *asynchronous* coordination mechanism. Powered by the above innovations, Elan can provide high-performance ($\sim$1s) migration, scaling in and scaling out support with negligible runtime overhead ($<3‰$). For elastic training of ResNet-50 on ImageNet, Elan improves the time to solution by 20%. For elastic scheduling, with the help of Elan, resource utilization is improved by $21\%+$ and job pending time is reduced by $43\%+$.

***Index Terms*—Distributed Deep Learning Training, Elasticity, Elastic Training, Elastic Scheduling**

I. Introduction

Deep learning (DL) has made a significant success in a wide range of areas, such as face detection [1], automated driving [2] and natural language processing [3]. Due to its compute-intensive nature, distributed training and special hardware are utilized to accelerate the training process, e.g., enterprises and cloud providers typically build large-scale GPU clusters for DL training [4]. Showing a promising future in improving resource utilization and accelerating training, elastic DL training becomes the frontier of new training paradigm development and attracts more and more attention [5], [6].

On one hand, elasticity is beneficial to DL training job scheduling that resources allocated to jobs can be dynamically adjusted for better resource usage. For example, training jobs can scale out to leverage extra resources when a cluster is under-utilized, while they scale in to concede resources in case of no enough available resources for new submitted jobs. In this way, we can mitigate the poor resource usage problem, as observed in a production cluster of Sensetime company (Figure 1) where the dramatic fluctuation in GPU resource utilization and unnecessary job pending are mainly due to the

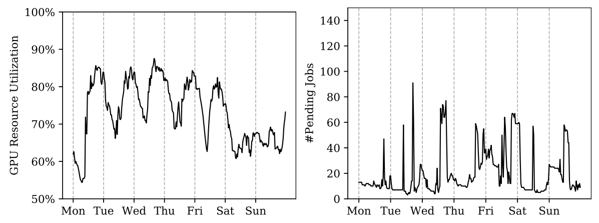


Fig. 1: GPU resource utilization (#Used GPUs / #Total GPUs) and the number of pending jobs in a DL training cluster of Sensetime company with about 2,000 GPUs.

lack of elasticity. Previous research work, such as Gandiva [5] and Optimus [6], has also shown that elasticity can improve the quality of service by reducing the job completion time.

On the other hand, DL training jobs themselves are becoming more and more elastic. For instance, researchers have been proposing new elastic training algorithms, e.g., training using dynamic batch sizes [7], [8] or optimizing with elasticity-aware SGD [9], [10]. Different from static training schemes, these algorithms present a dynamic resource requirement during training, which, however, is not fully considered in current DL frameworks so that training with dynamic resource requirements but on a fixed number of resources even results in an increased time to solution (§VI-B).

A. Challenges

There are two main challenges to provide elasticity support for DL training jobs. One of them is *how to explore more parallelism when scaling out*. It is challenging because the parallelism of data-parallel DL training is mainly related to the batch size, while the batch size influences not only training efficiency (e.g., training throughput) but also model performance (e.g., classification accuracy) [11]. Scaling out with a fixed batch size comes with limited parallelism and diminishing marginal gains, which harms training efficiency due to under-utilized resources. While scaling out with an increased batch size enlarges parallelism but damages model performance since model performance tends to reduce with larger batch sizes.

Previous work has tried some potential solutions towards this challenge, but they have many limitations. Most systems (e.g., Optimus [6], Falcon [12], [13]) do not alter the batch size in fear of reducing model performance, thus limiting training efficiency. Only a few systems adjust parallelism when scaling out, e.g., Gandiva [5] scales the batch size proportionally with the number of workers, but it leaves the convergence problem to users.

The other main challenge is *how to achieve an efficient state replication*. When we give more resources to an existing DL training job or migrate it to other resources, we need to replicate training states (e.g., model parameters, data loading states, etc) efficiently to these new resources. States can be very large, e.g., BERT [3] has more than 340 million parameters, which occupy more than 1GB memory. Moreover, states are usually stored in heterogeneous devices, e.g., model parameters are usually stored in GPU memory for fast computation while data loading states always reside in CPU memory. Due to the complex topology of underlying devices and the varied bandwidth of different links among devices, it is quite challenging to replicate states efficiently.

Previous work about state replication includes Parameter Server (PS) [14] and checkpoint [5], [6]. PS manages states in a set of centralized CPU servers while checkpoint dumps states to persistent storage. Such design simplifies the replication since all states can be obtained from one place. But they are inefficient because PS can suffer from the communication bottleneck in large-scale training [15] and checkpoint involves heavy-weight IO operations and CPU-GPU memory copy (§V-B).

B. Contributions

We design Elan, a generic and efficient elastic training system for DL, which targets the most popular training scheme, data-parallel distributed training with collective communication [15], [16], [17], and contains novel solutions to the above challenges. In summary, this paper makes the following contributions.

- To explore more parallelism when scaling out, we propose a *hybrid scaling* mechanism, which is a co-design of system and algorithm. We make a trade-off between training efficiency and model performance through adaptively choosing strong scaling or weak scaling. And we also apply a *progressive linear scaling* rule to keep model performance when performing weak scaling.
- To replicate training states efficiently, we propose a *concurrent IO-free replication* mechanism. Considering the varied bandwidth of different links among devices, we leverage the topology information to conduct multiple replications in parallel. We also utilize direct memory access among devices with Peer-to-Peer communication and among nodes with RDMA to avoid IO operations and CPU-GPU memory copy.
- For efficient elasticity implementation, we design an asynchronous coordination mechanism, which is used to coordinate all workers to perform a resource adjustment. As allowing the asynchronous start of new workers and avoiding the shutdown of existing workers, the mechanism hides the high overhead of start and initialization, achieving high-performance resource adjustments.

Moreover, we also provide data consistency and fault tolerance support in our system. To demonstrate the generality of Elan, we integrate it with Caffe [18] and Pytorch [16] and apply it to training different types of neural networks. Experimental results show that Elan achieves very high performance ($\sim$1s) on migration, scaling in and scaling out under multiple adjusting scales with different models. Elan also incurs negligible overhead ($<3‰$) when no resource adjustments are performed. With the support of Elan, we achieve 20% speedup for elastic training of ResNet-50 on ImageNet and show a huge advantage in elastic DL training job scheduling.

II. Overview

Elan is designed to use with a job scheduler in DL training clusters. Typically, there are only workers in data-parallel distributed training jobs with collective communication. Elan attaches an application master (AM) to each job. The AM offers a resource adjustment service to the scheduler and helps coordinate all workers to perform resource adjustments. As shown in Figure 2, the procedure of a resource adjustment using Elan consists of 5 steps. For the ease of understanding, all examples in this paper are about scaling out, while Elan supports scaling in and migration in the same way.

