ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning

Diandian Gu
Peking University
China

Yihao Zhao
Peking University
China

Yinmin Zhong
Peking University
China

Yifan Xiong
Microsoft Research
China

Zhenhua Han
Microsoft Research
China

Peng Cheng
Microsoft Research
China

Fan Yang
Microsoft Research
China

Gang Huang
Peking University
China

Xin Jin
Peking University
China

Xuanzhe Liu
Peking University
China

ABSTRACT

This paper proposes ElasticFlow, an elastic serverless training platform for distributed deep learning. ElasticFlow provides a serverless interface with two distinct features: (*i*) users specify only the deep neural network (DNN) model and hyperparameters for a job, but not the number of GPUs; (*ii*) users specify the deadline for a job, but not the amount of time to occupy GPUs. In contrast to existing server-centric platforms, ElasticFlow provides *performance guarantees* in terms of meeting deadlines while alleviating tedious, low-level, and manual resource management for deep learning developers.

The characteristics of distributed training introduce two challenges. First, the training throughput scales *non-linearly* with the number of GPUs. Second, the scaling efficiency is affected by worker *placement*. To address these challenges, we propose *Minimum Satisfactory Share* to capture the resource usage of training jobs to meet deadlines, and ElasticFlow performs admission control based on it. We develop a greedy algorithm that dynamically allocates resources to admitted jobs based on diminishing returns. We apply buddy allocation to worker placement to eliminate the effect of topology. Evaluation results on a cluster of 128 GPUs show that ElasticFlow increases the number of jobs that can meet their deadlines by 1.46–7.65× compared to existing solutions.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**; • **Computer systems organization** → **Cloud computing**.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575721

KEYWORDS

Distributed Deep Learning, GPU Cluster, Serverless Computing, Cluster Scheduling

ACM Reference Format:
Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575721

1 INTRODUCTION

Deep learning (DL) powers many applications and services we use every day. Training DL models is an important workload in datacenters. Most of the current efforts for DL training follow the *server-centric* style [12, 19, 21, 63], where DL developers request hardware resources in the form of machine instances (physical machines, virtual machines, or containers) to run DL training jobs. Although the server-centric approach substantially advances DL applications, it has two limitations.

First, the server-centric model is too *low level* for DL developers, who need to explicitly request hardware resources and configure machines to run their jobs. Also, DL developers face a *system-wide* problem of adapting the local batch size based on GPU memory and deciding the number of workers, as well as a *DL* problem of choosing the hyperparameters when training deep neural network (DNN) models. This is particularly challenging for DL developers who are likely to have limited expertise in systems. Second, the server-centric model does not have the flexibility to elastically scale the resource provisioning of DL jobs to promise *performance guarantees*, i.e., guaranteeing to finish a job before a given deadline. Performance guarantees are critical for production environments that require models to be (re-)trained and onboarded in time for regular product releases [19, 38], e.g., fine-tuning the BERT model with daily news to update recommendation services every day.

While some DL training platforms [42, 52] are not server-centric (i.e., DL developers do not need to configure the system-wide configurations), they are not aware of DL developers' performance requirements.

Our proposal. A good platform for DL training should alleviate low-level resource management for DL developers while providing guarantees for their requirements. Therefore, we propose ElasticFlow, an elastic serverless training platform for distributed DL. Compared to existing DL platforms [12, 19, 21, 42, 49, 52, 63], the key difference of ElasticFlow is that it exposes a high-level *deadline-driven* serverless interface to DL developers with two distinct features: (*i*) DL developers specify *only* the to-train DNN model and hyperparameters for a job, without the need to explicitly declare the required number of GPUs; (*ii*) DL developers specify the desired deadline for a job, but not the exact amount of time to occupy GPUs. This deadline-driven serverless interface decouples the DL problem from the system management problem. The benefit of this design is that it alleviates low-level tedious resource management tasks for DL developers and allows them to focus on model development in a *low-code* fashion. Importantly, by decoupling static resource configuration from the DL training code, ElasticFlow is able to leverage elastic scaling to dynamically allocate resources for every single job and provide *performance guarantees* to meet their deadlines.

Challenges. Although serverless interfaces can provide possible design space, the characteristics of DL jobs introduce two challenges to elastic resource allocation while guaranteeing given deadlines. First, the throughput of a DL job scales *non-linearly* along with the number of GPUs. The reason is that the communication overhead among workers increases with the number of workers. Second, the scaling efficiency is affected by the *placement* of workers [63]. The workers within the same server can leverage the high-bandwidth NVLink or PCIe to communicate among GPUs, while the bandwidths across servers are usually at a lower level. Thus, the placement of workers changes the scaling curve of a job. As ElasticFlow needs to consider a set of non-linear scaling curves for each job, these two challenges intertwine with each other, which further complicates resource allocation.

Key techniques. To maximize resource efficiency under non-linear scaling curves, we propose *Minimum Satisfactory Share*. It is known that the scaling curves of DL jobs are *concave*, which means adding resources to a distributed job may have *diminishing returns*. We use the minimum satisfactory share to capture the minimum resource allocation a job needs to meet its deadline. The key techniques of ElasticFlow include an admission control module that decides whether to admit an arriving job and a resource allocation module that dynamically allocates resources to guarantee the jobs' deadlines. For each admitted job, the resource allocation module allocates at least the minimum satisfactory share for each job to guarantee their deadlines. For the remaining resources, the module uses a greedy algorithm to prioritize resource allocation for the most efficient jobs based on the diminishing returns of their scaling curves. We prove that the algorithm is optimal under concave scaling curves. We also describe the extensions of ElasticFlow to accommodate best-effort jobs (i.e., jobs without deadlines).

We apply buddy allocation to address the challenge of topology-dependent placements, which enables ElasticFlow to *decouple* job placement from admission control and resource allocation. Where jobs can be migrated and the number of workers is restricted to a power of two, buddy allocation guarantees to eliminate fragmentation, i.e., a job can always find a set of GPUs that are "close" to each other in topology as long as the number of idle GPUs is no smaller than that needed by the job.

Contributions. Our contributions are as follows.

- We propose ElasticFlow, an elastic serverless computing platform for distributed DL training. ElasticFlow provides a serverless deadline-driven interface to alleviate DL developers' resource management efforts, and exploits elastic scaling to guarantee deadlines for DL training jobs.
- We propose the metric of minimum satisfactory share to capture the minimum number of GPUs a job needs to meet its deadline under non-linear scaling curves.
- We design an admission control algorithm that decides whether a job can be admitted guaranteeing its deadline, and an elastic resource allocation algorithm that allocates GPUs to admitted jobs to maximize resource utilization.
- We implement a system prototype of ElasticFlow and integrate it with PyTorch. Evaluation results on a 128-GPU cluster show that ElasticFlow can substantially increase the number of jobs that can meet their deadlines by 1.46–7.65× compared to existing state-of-the-art solutions.

Open-source. The code of ElasticFlow is open-source and is publicly available at https://github.com/pkusys/ElasticFlow.

2 BACKGROUND AND MOTIVATION

2.1 Background

Serverless computing. Serverless computing, or Function-as-a-Service (FaaS), is an emerging paradigm to run workloads in the cloud [34]. Traditional Infrastructure-as-a-Service (IaaS) uses a server-centric model: cloud providers expose hardware resources as bare-metal servers, virtual machines, or containers, where users have to figure out how many hardware resources are needed to run a certain workload. Comparatively, in serverless computing, resource management is entirely offloaded to cloud providers. Users need to code only their workloads using functions and submit functions to the serverless platform. In addition, *low-code* development is promising for DL jobs in serverless computing [61]. As of now, serverless computing platforms do not have mature support for accelerators such as GPUs. A natural next step is to enable accelerator support on serverless computing platforms to power a wider family of workloads.

Distributed and elastic training. A DL training job trains a DNN model with a dataset. The job includes many iterations, and each uses a batch of samples from the dataset to train the model with a forward pass and a backward pass. It is time-consuming to train DNN models, so distributed training is widely used to speed up the training process. In the data parallelism [35–37, 48] strategy of distributed training, each worker maintains a copy of the DNN model locally. For each iteration, each worker first trains the model

independently, exchanges its gradients with other workers to aggregate them, and then updates its local copy of the model and begins the next iteration. The batch size of each worker is called the local batch size, and the sum of the batch size of all workers is called the global batch size. There are also other strategies, such as model parallelism [15], hybrid parallelism [32], and pipeline parallelism [9, 26].

Many solutions have been proposed to optimize single-device training [31, 56, 57, 69], the communication between workers in distributed training [13, 18, 24, 28, 33, 59, 65, 66], and distributed training algorithms [39, 40]. Several efforts have explored elastic training [7, 23, 27, 43, 45, 46, 50, 51, 62, 64]. Based on these advancements, it has become viable to apply elastic scaling in distributed training platforms. Note that enabling elasticity to speed up a single training job is not the focus of ElasticFlow. Instead, ElasticFlow focuses on scheduling multiple jobs in the cloud and exploits elasticity to provide performance guarantees in a serverless manner.

2.2 Limitations of Existing Solutions

Early efforts used cluster managers like Kubernetes or YARN to schedule DL jobs in the cloud without considering the characteristics of DL jobs, which results in low performance [11, 29, 30]. Recent efforts proposed specialized cluster schedulers for DL training jobs [12, 19, 21, 42, 49, 52, 63, 68]. These efforts either followed the *server-centric* model or disregarded DL developers' performance requirements, which have the following two limitations.

First, the server-centric model is too *low level* for DL developers: it tightly couples a DL problem with a system problem. DL developers need to explicitly request hardware resources and configure machines to run their jobs. While containers simplify system configurations and make programs more portable, DL developers are still responsible for wrapping their programs in containers with low-level system configurations. More importantly, DL training jobs are constrained by GPU hardware resources. DL developers face a *system* problem of adapting the local batch size based on GPU memory and deciding the number of workers, which affects both the global batch size and the training throughput. Moreover, DL developers face a *DL* problem of choosing the hyperparameters when training DNN models (e.g., global batch size, learning rate, etc.). The low-level server-centric interface conflates the DL problem with the system problem. This is particularly challenging for DL developers who do not normally have expertise in systems [14]. Although systems such as Amazon SageMaker [3] provide a managed service for DL training, DL developers are still faced with both the *DL* problem and the *system* problem of hardware resource configurations.

Second, the existing solutions do not have the flexibility to elastically scale the resources of DL jobs to provide *performance guarantees* (i.e., guarantee to finish a job before a particular deadline). Most existing solutions focus on optimizing job completion time (JCT) [21, 42, 63]. While this is meaningful for many scenarios, there is another important class of scenarios where DL developers require *performance guarantees* when they have an explicit expectation of their jobs' deadline [19, 38]. For example, some production environments require models to be (re-)trained and onboarded in time for regular product releases. While some recent work [19] made attempts to consider deadlines, it still adopts a server-centric approach that lacks the flexibility to elastically scale the resources of a job up or down to optimize cluster-wide resource utilization and meet deadlines.

3 ELASTICFLOW OVERVIEW

3.1 Architecture

ElasticFlow exposes a high-level serverless interface to DL developers, based on which they submit jobs to ElasticFlow. Then, ElasticFlow exploits resource elasticity to dynamically allocate resources to jobs based on their deadlines and the cluster status.

ElasticFlow interface. DL developers submit their training jobs as serverless functions to ElasticFlow. A function of a training job includes the following parameters.

- *DNN model*, which is the DNN model to be trained.
- *Hyperparameters*, which are the training hyperparameters such as global batch size, learning rate, etc.
- *Termination condition*, which is the condition indicating the completion of the job. DL developers only need to specify a maximum number of iterations. They can also add other conditions such as reaching a certain accuracy.
- *Deadline*, which is a point in time by which the DL developer expects the training job to finish.
- Other training components (dataset, optimizer, etc.).

This serverless interface differs from today's server-centric interface in two aspects. First, DL developers submit only a function that encodes a job to the platform, and the platform takes care of resource management. DL developers encode a training job in a function as *single-device* training and need to specify only the global batch size and other hyperparameters. This allows DL developers to focus on solving the DL problem, i.e., tuning the *global* batch size and the learning rate to achieve high model accuracy. The system-level resource management problem of deciding the *local* batch size and the number of workers based on the GPU memory is handled by ElasticFlow. Second, DL developers specify only the deadline for each job. This low-code style development simplifies the interaction between DL developers and the platform, as DL developers no longer need to control when to terminate allocated machine instances. It also provides flexibility for the system to dynamically adjust the resources allocated to each job based on their deadlines and termination condition.

Note that, in practice, DL developers can use a variety of conditions to decide the termination of a training job, e.g., the training accuracy is above a threshold, etc. Because of the stochastic and unpredictable nature of DL jobs (e.g., the accuracy may never exceed the threshold due to bad initialization parameters), it is common to assign a maximum number of iterations to bound the running time of a job. We use the maximum number of iterations as the main termination condition in the interface. ElasticFlow also supports scheduling jobs without deadlines (§ 4.4).