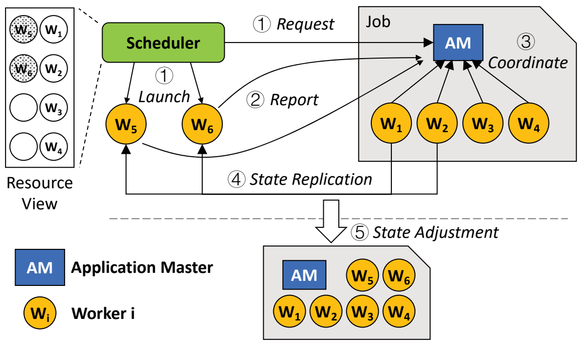


Fig. 2: The resource adjustment procedure of Elan.

① Request. The scheduler requests a resource adjustment (i.e. scaling in, scaling out or migration) via the service offered by the AM. It also launches new workers if any. For example, the scheduler in Figure 2 decides to add two new workers (W_5, W_6) into a 4-worker (W_{1-4}) job according to its resource view of the cluster, it requests the AM and launches the new workers.

② Report. After start and initialization, new workers report to the AM that they are ready to join the training.

③ Coordinate. Existing workers coordinate with the AM at intervals. During coordination, the AM can decide to perform a resource adjustment when all the new workers has reported. We detail the coordination mechanism in §V-B.

④ State Replication. In case of a resource adjustment, training states need to be replicated to new workers. We propose a concurrent IO-free replication mechanism for efficient state replication (§IV).

⑤ State Adjustment. Some states need to be adjusted to reflect the latest status. For example, we perform data

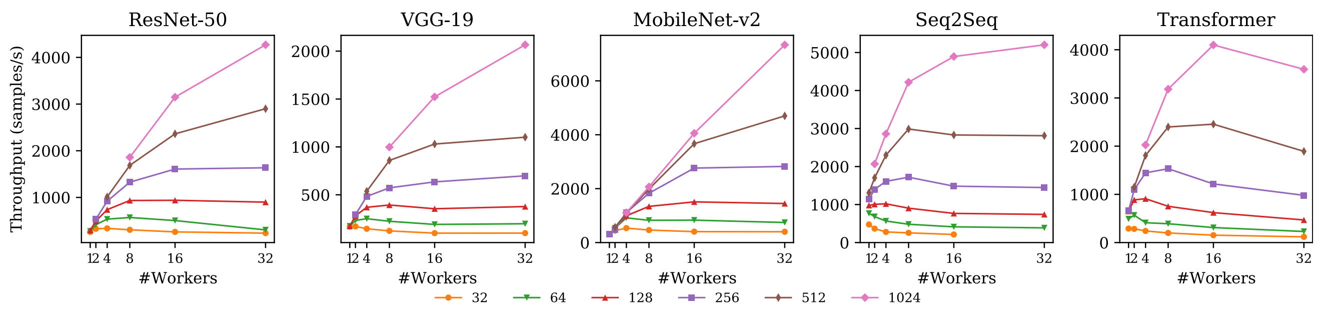


Fig. 3: Strong scaling. Executions on the same curve have the same total batch size.

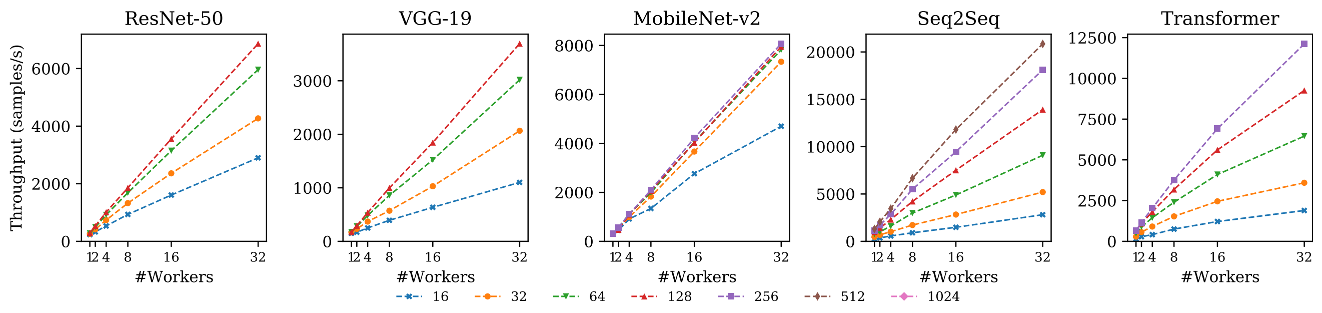


Fig. 4: Weak scaling. Executions on the same curve have the same batch size per worker.

repartition (§V-C) and communication group reconstruction after the adjustment. Most importantly, we propose a hybrid scaling mechanism to adjust the batch size and the learning rate to explore more parallelism when scaling out (§III).

The hybrid scaling mechanism and the concurrent IO-free replication mechanism are proposed to tackle the two main challenges. We elaborate them in the following two sections respectively.

III. Hybrid Scaling Mechanism

We address the challenge of *parallelism exploration* with a novel *hybrid scaling* mechanism, which is a co-design of system and algorithm. We first analyze two straight-forward strategies for scaling out data-parallel DL training jobs, i.e., strong and weak scaling, to gain some observations.

Strong scaling scales the number of workers with the *total batch size* fixed, while *weak scaling* scales the number of workers with the *batch size per worker* fixed. For example, assume there are 4 workers with a total batch size of 128, thus the batch size per worker is $128/4 = 32$. After adding 4 more workers, the batch size per worker will be $\mathbf{128}/8 = 16$ using strong scaling, while the total batch size will be $\mathbf{32} \times 8 = 256$ using weak scaling.

To analyze the characteristics of the two strategies, we study their effect from the view of both system and algorithm.

1) System View: Effect on Training Efficiency: We select 5 important models, which each is or was the state-of-the-art in the development of computer vision (CV) or natural language processing (NLP). These models are summarized in Table I. We evaluate them on servers with 8 NVIDIA Tesla V100 GPUs per server and use one worker per GPU. Figure 3 and Figure 4 present the training throughput with respect to the number of workers using strong and weak scaling respectively. There are two key observations. (We have evaluated many other models, and they present a similar pattern.)

Model	Type	Domain	#Parameters	Dataset
ResNet-50 [19]	CNN	CV	25M	ImageNet [20]
VGG-19 [21]	CNN	CV	143M	ImageNet
MobileNet-v2 [22]	CNN	CV	3M	ImageNet
Seq2Seq [23]	RNN	NLP	45M	Tatoeba [24]
Transformer [25]	-	NLP	47M	WMT'16 [26]

TABLE I: DL models for scaling out strategy analysis.