ElasticFlow architecture. Figure 1 shows the architecture of ElasticFlow. DL developers submit DL training jobs and, for each incoming job, ElasticFlow first uses the admission control module to decide whether to admit or drop the job. The admission control module obtains the current cluster status from the monitor

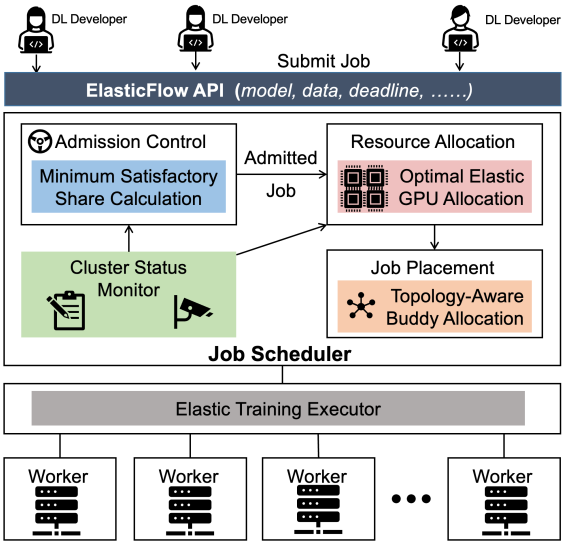


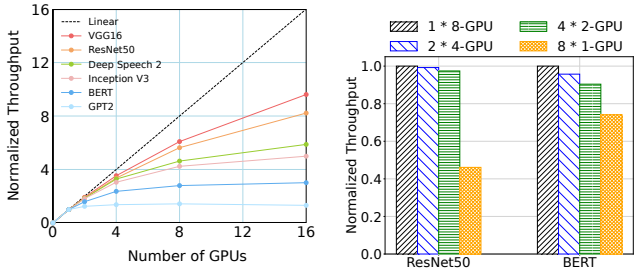
Figure 1: ElasticFlow architecture.

module and computes the minimum satisfactory share for the job (§ 4.1). The resource allocation module then schedules the admitted jobs to efficiently utilize resources and meet their deadlines (§ 4.2). Upon each scheduling event, such as job arrival or completion, the resource allocation module may update the resource allocation of some jobs through elastic scaling, i.e., adjusting the number of GPUs allocated to a job based on the deadlines of admitted jobs and the number of available GPUs. The module also computes the local batch size (i.e., dividing the global batch size by the number of GPUs) for a given job. The job placement module selects GPUs from the cluster for each job based on topology (§ 4.3). After the placement is decided, the module sends the jobs to the elastic training executor, a plugged-in component that can be replaced by any elastic DL framework. The elastic training executor ensures that each machine executes DL jobs correctly.

Performance guarantee. ElasticFlow provides the following performance guarantee: when a DL training job is admitted to the system, the deadline of the job is guaranteed to be satisfied.

3.2 Challenges

Non-linear scaling. Distributed training improves the job throughput with more workers. However, the throughput does not increase linearly with the number of workers due to parameter synchronizations and other overheads. The scaling curves for DL jobs are typically concave. Figure 2(a) shows the normalized scaling curves of six DNN models measured on varying numbers of GPUs. Each machine has eight NVIDIA A100 GPUs and eight NVIDIA Mellanox HDR InfiniBand HCAs, interconnected by a 200 GB/s InfiniBand network. We observe concave scaling curves from the collected throughputs. For example, the training throughput of VGG16 using a global batch size of 256 with eight GPUs is expected to be 8× that of with 1 GPU under linear scaling, while the actual throughput with eight GPUs is only 76.07% of that with linear scaling.



(a) Different DNN models. **(b) Different placements.**

Figure 2: Characteristics of distributed training jobs. (a) Scaling curves of popular DNN models. (b) Throughputs of different placements for 8-GPU jobs.

ElasticFlow's deadline-aware scheduling should take non-linear scaling into account. Deadline-aware scheduling is not a new problem: traditionally, Earliest-Deadline-First (EDF) is known to be effective for meeting deadlines [20]. The canonical setup of EDF assumes a single worker per job, and jobs are ordered by their deadlines. A straightforward solution to apply EDF to scheduling DL jobs is to view the entire cluster as a logical worker, the throughput of which is the sum of the throughput of all the machines. However, this solution does not work for non-linear scaling jobs. For example, consider jobs A and B, which both have the same scaling curve, as shown in Figure 3(a): the throughput is 1 unit with one worker, and 1.5 units with two workers. Let the deadlines of A and B be at time unit 3 and 3.5, respectively. Let the job sizes of both A and B be 3 units of iterations. EDF first runs A and then runs B; A's deadline is satisfied, but B's deadline is violated (Figure 3(b)). Alternatively, if we use one worker for each job, the deadlines of both jobs are satisfied (Figure 3(c)).

Topology-dependent placement. The performance of a distributed training job not only depends on the number of workers but also on the placement of the workers topologically. This is because the speed of parameter synchronization between workers is decided by the communication bandwidth, which changes based on how the workers are connected. When the workers are on the same server, the communication can use PCIe (32 GB/s bandwidth for PCIe 4.0×16) or NVLink (600 GB/s bidirectional bandwidth for third-generation NVIDIA NVLink). When the workers are on different servers in the same rack, the bandwidth is determined by networking, which is typically 40 Gbps Ethernet, 200 Gbps InfiniBand, or 8×200 Gbps InfiniBand for high-end GPU servers. The bandwidth is lower when the workers are in different racks.

To illustrate the problem, we measure the throughputs of training ResNet50 and BERT with different worker placements. The global batch size is 256. Each machine is configured with eight NVIDIA A100 GPUs, which are intra-connected by NVLink with 600 GB/s bidirectional bandwidth and inter-connected by an InfiniBand network with 200 GB/s bandwidth. As shown in Figure 2(b), the throughputs are different depending on the placement of the workers. When the eight workers are on the same server, the throughput is 2.17× of that when the eight workers are on eight different servers for ResNet50. We only show the scaling throughputs

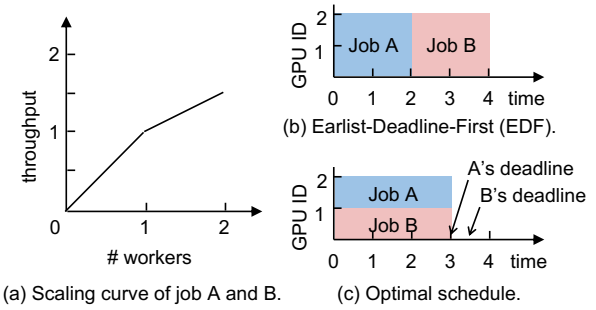


Figure 3: Motivating example to show that EDF does not work well for jobs with non-linear scaling curves.

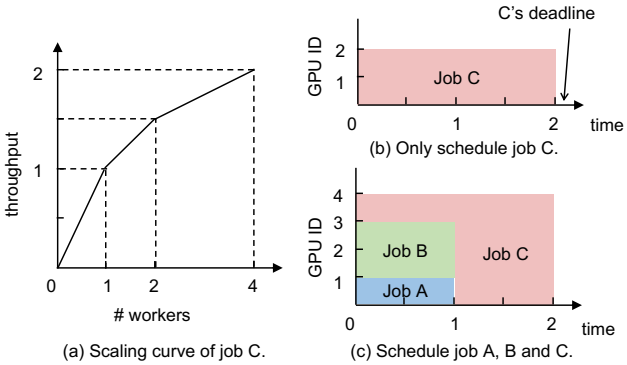


Figure 4: Example for ElasticFlow admission control. The GPU time to finish a job changes depending on other jobs. Finishing job C with the scaling curve in (a) requires 4 units of GPU time in (b) and 5 units of GPU time in (c).

for two models and four placements. There are more combinations, as eight workers can be divided between servers in many different ways.

The topology-dependent placement further complicates the resource allocation problem in ElasticFlow. When allocating resources to jobs, the scheduler needs to consider a set of scaling curves instead of a single scaling curve; the curves also depend on job placement.

4 ELASTICFLOW DESIGN

In this section, we describe the detailed design of ElasticFlow to address the above challenges. We propose Minimum Satisfactory Share to capture the minimum resource usage of DL jobs to meet deadlines for admission control. ElasticFlow applies buddy allocation to eliminate the effect of topology on job placement, thus reducing the number of scaling curves to be considered for each job. ElasticFlow dynamically allocates resources to admitted jobs to maximize resource efficiency, taking non-linear scaling into account.

4.1 Admission Control

Minimum Satisfactory Share. The scaling curves for DL training jobs are concave. This implies diminishing returns: the benefit of adding an extra GPU to a training job decreases with the number of GPUs. As a result, the per-GPU throughput drops as the number of GPUs increases and using a single GPU for a training job is the most efficient. We define resource usage as the number of GPUs times the running time, i.e.,

$$\textit{resource usage} = \textit{number of GPUs} \times \textit{running time}.$$

For example, suppose we have a training job with the scaling curve in Figure 4(a). The throughput is 1 unit, 1.5 units, and 2 units with one, two, and four GPUs, respectively. If it requires 1 time unit to train a job with one GPU, then it would require 2/3 of a time unit with two GPUs and 1/2 a time unit with four GPUs to train the job. The resource usage is 1, 4/3, and 2 units of GPU time with one, two, and four GPUs, respectively. Using a single GPU has the lowest GPU resource usage. However, because jobs have deadlines, we cannot train every job with a single GPU as it may violate their deadlines.

We define minimum satisfactory share as the lowest number of resources needed to train a job in order to meet its deadline. Resource usage is minimized when jobs are assigned with their minimum satisfactory shares to run. For the preceding example, if the deadline of the job is 1 time unit, then we can simply allocate one GPU to the job. But if the deadline is 2/3 of a time unit then we need to allocate two GPUs to the job to meet its deadline. Allocating two GPUs minimizes resource usage while meeting the deadline of the job.

Admission control. Jobs are only admitted if ElasticFlow can guarantee their deadlines. Specifically, ElasticFlow computes the minimum satisfactory share of each arriving job. If the system has more resources than the minimum satisfactory share, the job can be admitted as its deadline can be guaranteed. Otherwise, the job is dropped to avoid violating the performance guarantee.

When there is no running job in the cluster, computing the minimum satisfactory share for an incoming job is straightforward: we only need to allocate the smallest number of GPUs so that the completion time of the job is no later than its deadline. This can be done efficiently with a binary search.

When there are already jobs admitted by the system, directly using binary search to solve the above problem is not sufficient. For example, suppose we have a job C with the scaling curve shown in Figure 4(a). Let the deadline of job C be 2 time units. When there is only job C, it can use two GPUs to finish in 2 time units, as shown in Figure 4(b). Now suppose there are already two jobs (A and B) in the system, and suppose those need to use one and two GPUs for 1 time unit, respectively. Because there are only four GPUs in total, job C cannot use two GPUs for the first time unit. In this scenario, the minimum satisfactory share for job C is to use one GPU for the first time unit and four GPUs for the second time unit, in order to meet its deadline.

We first describe how to solve the simple case of linear scaling curves to obtain intuitions, and then extend to non-linear curves. For the linear scaling case, the per-GPU throughput is the same regardless of the number of allocated GPUs. Formally, let $T(x_i) = k_i \cdot x_i$ be the scaling curve of job i, i.e., the throughput of job i is $T_i(x_i)$ when the number of GPUs allocated to it is x_i and k_i is the per-GPU throughput. The latter is measured as the number

of iterations per time unit. Let M_i be the maximum number of iterations to run, which represents the termination condition. Let D_i be the deadline of job i. To meet the deadline, we need to allocate $M_i/(k_i D_i)$ GPUs to job i for D_i amount of time, or equivalently a total amount of M_i/k_i GPU time. Let G be the total number of GPUs. We have the following theorem.

Theorem 1. *For n jobs with linear scaling curves, we sort the jobs by their deadlines in increasing order. If the condition*

$$\forall\ i \le n-1,\ \sum_{j=0}^{i} M_j/k_j \le G \cdot D_i. \tag{1}$$

holds, then there exists an allocation that can guarantee the deadlines for all jobs. Otherwise, no allocation can guarantee the deadlines for all jobs.

The intuition of the theorem is that because the scaling curves are linear, it does not matter how many GPUs are used for a job in each time slot. We only need to: (i) sort jobs by deadlines, and (ii) account for the total GPU time consumed by a job and check if it exceeds the amount needed by the job before its deadline. Condition (1) ensures that the required GPU time of the jobs does not exceed the available GPU time. By ordering the jobs by their deadlines in increasing order, we can prove the theorem by induction.

The problem becomes challenging for DL jobs due to their non-linear scaling curves. In this case, we cannot simply use M_i/k_i to compute the amount of GPU time needed by job i. Let $x_i(t)$ be the number of GPUs allocated to job i at time slot t. Then the admission control is to check whether we can find an allocation $A = \{x_i(t)\}$ that satisfies the following two conditions:

$$\forall\ i,\ \sum_{t=0}^{D_i} T_i(x_i(t)) \ge M_i; \tag{2}$$

$$\forall\ t,\ \sum_{i=0}^{n-1} x_i(t) \le G. \tag{3}$$

The first condition represents that the total number of iterations performed by a job is no smaller than the maximum number of iterations of the job, i.e., the deadline is met. The second condition represents that the total number of GPUs allocated to the jobs in any time slot is no bigger than the number of available GPUs.