- First, training throughput increases linearly with the number of workers using weak scaling, but it increases and then decreases using strong scaling. In other words, weak scaling has a constant marginal gain on training efficiency while strong scaling has a diminishing one.
- Second, a larger batch size (i.e., a higher parallelism) with the same computation resource usually yields a higher training throughput[1], which is reflected in two aspects. On one hand, the optimal number of workers (that yields the maximum training throughput) using strong scaling tends to be larger when using a larger total batch size. On the other hand, the slope of weak scaling curves also increases with a larger batch size per worker.

The two observations are mainly the result of the mutual influence of both computation and communication. A larger batch size represents more computation, and more workers mean more communication and synchronization overhead. Strong scaling is associated with fixed total computation, when

[1]As discussed later, a larger batch size means more computation. This observation holds when the computation does not exceed the computing capacity of devices, which is true in most time in practice.

adding more workers, training will benefit at first since the computation per worker decreases, but it will suffer later when communication overhead becomes the dominance. Whereas training will always benefit from adding more workers using weak scaling since the total computation is increased.

To summarize, by only considering training efficiency, an elastic training system is supposed to prefer weak scaling than strong scaling.

2) Algorithm View: Effect on Model Performance: Weak scaling benefits from increased computation but also enlarges the total batch size at the same time, while the total batch size is an important hyperparameter in DL training. To analyze its effect on model performance, we show the results of training MobileNet-v2 on Cifar100 with different total batch sizes in Figure 5 (Default) for an example, but the effect is widely observed [11], [27]. We keep all other hyperparameters the same among different training.

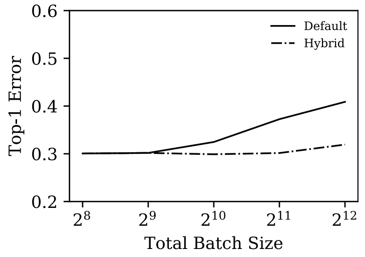


Fig. 5: The effect of total batch sizes on training MobileNet-v2 on Cifar100.

It is shown that, for a given model, the model performance tends to get worse with a larger total batch size. In other words, the benefit of weak scaling is built on the sacrifice of model performance. What's worse, to the best of our knowledge, the relation between total batch sizes and model performance has no theoretical analysis. In contrast, strong scaling does not change any hyperparameter including the total batch size, i.e., it scales in an algorithm-transparent way, which reveals that strong scaling has no influence on model performance.

To summarize, since model performance matters, we should be very careful if using weak scaling in elastic training.

3) Hybrid Scaling Mechanism: Based on the observations from the view of both system and algorithm, we can regard strong and weak scaling as two extremes of the trade-off between model performance and training efficiency. To make a good balance, we propose a *hybrid scaling* mechanism as shown in Figure 6.

First, for the parallelism (total batch size), we desire to find the *minimum* total batch size with which training will not suffer from resource under-utilization, i.e., the optimal number of workers using strong scaling with that total batch size is not smaller than the number of workers after the adjustment (Line 10). The *minimum* here means that we try to minimize the effect of batch size change. Therefore, when adding more workers, we try strong scaling first (Line 6), otherwise we perform weak scaling by increasing the total batch size by 2x each time (Line 13) and repeat until the requirement (Line

Algorithm 1: Hybrid Scaling Mechanism

```
Parallelism Scaling:
1  Procedure GetTotalBatchSize()
2      $N \leftarrow$ #Workers before the adjustment
3      $TBS \leftarrow$ Total batch size before the adjustment
4      $N' \leftarrow$ #Workers after the adjustment
5
6      $k \leftarrow 1$                    // Scaling factor
7      while $k \leq N'/N$ do
8          $TBS' \leftarrow k \times TBS$
9          $N_{opt} \leftarrow$ the optimal #workers using strong
             scaling with a total batch size of $TBS'$
10         if $N_{opt} \geq N'$ then
11             return $TBS'$
12         end
13         $k \leftarrow k \times 2$
14     end
15     $k \leftarrow N'/N$
16     return $TBS \times k$
17 end

Progressive Linear Scaling:
18 Procedure GetLearningRate()
19     $lr_0 \leftarrow$ Learning rate before the adjustment
20     $k \leftarrow$ Scaling factor
21     $T \leftarrow$ #Iterations for the learning rate adjustment
22     $T_0 \leftarrow$ The iteration that the adjustment begins
23     $t \leftarrow$ Current iteration
24
25     $lr_T \leftarrow lr_0 \times k$   // Target learning rate
26     if $t < T$ then
           // Progressive linear scaling
27         return $lr_0 + \frac{t-T_0}{T}(lr_T - lr_0)$
28     else
29         return $lr_T$
30     end
31 end
```

Fig. 6: Hybrid Scaling Mechanism

10) is satisfied. If all the trials fail, we simply apply the weak scaling strategy according to the resource change (Line 15).

Second, to deal with reduced model performance with large batch sizes, we propose a *progressive linear scaling* rule to adjust the learning rate.

According to the parameter update equation using Stochastic Gradient Descent [28] (Equation 1), we should scale the learning rate by k times when we scale the total batch size by k times.

$$w = w - \frac{\eta}{TBS}\nabla w = w - \frac{\eta \cdot k}{TBS \cdot k}\nabla w \quad (1)$$

where w is the parameter, ∇w is the gradient of w, η is the learning rate and TBS is the total batch size. But, a sharp change in the learning rate may lead the model to divergence [29], which reveals that we should finish the learning rate adjustment in a progressive way, e.g., after T iterations. Therefore the learning rate is given by

$$lr_T = lr_0 \times k \tag{2}$$

$$lr_t = \begin{cases} lr_0 + \frac{t-T_0}{T}\left(lr_T - lr_0\right) & t \in [T_0, T) \\ lr_T & t \geq T \end{cases} \tag{3}$$

where lr_t is the learning rate of iteration t, T_0 is the iteration that the adjustment begins, lr_0 is the learning rate before the adjustment and lr_T is the target learning rate.

Figure 5 (Hybrid) shows the effectiveness of this rule where model performance holds with large batch sizes. But it is also shown that model performance still goes down when the total batch size is too large (2^{12}) even using the progressive linear scaling rule. We can only try our best to alleviate this problem because convergence with large batch sizes is still an open problem in research. But we find that the hybrid scaling mechanism is sufficient in practice because the optimal number of workers will be quite large after weakly scaling several times (Line 13 in Figure 6), which means that we will first run out of resources (or resource quota) before the model diverges. We have also demonstrated the effectiveness of the hybrid scaling mechanism in the elastic training (§VI-B) and the elastic scheduling (§VI-C) experiments.

IV. Concurrent IO-free State Replication

In Elan, we propose a concurrent IO-free replication mechanism to make the state replication efficient. This mechanism corresponds to ④ State Replication in the adjustment procedure (§II). In this section, we first summarize the characteristics of training states and study the bandwidth variation of different links among devices to motivate our design.