We apply the intuition of solving the problem under linear scaling curves and adapt progressive filling [10] to compute the minimum satisfactory share under non-linear scaling curves. The key idea is to: (i) sort the jobs by deadlines, and (ii) increase the number of GPUs for each job progressively until the deadline of the job can be met. Algorithm 1 shows the pseudocode of admission control. The algorithm first sorts the jobs by deadlines in increasing order (line 3). For each job, it checks whether the deadline of all admitted jobs can still be satisfied after adding the new job (lines 4-7). It increases the number of GPUs j for job i from one (line 13). But j should not be larger than the number of remaining GPUs in the cluster (line 15). The algorithm sums up the number of iterations job i can perform at time slot t. If the sum is no smaller than the maximum number of iterations, the job can finish before its deadline (lines 17-19). If job i cannot be satisfied after enumerating the number of GPUs, it means adding the new job would violate the deadline of at least one job. Therefore, the new job is dropped (lines 5-7). Otherwise, it is safe to admit the new job (lines 8-9).

Algorithm 1 Admission Control

```
1:  function AdmissionControl(job)
2:      Add job to jobs
3:      Sort jobs by deadline in increasing order
4:      for i in jobs do
5:          if ProgressiveFilling(i, 0) is False then
6:              // Drop job
7:              return False
8:      // Admit job
9:      return True
10:
11: function ProgressiveFilling(i, startTime)
12:     isSatisfied ← False
13:     for j from 1 to G do
14:         for t from D_i to startTime do
15:             x_i(t) ← min(j, G − Σ_{k=0}^{k=i−1} x_k(t))
16:             // Check if the job can be satisfied
17:             if Σ_{t=0}^{D_i} T_i(x_i(t)) ≥ M_i then
18:                 isSatisfied ← True
19:                 Break
20:         if isSatisfied is True then
21:             Break
22:     return isSatisfied
```

We use the example in Figure 4 to illustrate how the algorithm works. Let $D_c = 2$ and $M_c = 3$. Suppose there are already two jobs A and B in the system and they need to use 3 GPUs for the first time slot. If j is 2, then job C can use only 1 GPU for the first time slot and 2 GPUs for the second time slot. In this case, $\sum_{t=0}^{2} T_c(x_c(t)) = T_c(1) + T_c(2) = 2.5 < M_c$, which means job C does not finish in 2 time slots. If j is 4, then job C gets 1 GPU for the first time slot and 4 GPUs for the second time slot, and we have $\sum_{t=0}^{2} T_c(x_c(t)) = T_c(1) + T_c(4) = 3 \ge M_c$, which means the deadline of job C can be met.

4.2 Resource Allocation

Allocating the minimum satisfactory share to each job can already provide a performance guarantee, but after each job is allocated with its minimum satisfactory share, the cluster may still have idle GPUs left. It is beneficial to allocate the remaining GPUs to the admitted jobs, because if allocating more GPUs further speeds up the jobs, ElasticFlow would free up more GPUs even before the deadlines of the admitted jobs. This would allow the system to accommodate more jobs in the future. Our goal is to find an *efficient* resource allocation that fully utilizes the GPUs while guaranteeing the deadlines of all admitted jobs.

A straw-man solution is to allocate all the remaining GPUs to the job with the earliest deadline. But this does not make the best use of GPU resources, because the marginal return of adding one extra GPU diminishes with more GPUs under non-linear scaling curves. Another straw-man solution is to allocate the remaining GPUs evenly to the admitted jobs. This solution avoids adding all resources to one job, but it does not consider the current number of

GPUs a job has, which affects the benefit of adding one extra GPU to a job.

From the straw-man solutions, we can see that the key to achieving our goal is to account for the diminishing returns of each job given its allocated GPUs. To derive the optimal solution, we formally formulate the problem as follows. We consider the next time slot for resource allocation. Note that it is unnecessary to consider further time slots, because even if we do, new jobs may arrive and be admitted, and we would have to recompute the resource allocation plan. We use $a_i(t)$ to denote the GPU allocation that has already been assigned to job i over time. When resource allocation starts, it is equal to the number of GPUs allocated to job i calculated by admission control (Algorithm 1). Let $x_i(t)$ be the actual number of GPUs to allocate to job i over time. Then resource allocation is to solve the following optimization problem:

$$\min \sum_{i=0}^{n-1} \sum_{t=0}^{D_i} x_i(t) \tag{4}$$

$$s.t.\ \forall\ i,\ \sum_{t=0}^{D_i} T_i(x_i(t)) \geq M_i; \tag{5}$$

$$\forall\ t,\ \sum_{i=0}^{n-1} x_i(t) \leq G; \tag{6}$$

$$\forall\ i,\ T_i(x_i(0) + \min(1,\ G - \sum_{i=0}^{n-1} x_i(0))) <= T_i(x_i(0)). \tag{7}$$

In the formulations, $\sum_{t=0}^{D_i} x_i(t)$ is the total GPU time used by job i with $x_i(t)$ GPUs at time slot t. Under sub-linear scaling, the more GPUs are allocated to a job, the more GPU time it needs to reach the termination condition. The objective is to minimize the total GPU time consumed by all jobs. There are three constraints: (i) all jobs' deadlines are guaranteed; (ii) the total allocated GPUs do not exceed the number of available GPUs; (iii) all GPUs are allocated in the next time slot unless adding more GPUs to any job would make the job slower.

We develop a greedy algorithm to solve the problem. The intuition is to allocate the remaining GPUs to the job with the highest marginal return. Algorithm 2 shows the pseudocode. The algorithm allocates at least the minimum satisfactory share a_i to each job so that the deadlines are guaranteed (line 4) and then maintains a priority queue to order the jobs (lines 5-11). The priority of a job is the marginal return to add one extra GPU to it. The algorithm greedily allocates one GPU each time (lines 12-24): for each iteration, the algorithm removes the head from the queue (lines 13-14), allocates one GPU to the job (line 18), computes the new marginal return of the job (lines 18-22), and inserts the job back into the queue (lines 23-24). The iterations do not finish until all GPUs are allocated. We have the following theorem for the optimality of the algorithm.

Theorem 2. *Algorithm 2 computes an optimal allocation for the resource allocation problem in formula (4-7), i.e., it finds the most efficient allocation so that the GPUs are fully utilized while guaranteeing the deadlines of all admitted jobs.*

The main idea of the proof of the theorem is to show that allocating each GPU to the job with the highest marginal return maximizes GPU utility. It applies induction to prove the general case of multiple GPUs.