1) Characteristics of Training States: Data-parallel DL training is a *stateful* iterative optimization process. As shown in Figure 7, it first *loads* `data`, performs *forward* and *backward* propagation using the `model` parameters, aggregates gradients by *allreduce* communication among a `communication group` and ends with *updating* the model parameters using an `optimizer`.

`Data (loading)`, `model`, `communication group` and `optimizer`, together with some runtime information (e.g. the current `epoch` and `iteration`), compose the whole training states of a data-parallel DL training job. We summarize their characteristics in Table II. It is shown that (1) states reside in heterogeneous devices and (2) GPU states are much larger than CPU states. These two observations require that our mechanism must (1) manage the heterogeneity of states well and (2) provide efficient replication for GPU states. Moreover, due to data-parallel scheme, every existing worker has one identical copy of the whole training states, which we can exploit to further optimize the replication.

2) Bandwidth Variation of Links: There are three ways to communicate between PCIe interconnected GPUs, i.e, via Peer-to-Peer memory access (P2P), using CPU shared memory as a bridge (SHM) and through network (NET; 56Gbps Infiniband in our case). Figure 8 compares the bandwidth of the

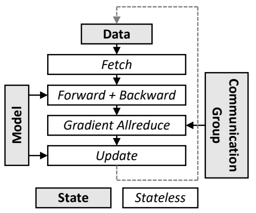


Fig. 7: Data-parallel DL training iterations.

State	Device	Typical Size
Data loading	CPU	< 1KB
Communication group	CPU	
Optimizer	CPU	
Runtime Information	CPU	
Model	GPU	10MB - 1GB

TABLE II: Characteristics of training states.

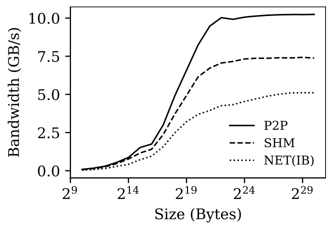


Fig. 8: Measured bandwidth of different links. NET refers to 56Gbps Infiniband (IB).

three ways, where P2P performs the best followed by SHM, and the worst is NET. To replicate GPU states efficiently, we must make full use of higher-bandwidth ways.

However, choosing which way to communicate depends on the topology of GPUs. Given two GPUs in a cluster, there are 4 typical levels of links between them (See Figure 9 for an example). *L1*: The link traverses only PCIe switches. *L2*: The link traverses a PCIe host bridge. *L3*: The link traverses a socket-level link (e.g., QPI). *L4*: The link traverses network. For communication, P2P is only enabled on *L1*; *L2*, *L3* can use SHM; and NET is the only way for *L4*.

3) Concurrent IO-free Replication: Motivated by the above observations, we propose a concurrent IO-free state replication mechanism.

First, we utilize topology information to perform efficient IO-free replication. We construct a topology tree of all workers, and select the *nearest* neighbor in the existing workers to replicate states. *Nearest* means that we try to use the way of as high bandwidth as possible (P2P > SHM > NET).

Second, we perform multiple replications concurrently. We select one neighbor for each new worker rather than one for them all. Therefore, replications between different worker pairs can proceed concurrently to increase the bandwidth utilization. However, when multiple replications incur contention due to physical link constraints (typically when replications traverse *L3*), we instead perform them in turn.

Figure 9 gives an example where 2 new workers (E, F) are added into a 4-worker (A, B, C, D) training job. A, B are connected to the same PCIe switch while C is on the other socket of the same node, and D resides in a different node. Since newly added worker E is most close to C under the same socket and F is most close to D under the same node, we perform two parallel replications simultaneously to maximize the bandwidth utilization.

Furthermore, we overlap the replications of states on heterogeneous devices. We replicate CPU states along with GPU

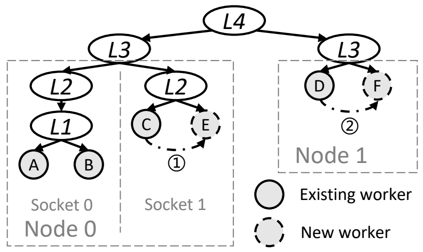


Fig. 9: An example of adding 2 new workers (E, F) into a 4-worker (A, B, C, D) training job.

Category	API	Description
Service	`Request-Adjustment`	The scheduler requests resource adjustments (migrate, scale in or scale out).
Resource Adjustment	`Report`	New workers report their existence to the AM.
	`Coordinate`	Existing workers coordinate with the AM to perform resource adjustments if any.
State Management	`RegisterState`	Register states that need to be replicated.
	`RegisterHook`	Register hooks for state adjustments.

TABLE III: The core APIs of Elan.

states simultaneously. Since CPU states are quite small, the replication of CPU states can be well overlapped even we use web socket to replicate them.

V. Implementation

In this section, we first present the APIs of Elan. Then we detail the asynchronous coordination mechanism in ③ Coordinate of the adjustment procedure (§II). We also discuss the data consistency and fault tolerance support in Elan.

A. API

The core APIs of Elan are listed in Table III. Service API is offered to the scheduler to request resource adjustments in ① Request. New workers report their existence to the AM in ② Report via `Report`. `Coordinate` triggers the coordination mechanism in ③ Coordinate. The states to replicate in ④ State Replication are registered by `RegisterState`, and state adjustment procedures in ⑤ State Adjustment are encapsulated in hook functions registered via `RegisterHook`. With the unified hook API, integration with a new framework simply requires implementing some hook functions.

We implement Elan using C++ and integrate it with Caffe [18] and Pytorch [16] as a demonstration of generality since Caffe has a static execution engine while Pytorch has a dynamic one.

B. Asynchronous Coordination Mechanism

The coordination mechanism is used to coordinate all existing workers with the AM to perform a resource adjustment if any. We motivate our design by analyzing a widely used approach Shutdown-Restart (S&R) [5], [6]. Figure 10 shows the timeline of scaling out using S&R. It firstly checkpoints all training states and shuts down existing workers, then the

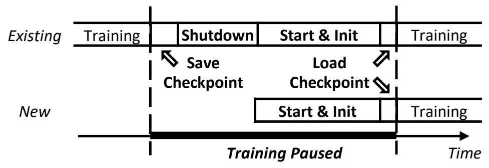


Fig. 10: Timeline of scaling out using S&R.

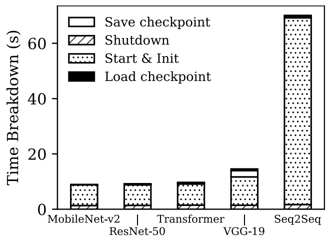


Fig. 11: Breakdown of scaling out from 4 to 8 workers using S&R.

job restarts with new resource configuration and loads the checkpoint.