Algorithm 2 Resource Allocation

```
1:  function ResourceAllocation(jobs)
2:      for i in jobs do
3:          // Allocate a_i(0) to job i
4:          G ← G − a_i(0)
5:          // Assume job i gets 1 more GPU at time slot 0
6:          // Calculate the marginal return of job i
7:          x_i(0) ← a_i(0) + 1
8:          ProgressiveFilling(i, 1)
9:          i.priority ← Σ_{t=0}^{D_i} a_i(t) − Σ_{t=0}^{D_i} x_i(t)
10:         if i.finish_time(x_i) < i.finish_time(a_i) then
11:             Add jobs to priority queue Queue
12:     while G > 0 and !Queue.empty() do
13:         // Pick the job with largest marginal return
14:         i ← Queue.dequeue()
15:         // Assign x_i(t) as the new allocation of job i
16:         for t from 0 to D_i do
17:             a_i(t) ← x_i(t)
18:         G ← G − 1
19:         // Update the marginal return of job i
20:         x_i(0) ← a_i(0) + 1
21:         ProgressiveFilling(i, 1)
22:         i.priority ← Σ_{t=0}^{D_i} a_i(t) − Σ_{t=0}^{D_i} x_i(t)
23:         if i.finish_time(x_i) < i.finish_time(a_i) then
24:             Queue.insert(i)
```

4.3 Job Placement

After the number of GPUs for a job is decided, the job needs to be placed in the cluster, i.e., selecting which GPUs to run the job. As we have shown in §3.2, the scaling curve of a job depends on the placement. To make the problem more challenging, job placement intertwines with admission control and resource allocation, because they require scaling curves to make decisions. If admission control uses a different scaling curve from the actual placement, the actual job throughput may be lower than the one used by admission control, leading to deadline violations.

A naive approach is to always use the most pessimistic scaling curve, i.e., when all workers of a job are on different machines. However, this potentially *underestimates* the throughput of a job, and thus *overestimates* the resource usage. This would unnecessarily reserve GPUs for admitted jobs, preventing the system from admitting more.

Topology-aware job placement. We develop a topology-aware job placement algorithm to address this problem. We use a multi-layer hierarchical tree to capture the GPU topology where GPUs are connected by different types of links with different bandwidths. The leaf nodes of the tree represent GPUs. Each internal node indicates a particular connection. Figure 5 shows an example of a server with a 4-layer hierarchy. The server has two CPU sockets connected by QPI. Each CPU is connected to four GPUs via PCIe. In GPU clusters, multiple such servers can be connected by a Top-of-Rack (ToR)

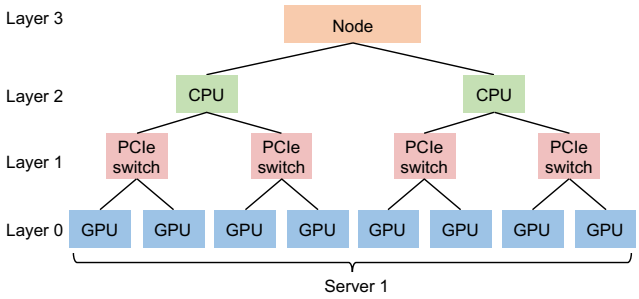


Figure 5: ElasticFlow organizes GPUs in a multi-layer hierarchy, and uses buddy allocation to minimize fragmentation.

switch in the same rack, which is the 5^{th} layer in the hierarchy of the cluster.

To place a job, we find a suitable subtree that contains sufficient idle GPUs to accommodate the job. There can be multiple subtrees that satisfy this requirement. This is a bin packing problem and there are multiple heuristics. We use Best-Fit [54], which chooses the subtree in which the number of idle GPUs is closest to the number of GPUs needed. In this way, the job is allocated with the highest possible bandwidth between its workers, thus avoiding overestimating the resource usage of jobs. Therefore, job placement can be decoupled from admission control and resource allocation.

Defragmentation with buddy allocation. Our algorithm is general to jobs that require any number of GPUs, but the algorithm cannot avoid resource fragmentation. We may not be able to find a subtree for a job, but the cluster contains more GPUs than needed. For example, suppose that there are two servers with the hierarchy in Figure 5. If we allocate seven GPUs from server 1 to job A and seven GPUs from server 2 to job B, we cannot find a layer-1 subtree with two GPUs for job C, although there are two idle GPUs in the cluster.

We apply buddy allocation combined with job migration used by prior DL schedulers [27, 67] to eliminate resource fragmentation. Similar to CoDDL [27], we restrict the number of workers of each job to be a power of two. Under this condition, our algorithm guarantees that there is no resource fragmentation. If the cluster has enough idle GPUs for a job, there is a subtree that contains enough GPUs.

4.4 Discussion

Handling jobs without deadlines. ElasticFlow can be extended to support both SLO jobs (with deadlines) and best-effort jobs (without deadlines). While best-effort jobs do not have deadlines, it is desirable to finish them as soon as possible. The objective for this scenario is to guarantee deadlines for SLO jobs and minimize the average JCT for best-effort jobs. ElasticFlow uses a unified framework to support both kinds of jobs. It sets the deadlines of best-effort jobs to infinite and allocates GPU resources to them after allocating the minimum satisfactory shares to SLO jobs in Algorithm 2. We show the evaluation results of supporting best-effort jobs in §6.5.

Dropped jobs and hard vs. soft deadlines. ElasticFlow provides performance guarantees for hard deadlines and drops a job if its deadline cannot be satisfied. In practice, there are jobs with soft deadlines: finishing them is still meaningful even if the deadlines are missed. Supporting jobs with soft deadlines is similar to supporting best-effort jobs. ElasticFlow puts them in the queue in Algorithm 2 and allocates resources to them after allocating the minimum satisfactory shares to admitted jobs. This ensures that the deadlines for admitted jobs are always guaranteed; other jobs finish as early as possible.

Malicious users and admission control policies. We assume that users are not malicious and set their job deadlines according to their needs. Users may be able to game the system by manipulating job submission frequency and deadlines. For example, a user may submit many jobs with close deadlines to occupy all GPUs in the cluster, preventing the system from admitting jobs from other users. To address this problem, the cloud operator can use quotas (e.g., set a maximum number of jobs that can be submitted by each user per day) or charge users (e.g., the cost depends on the job size and the deadline). The cloud operator can apply an extra policy or charge the user before line 9 of Algorithm 1 to decide whether to actually admit the job. Such policies and pricing models are orthogonal to ElasticFlow and are interesting directions for future work.

Node failures. In real-world clusters, there might be random node failures. ElasticFlow can be extended to consider node failures by calculating their probability and reserving enough resources to ensure a performance guarantee in such cases.

Table 1: ElasticFlow API.

API	Description
init_model()	Initialize the DNN model to be trained.
set_hyperparameter(bs, lr, kwargs)	Set the DL-related hyperparameters.
term_condition(kwargs)	Set the termination condition.
set_deadine(ddl)	Set the deadline.
init_dataset()	Process the data and return a dataset that can be passed to a DataLoader.
init_optimizer()	Return the optimizer for training.
train(dataset, model, optimizer, kwargs)	Train the model for one iteration.

5 IMPLEMENTATION

We have implemented a prototype of ElasticFlow in 8,800+ LOC of Python, including 5,600+ LOC for the scheduler and 3,200+ LOC for elastic training. We integrate it with PyTorch 1.10.0 [48]. We use PyTorch DistributedDataParallel (DDP) for distributed training, NCCL [1] for communicating between workers, and gRPC [5] to exchange control messages between the scheduler and workers.

ElasticFlow interface. ElasticFlow currently supports DL training jobs written in PyTorch. PyTorch provides an official launch API in PyTorch DDP to start distributed training, and ElasticFlow uses this to launch distributed training jobs. As specified in §3, developers submit their custom DNN model, the hyperparameters of the job, the termination condition, the deadline and other training components like dataset and optimizer to ElasticFlow. Table 1 shows details of the API.

Table 2: DNN models used in the evaluation.

Task	Dataset	Model	Batch Size
CV	ImageNet [16]	ResNet50 [25]	64, 128, 256
		VGG16 [55]	64, 128, 256
		Inception V3 [58]	64, 128
NLP	CoLA [60]	BERT [17]	64, 128
	aclImdb V1 [41]	GPT-2 [53]	128, 256
Speech Recognition	LibriSpeech [47]	Deep Speech 2 [8]	32, 64

Elastic scaling. ElasticFlow enables elastic scaling in a stop-free manner in multi-node multi-GPU environments. Specifically, when a new scheduling decision is made, ElasticFlow sends the parameters of the running jobs to the workers based on the scheduling decision and then restarts the jobs from the received parameters. The local batch size of each worker is adjusted to maintain the same global batch size. If a running job is suspended, ElasticFlow checkpoints the parameters until it restarts. To reduce the scaling overhead, ElasticFlow does not delete the CUDA contexts on GPUs and keeps all NCCL process groups active. We evaluate the scaling and migration overheads in §6.6.

Throughput profiling. ElasticFlow makes scheduling decisions based on simulating future job events such as job scaling and job completion. The simulation is done by calculating the number of iterations that can be completed by each job on every job event. To precisely calculate the executed iterations, ElasticFlow first pre-runs each job to profile its throughput with different numbers of GPUs. Then, ElasticFlow profiles its throughput during job execution and constantly adjusts the profiled throughput and scheduling decisions accordingly. We evaluate the overhead of pre-run profiling in §6.6. The overhead of profiling during job execution is about 0.01 s per epoch, which is rather small as most jobs run for hours or even days.

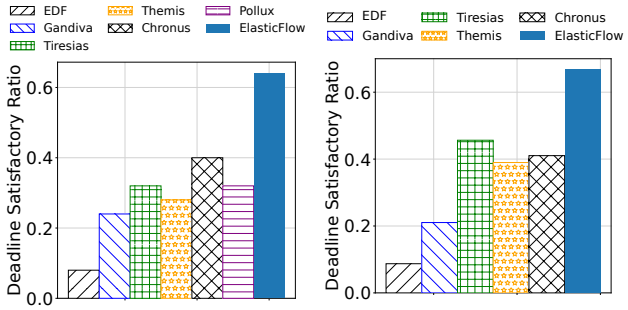
6 EVALUATION

6.1 Methodology

Testbed. Unless specified otherwise, the experiments are conducted on a cluster with 16 ND A100 v4 servers [6] that includes 128 NVIDIA A100 GPUs in total. Each server is equipped with eight 40 GB NVIDIA A100 GPUs, 96 CPU cores, 900 GB RAM, and eight NVIDIA Mellanox HDR InfiniBand HCAs.

Simulator. To evaluate ElasticFlow on larger scales, we develop a simulator using the profiled information in real A100 GPUs. The simulator simulates all job-level events, including job arrival, scaling, and completion. We profile the throughputs of each job with real GPU servers on the testbed as the input of the simulator. To make the simulator more realistic, we have also measured the job scaling overhead and incorporated it. The simulator assigns the overhead to each job on each scheduling event, according to the number of GPUs and the DNN model of the job. Our simulator has very high fidelity, with an error rate of no more than 3% compared with the results in our real cluster experiments.

Workloads. We collect two-month real-world traces from ten Microsoft's internal ITP clusters [4]. The cluster size ranges from 164



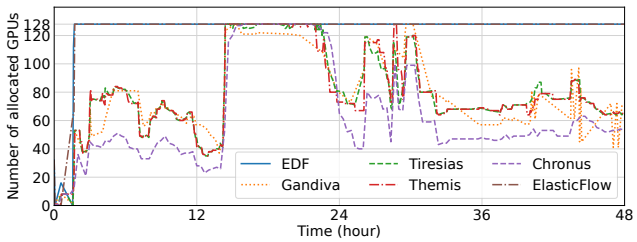
(a) Testbed experiments on 32 GPUs. (b) Testbed experiments on 128 GPUs.

Figure 6: Deadline satisfactory ratio in testbed experiments.

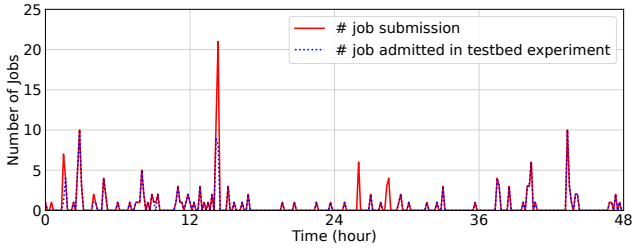
GPUs to 2,783 GPUs; the number of jobs in each trace ranges from 260 to 15,802. Testbed experiments use traces from one of the clusters; simulations use all the collected traces to evaluate ElasticFlow more comprehensively and on larger scales. Each job in the original traces has information about submission time, number of GPUs, and duration. For each job, we randomly choose a DNN model with a batch size from a pool of representative settings listed in Table 2. Similar to previous work [27, 44], we use the duration in the trace and the pre-measured throughput to calculate the number of iterations needed to complete each job. As the traces do not contain any deadline information, we set the deadline for a job at $\lambda \cdot duration$ after its submission, where λ is drawn uniformly from $[0.5, 1.5]$. λ represents the tightness of the deadline. Note that even when a job's $\lambda < 1$, the cluster may still be able to complete it before the deadline by scaling out the job with more GPUs. For a fair comparison, we also evaluate ElasticFlow with the public Microsoft Philly cluster trace [29].