Figure 11 presents the time breakdown of each phase, which shows that it is the long time of start and initialization that leads to the inefficiency of S&R. However, initialization is necessary to prepare runtime context, high-performance libraries and hardware accelerators, which motivates us to hide such overhead in our design.

To achieve this goal, we introduce an asynchronous coordination mechanism, which is triggered by the `Coordinate` API at the end of training iterations. The mechanism avoids the overhead due to two important features.

First, the coordination mechanism allows *asynchronous* start of new workers. As shown in Figure 12, new workers start and initialize in parallel with training on existing workers. The resource adjustment happens if and only if all the new workers have reported. In other words, if some of the new workers do not report when coordinating, the existing workers will not wait for them but proceed, and the adjustment is left for future coordination. In this way, we also hide the variance of start and initialization.

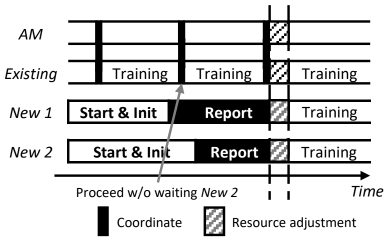


Fig. 12: Timeline of scaling out using Elan.

Second, the coordination mechanism provides *shutdown-free* elasticity. Suppose that resource adjustments involve shutdown and restart of existing workers (as S&R), then the hide of the start of new workers will be in vain because the shutdown and restart is on the critical path. In our design, no shutdown of existing workers is required, and they can continue training immediately after the adjustment. Together with the asynchronous start of new workers, elasticity is achieved gracefully without pausing training for a long time.

Furthermore, the frequency of coordination is *configurable*. It can be performed every *few* iterations, which gives users the flexibility to make a trade-off between elasticity and training efficiency.

C. Data Consistency

After resource adjustments, data require repartition to ensure data consistency. We propose a *serial data loading* semantics where workers fetch data in a global serial manner, thus the remaining data are always continuous. In contrast, chunk-based data loading, which is widely used in modern DL frameworks, partitions all data into chunks, and the remaining data are fragmented during training. These two semantics are compared in Figure 13. The advantage of the serial semantics over the chunk-based one is the high efficiency of management. With the serial semantics, data loading states can be represented as a single integer, i.e., the position of the start of the remaining data, while the chunk-based semantics may need a record table and a complex management logic.

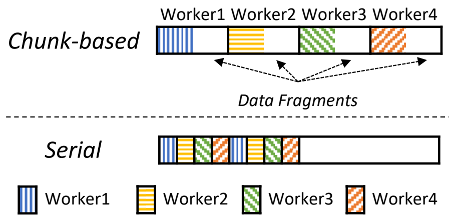


Fig. 13: Data status during training using the chunk-based semantics and the serial data loading semantics.

D. Fault Tolerance

The application master is a single point of failure in Elan. Thus, we apply a fault-tolerant design to improve availability. First, we design the application master as a state machine and save the state machine on distributed storage. For example, we deploy Elan in a Kubernetes [30] cluster, so we save it on `etcd`. Second, we tag every message with a unique ID and resend it in case of timeout. We also use `ZeroMQ` [31] for socket implementation, which can reconnect if any party of communication restarts.

VI. Evaluation

We evaluate Elan from three aspects. We first compare its runtime overhead and resource adjustment performance with two other baselines. Then we present the benefit of using Elan in two scenarios, i.e., elastic training with dynamic batch sizes and elastic DL training job scheduling. (All error bars in the figures of this section represent the standard deviation.)

A. Runtime Overhead and Resource Adjustment Performance

Testbed To report the real performance in production, we evaluate Elan in a production cluster of Sensetime company. We use up to 8 servers where each server has 2 20-core Intel Silver 4114 CPUs and 8 GeForce 1080Ti GPUs. All the servers share a Lustre file system and are connected with a 56Gbps Infiniband network and a 1,000Mb/s Ethernet. We use Pytorch 1.3.0 as the DL framework.

Baselines We compare with two state-of-the-art baselines.

- Shutdown-restart (S&R) is the most common practice [5], [6]. We use the `Coordinate` API to coordinate existing workers and check if there is a resource adjustment. If any, we checkpoint all training states, shut down existing workers, and then restart with the checkpoint. Note that S&R cannot fully exploit the asynchronous feature of the coordination mechanism because the shutdown and restart of existing workers is on the critical path.
- Litz [32] is a representative of the new programming model based elastic training systems (§VII). According to its paper, we perform context switches between different executors on shared GPU workers. We also implement local gradient aggregation for better performance. Since the number of executors on each worker is crucial to training efficiency, we use two variants, 2 executors on each worker (Litz-2) and 4 executors on each (Litz-4).

1) Runtime Overhead: We define runtime overhead as the wasted time on elasticity maintenance when training without resource adjustments. For Elan, the overhead comes from the asynchronous coordination mechanism. For S&R, the overhead comes from the coordination of existing workers triggered by the `Coordinate` API. Though these two systems are different, the runtime overhead is the same because both systems perform the same coordination in case of no adjustments. Thus, we only present the overhead of Elan here. For Litz, the overhead comes from context switches, but it is difficult to define the wasted time on context switches, instead we measure the training throughput as an indication of the runtime overhead. We repeat all the experiments 5 times.

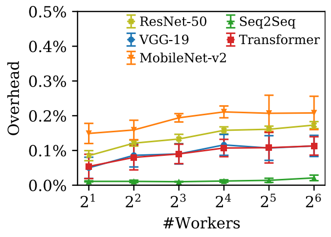
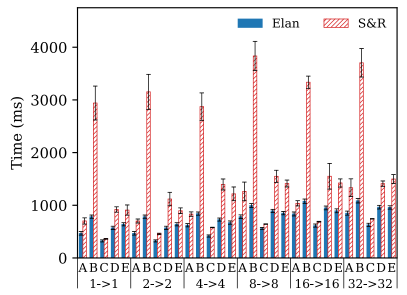
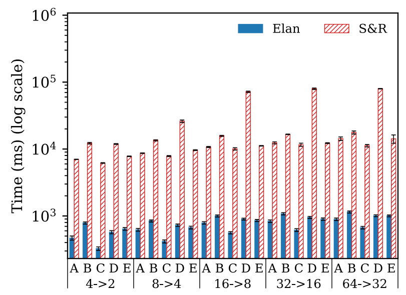


Fig. 14: Runtime overhead of Elan when training 5 different models on 2 - 64 workers.

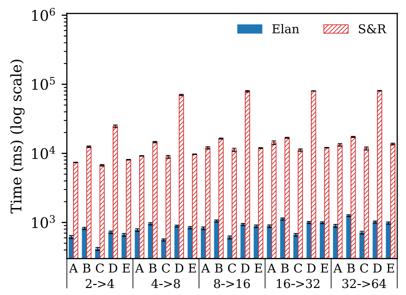
Figure 14 shows the runtime overhead of Elan (as well as S&R) when training 5 typical models (summarized in Table I) on 2 - 64 workers. It is shown that the overhead is always negligible (<3‰) regardless of the model and the number of workers. This is because there is only a coordination in case of no adjustments. The variance of runtime overhead on different models is due to the difference of iteration time, and the overhead tends to be higher with more workers because coordination is a form of synchronization, which usually takes a longer time with more participants. However, recall the coordination mechanism is designed to be configurable, we can further reduce the frequency to lower the overhead if we



(a) Migration

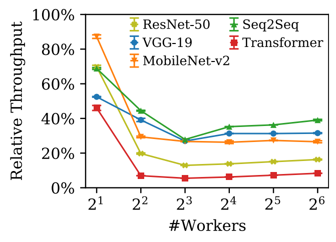


(b) Scaling in

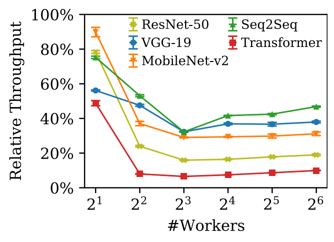


(c) Scaling out

Fig. 15: The performance of migration, scaling in and scaling out using Elan and S&R. `M->N` denotes migrating/scaling from `M` workers to `N` workers. Models are denoted by A - E. A: ResNet-50, B: VGG-19, C: MobileNet-v2, D: Seq2seq, E: Transformer.



(a) Litz-2



(b) Litz-4

Fig. 16: Relative training throughput of Litz versus Elan.

find it too high.

Figure 16 shows the relative training throughput of Litz-2 and Lit-4 compared with Elan. Because GPU memory is limited, when performing context switches, the context on GPU has to be moved out to CPU, and another context on CPU is moved into GPU. Frequent CPU-GPU memory movement damages training efficiency significantly. For example, although Litz-4 performs more computation than Litz-2, it still cannot match the efficiency of Elan. The reduction of throughput even exceeds 90% on Transformer. With more workers involved, the training throughput goes up slightly because of the benefit from local gradient aggregation, but it still suffers a lot from context switches.

To summarize, Elan has negligible runtime overhead and outperforms a lot than the new programming model based elastic training system, Litz.

2) Performance of migration and scaling in/out: We use the time spent on migration or scaling as a measure of the resource adjustment performance. Since Litz incurs too much runtime overhead, we focus on the comparison with S&R in this section. For each comparison, we repeat the experiment 5 times.

Figure 15 compares the resource adjustment performance of Elan and S&R in all the three cases. First, Elan achieves high performance ($\sim$1s) on migration, scaling in and scaling out on all the scales and models in the experiment. Second, Elan can perform up to $4\times$ faster on migration and 10$\sim$80x faster on scaling in or out than S&R.

We explain the huge improvement as follows. For migration, S&R can benefit from the asynchronous feature of our coordination mechanism because existing workers are discarded after migration, thus the main advantage of Elan over S&R is that Elan utilizes the efficient state replication mechanism to avoid IO operations and CPU-GPU memory copy. For scaling in or out, besides the advantage of state replication, the improvement mainly comes from the asynchronous coordination mechanism, which hides the high overhead of start and initialization. However, for S&R, since the shutdown and restart of existing workers is on its critical path, it cannot benefit from the asynchronous feature anymore.

To summarize, Elan achieves high performance on migration, scaling in and scaling out, which outperforms S&R by up to 80x.

B. Elastic Training

In this section, we present the benefit of using Elan in the elastic training scenario via an elastic training experiment with dynamic batch sizes. This experiment also demonstrates the effectiveness of the hybrid scaling mechanism in §III. We use the same testbed as in the previous section.

Workload AdaBatch [7] is a representative of elastic training with dynamic batch sizes. The core idea of AdaBatch is to train neural networks with a small batch size at the beginning and then double the batch size at intervals. We adapt AdaBatch to train ResNet-50 on Imagenet that we begin with a batch size of 512, double it per 30 epochs and finish training after 90 epochs. According to the hybrid scaling mechanism, we also double the learning rate when doubling the batch size, and we finish the learning rate adjustment in 100 iterations. All the other hyperparameters are obtained from the official scripts of Pytorch.

Configurations There are three batch sizes of 512, 1024 and 2048 during the elastic training. Guided by the strong scaling curves shown in Figure 17, we use 16 workers when the batch size is 512, 32 workers with 1024 batch size and 64 workers with 2048 batch size. We denote this configuration by 512-2048 (Elastic).

We compare it with two baselines. 512 (16) denotes training with a fixed batch size of 512 on 16 workers. This is used as the baseline of accuracy and static training. 512-2048 (64) denotes training with dynamic batch sizes but on fixed 64 workers. This is used to show the necessity of Elan.

Results Figure 18 compares the top-1 accuracy of static and elastic training. 512 (16) achieves an accuracy of 75.89%

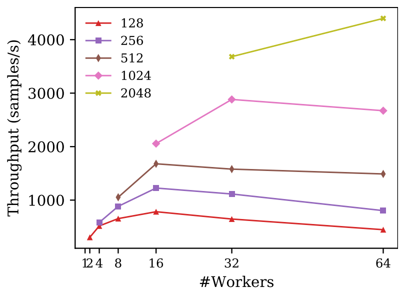


Fig. 17: Strong scaling of training ResNet-50 on Imagenet with different batch sizes.

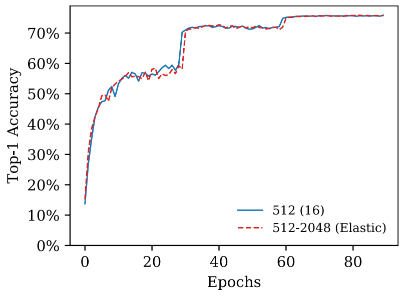


Fig. 18: Training accuracy of static and elastic training.

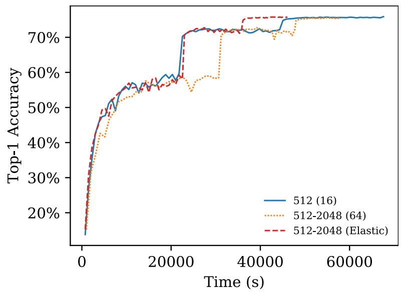


Fig. 19: Training efficiency of the three configurations.

Target Accuracy	Time to solution (s) 512 (16)	512-2048 (64)	512-2048 (Elastic)	Speedup
74.5%	45073.52	48021.25	36109.18	19.89%
75%	45824.74	48708.44	36452.78	20.45%
75.5%	48829.64	51800.76	38170.73	21.83%

TABLE IV: Time to solution with the three configurations and speedup of 512-2048 (Elastic) versus 512 (16).

and 512-2048 (Elastic) achieves 75.87%, which demonstrates that the hybrid scaling mechanism is effective to keep model performance.

Figure 19 shows the training efficiency using different configurations, and Table IV gives the time to solution and speedup. First, it is shown that 512-2048 (Elastic) achieves a 20% speedup versus 512 (16) under three settings of target accuracy, and elastic training tends to give a higher speedup for a higher target accuracy. Second, it is interesting that training with dynamic batch sizes but on fixed resources (512-2048 (64)) is hard to obtain a speedup because resources are under-utilized with small batch sizes, which reveals that resource elasticity provided by Elan is necessary for elastic training.

To summarize, Elan provides a 20% speedup for elastic training of ResNet-50 on Imagenet and the hybrid scaling mechanism succeeds to keep model performance. Considering the trend that models and training scales is getting larger, Elan will make a bigger difference in the future.

C. *Elastic Scheduling*

Elasticity can benefit DL training job scheduling in many aspects. For example, local training clusters can exploit elasticity to provide preemption, migration and over-subscription to improve resource utilization. In cloud, elasticity can be leveraged to utilize transient resources such as spot instances. In this section, we demonstrate the benefit of using Elan on elastic scheduling in a local training cluster.

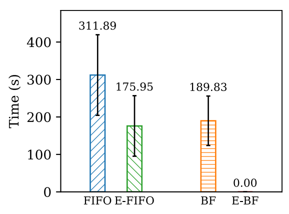
Trace We collect a real-world trace from a DL training cluster in Sensetime company (The GPU resource utilization of one week in this trace is shown in Figure 1). However, it is hard to reproduce the trace because the time span is too long and the code is not available to us. Therefore, we select and down-sample a two-day trace to make it finish within 10 hours. For each job, we randomly select one configuration from Table I. We also downscale the number of total GPUs to 128 to match the scale of jobs. The resource configuration of each job in the trace is static (each job specifies the number of its requested workers `req_res`), to enable elastic scheduling, we also specify `min_res` and `max_res` for them. Precisely, we ensure that the model can fit in GPU memory with `min_res` workers and converge with `max_res` workers.

Simulation We build a discrete-time simulator to execute the trace. The statistics for simulation are collected from real runs in the DL training cluster, which mainly include the training throughput under different resource and model configurations, the runtime overhead and the resource adjustment performance of Elan and S&R.

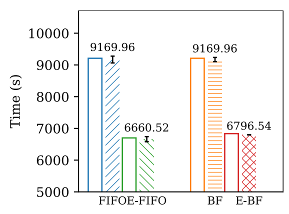
Baselines We use two popular static scheduling policies as baselines, FIFO and Backfill (BF) [33]. FIFO obeys a first-in-first-out rule and BF is the default scheduling policy of Slurm [34], which is based on FIFO but allows low priority jobs to start earlier if doing so will not delay high priority jobs to start.

Elastic Scheduling Policy We propose a simple policy here to demonstrate the benefit of elasticity, which includes (1) an *admission rule*: a submitted job can start if the `min_res` of the job is less than the current free GPUs or adding the job will not increase the average job completion time; (2) an *allocation rule*: this rule determines resource allocation among all running jobs. First, we allocate `min_res` workers to each job. Then, we calculate for each job the marginal gain of adding one worker to it, and then add one worker to the job with the maximum marginal gain. We repeat this step until all resources are allocated, or the number of workers of each job exceeds its `max_res`, or the marginal gain of each job is not positive. Readers can refer to [6] for the definition of the marginal gain. Besides, we follow our hybrid scaling mechanism to adjust the batch size and learning rate in case of resource adjustments. We apply the above policy to the two baselines and denote them by Elastic-FIFO (E-FIFO) and Elastic-Backfill (E-BF). A more complicated scheduling policy is out of the scope of this paper, we leave it for future work.

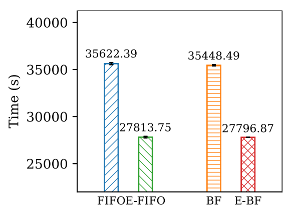
1) The Benefit of Elasticity on Scheduling: We run the simulation 3 times and summarize the statistics of job pending time (JPT), job completion time (JCT) and makespan in Figure 20. Note that we do not aim to compare the effectiveness of FIFO and BF, instead we focus on the comparison between the static policy and its elastic variant.



(a) Average JPT



(b) Average JCT



(c) Average makespan

Fig. 20: Metrics of scheduling using FIFO, E-FIFO, BF and E-BF policy.

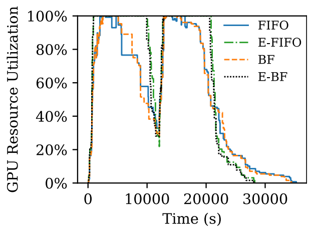
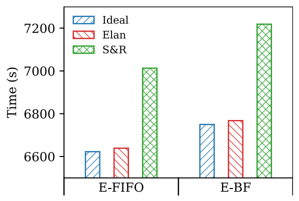
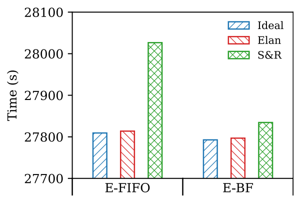


Fig. 21: GPU Resource utilization of one run.



(a) Average JCT



(b) Average makespan

Fig. 22: Metrics of scheduling using different elastic systems.

It is shown that JPT, JCT and makespan (an indication of resource utilization) are all reduced significantly with elasticity (JPT is reduced by 43%+, JCT is reduced by 25%+, and makespan is reduced by 21%+). This reduction is a comprehensive result of multiple factors. First, elastic policies permit a job starting with `min_res` workers, while static policies can only wait for `req_res` workers. Second, resources are dynamically allocated across all running jobs to optimize the average JCT, while static policies cannot adjust the resource allocation once determined. Third, when the cluster is under-utilized, running jobs can scale out to utilize the available resources under elastic policies.

Figure 21 shows the GPU resource utilization of one run in detail. It is shown that scheduling with elasticity can achieve better resource utilization and deal with the workload fluctuation better.

2) The Necessity of High-performance Elasticity: Not all elastic systems can enjoy the benefit of elastic scheduling. The improvement is related to the runtime overhead and the resource adjustment performance. Figure 22 shows the average JCT and makespan using different systems. It is shown that Elan achieves a similar performance to the ideal system (with no runtime overhead and instantaneous resource adjustments), while the JCT is increased by 6% using S&R. Since the time span of the simulation trace is limited, we believe there will be a larger gap between these two systems in real-world cases.

To summarize, we show the advantage and necessity of Elan in DL training job scheduling, which reduces JPT and JCT and improves resource utilization by a large margin.

VII. Related Work

The Use of Elasticity Typical use of elasticity includes elastic DL training, straggler mitigation and elastic DL training job scheduling. Elastic DL training involves algorithm innovations such as dynamic batch sizes [7], [8], [35], elasticity-aware SGD [9], [10], etc. They explore elasticity requirements and potential from training itself. In contrast, straggler mitigation [12], [13], [36] and elastic DL training job scheduling [5], [6], [37] exploit resource changes and try to adapt training to maximize resource utilization. All the above innovations can be supported naturally and efficiently by Elan via its high-performance elasticity. We have shown the advantage of Elan in two of these scenarios.

Scaling DL Training Scaling DL training is quite hard due to its poor convergence with large batch sizes. *Learning rate adjustment* [38] and *warmup scheme* [39] are two popular solutions to the convergence problem. Our hybrid scaling mechanism can be seen as an innovative extension of these two solutions in the elastic training scenario. The difference is that we adjust *both* the learning rate and the batch size *with the change of resources*, while they *only* adjust the learning rate *manually*.

Elastic Training Systems Most elasticity-aware DL training job schedulers are based on S&R such as Gandiva [5] and Optimus [6], or PS such as DL^2 [37]. As we evaluate, S&R is heavy-weight and has a negative effect on scheduling. And PS suffers from the communication bottleneck in large-scale training. Instead, Elan is powered by the asynchronous coordination mechanism and the concurrent IO-free replication mechanism to minimize the resource adjustment overhead. And Elan is based on collective communication to provide high-performance distributed training.

Litz [32], Cruise [40] and Pytorch-Elastic [41] are new programming model based elastic training systems. Litz and Cruise are both based on PS, and Pytorch-Elastic uses checkpoint to replicate training states. PaddlePaddle EDL [42] and ElasticDL [43] also use checkpoint, and they are specific to certain DL frameworks. Instead, Elan exploits topology-aware direct memory access to avoid IO operations, and Elan is generic to various DL frameworks.

VIII. Conclusion

This paper proposes Elan, a generic and efficient elastic training system for data-parallel DL training with collective communication. We propose a hybrid scaling mechanism to explore more parallelism when scaling out, which makes a good trade-off between training efficiency and model performance. For efficient state replication, we exploit the topology information to perform concurrent and IO-free replications on heterogeneous devices. We also propose an asynchronous coordination mechanism to avoid the high overhead of start and initialization. Evaluation shows that Elan achieves high-performance migration, scaling in and scaling out with negligible runtime overhead and shows a huge advantage in elastic training and elastic DL training job scheduling scenarios.

Acknowledgement

We thank the anonymous reviewers for their valuable comments and suggestions. This work is partially supported by the National Key R&D Program of China (Grant No. 2017YFB1003103), Beijing Natural Science Foundation (4202031), Beijing Academy of Artificial Intelligence (BAAI), Tsinghua University Initiative Scientific Research Program (20191080594). Jidong Zhai is the corresponding author of this paper.

References

[1] Y. Sun, X. Wang, and X. Tang, "Deep learning face representation from predicting 10,000 classes," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1891–1898.

[2] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[4] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," *arXiv preprint arXiv:1901.05758*, 2019.

[5] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[6] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.

[7] A. Devarakonda, M. Naumov, and M. Garland, "Adabatch: adaptive batch sizes for training deep neural networks," *arXiv preprint arXiv:1712.02029*, 2017.

[8] N. Mu, Z. Yao, A. Gholami, K. Keutzer, and M. Mahoney, "Parameter re-initialization through cyclical batch size schedules," *arXiv preprint arXiv:1812.01216*, 2018.

[9] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li, "Dynamic mini-batch sgd for elastic distributed training: Learning in the limbo of resources," *arXiv preprint arXiv:1904.12043*, 2019.

[10] S. Narayanamurthy, M. Weimer, D. Mahajan, T. Condie, S. Sellamanickam, and S. S. Keerthi, "Towards resource-elastic machine learning," in *NIPS 2013 BigLearn Workshop*, vol. 1, no. 2.1, 2013, pp. 2–3.

[11] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[12] Q. Zhou, K. Wang, S. Guo, H. Lu, L. Li, M. Guo, and Y. Sun, "Falcon: Towards computation-parallel deep learning in heterogeneous parameter server," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 196–206.

[13] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Fast distributed deep learning via worker-adaptive batch sizing," *arXiv preprint arXiv:1806.02508*, 2018.

[14] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[15] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[17] NVIDIA, "Nvidia collective communications library (nccl)," 2019. [Online]. Available: https://developer.nvidia.com/nccl

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR09*, 2009.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[23] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[24] T. Ho and A. Simon, "Tatoeba: Collection of sentences and translations," 2019. [Online]. Available: https://tatoeba.org

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[26] D. Elliott, S. Frank, K. Sima'an, and L. Specia, "Multi30k: Multilingual english-german image descriptions," *arXiv preprint arXiv:1605.00459*, 2016.

[27] Y. You, J. Hseu, C. Ying, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large-batch training for lstm and beyond," *arXiv preprint arXiv:1901.08256*, 2019.

[28] D. Saad, "Online algorithms and stochastic approximations," *Online Learning*, vol. 5, 1998.

[29] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.

[30] "Kubernetes," 2019. [Online]. Available: https://kubernetes.io

[31] "Zeromq," 2019. [Online]. Available: https://zeromq.org

[32] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing, "Litz: Elastic framework for high performance distributed machine learning," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 631–644.

[33] S. Leonenkov and S. Zhumatiy, "Introducing new backfill-based scheduler for slurm resource manager," *Procedia Computer Science*, vol. 66, pp. 661–669, 2015.

[34] "Slurm workload manager," 2019. [Online]. Available: https://slurm.schedmd.com

[35] H. Yu and R. Jin, "On the computation and communication complexity of parallel sgd with dynamic batch sizes for stochastic non-convex optimization," *arXiv preprint arXiv:1905.04346*, 2019.

[36] 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 *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 98–111.

[37] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "Dl2: A deep learning-driven scheduler for deep learning clusters," *arXiv preprint arXiv:1909.06040*, 2019.

[38] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[39] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[40] W.-Y. Lee, Y. Lee, J. S. Jeong, G.-I. Yu, J. Y. Kim, H. J. Park, B. Jeon, W. Song, G. Kim, M. Weimer *et al.*, "Automating system configuration of distributed machine learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 2057–2067.

[41] "Pytorch elastic," 2019. [Online]. Available: https://github.com/pytorch/elastic

[42] "Paddlepaddle edl: Elastic deep learning," 2019. [Online]. Available: https://github.com/PaddlePaddle/edl

[43] "Elasticdl: A kubernetes-native deep learning framework," 2019. [Online]. Available: https://github.com/sql-machine-learning/elasticdl