Baselines. We compare ElasticFlow to six baselines.

- **Earliest-Deadline-First (EDF):** EDF is a canonical scheduling policy that optimizes for meeting deadlines [20]. It orders jobs based on their deadlines and schedules jobs with the earliest deadline first. It uses as many GPUs as a job can scale out without decreasing the throughput.
- **Gandiva:** Gandiva [63] is a DL scheduler that uses introspective scheduling to refine scheduling decisions continuously. It is not elastic (i.e., uses the number of GPUs specified in job traces) and is not deadline-aware.
- **Tiresias:** Tiresias [21] uses two-dimensional scheduling algorithms customized for DL jobs. It is also not elastic and is not deadline-aware.
- **Themis:** Themis [42] provides finish-time fairness for DL jobs. We follow the open-source implementation of Themis in [44]. It is not deadline-aware.
- **Chronus:** Chronus [19] maximizes the number of SLO jobs that can meet deadlines and minimizes the average JCT of best-effort jobs. It is deadline-aware but not elastic.
- **Pollux:** Pollux [52] is a state-of-the-art scheduler for DL jobs. It improves JCT by adaptively co-optimizing statistical efficiency and system throughput. It is elastic but not deadline-aware.



(a) GPU allocation of different schedulers over time.



(b) Number of submitted and admitted jobs by ElasticFlow over time.

Figure 7: Comparison between different schedulers in testbed experiments.

Evaluation metric. The design goal of ElasticFlow is to meet deadlines for DL training jobs. Therefore, the evaluation metric is *deadline satisfactory ratio*, which is the ratio of jobs that can meet their deadlines. We also evaluate *cluster efficiency* (CE), which measures how efficiently the resources in a cluster are being utilized. Let M be the number of GPUs in the cluster and tpt represent throughput, then

$$CE := \frac{1}{M}\sum_{i=1}^{M}\frac{Current\ tpt\ of\ the\ job\ on\ GPU_i}{tpt\ of\ the\ job\ on\ GPU_i\ with\ 1\ GPU}. \quad (8)$$

Because the admission control module of ElasticFlow may drop the jobs with unachievable deadlines, we do not compare the scheduling results on metrics such as JCT. We compare the JCT of best-effort jobs when extending ElasticFlow to schedule both SLO and best-effort jobs (§6.5).

6.2 End-to-End Results on a Real Testbed

We compare ElasticFlow and the baselines on a real GPU testbed. We follow previous work [63] to speed up the experiments by fast-forwarding. i.e., skipping a number of iterations if there are no scheduling events for more than four minutes. The skipped time is calculated by measuring the throughput when the job reaches a stable state with the throughput profiling method described in §5. Because Pollux requires the gradients of the DL models' weight during the training process, running DL jobs with Pollux cannot be fast-forwarded. The makespan of running all DL jobs with Pollux depends on the scale of the trace to run. We estimate that using Pollux, running a two-day trace from our collected data on our testbed costs more than $124,000 on Amazon AWS [2]. Due to budget limitations, we first compare ElasticFlow and all baselines on a small cluster of four servers with 32 GPUs and a small trace with 25 jobs.

Figure 6(a) shows the results of comparing ElasticFlow with all baselines. ElasticFlow improves the number of jobs that can meet deadlines compared to EDF, Gandiva, Tiresias, Themis, Chronus, and Pollux by 8.0×, 2.7×, 2.0×, 2.3×, 1.6×, and 2.0×, respectively. ElasticFlow outperforms Gandiva, Tiresias, Themis, and Pollux because it is deadline-aware and drops jobs whose deadlines cannot be met. EDF and Chronus are not elastic and cannot fully utilize GPU resources, leading to low deadline satisfactory ratios.

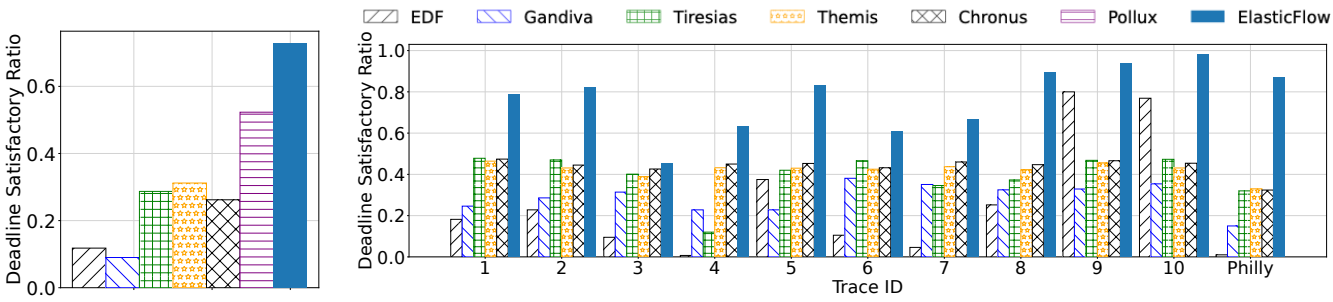
Then, we compare ElasticFlow with the baselines except for Pollux on a larger scale (on 16 servers with 128 GPUs and a larger trace with 195 jobs). As shown in Figure 6(b), ElasticFlow improves the number of jobs that can meet deadlines compared to EDF, Gandiva, Tiresias, Themis, and Chronus by 7.65 ×, 3.17×, 1.46×, 1.71×, and 1.62×, respectively. We will discuss the sources of improvement of ElasticFlow in § 6.4.

Figure 7 shows the number of allocated GPUs and the number of submitted and admitted jobs during execution on the testbed. From the figure, we observe that when there are fewer jobs submitted to the cluster, the resource contention in the cluster is low. ElasticFlow can take full advantage of the idle GPU resources so that admitted jobs can complete earlier. Schedulers such as Gandiva and Tiresias are not elastic and thus cannot utilize idle GPUs. When there is a burst of job submissions (e.g., the 13th hour), some jobs are dropped to guarantee the deadlines of admitted jobs.

6.3 End-to-End Results in Simulations

We compare ElasticFlow and the baselines with simulation in more settings. The simulation of Pollux requires profiling step time as well as other detailed training statuses of thousands of DL model settings and topology settings, and the estimated cost is about $90,000 on Amazon AWS. We do not profile them on our testbed due to budget limitations. Instead, we use the profiled data provided by Pollux and transform the 195-job trace used in § 6.2 into the form in which we can run the Pollux simulation. The data provided by Pollux was profiled on a cluster of 16 servers with 64 NVIDIA T4 GPUs. Figure 8(a) shows the simulation results. Similar to the results of testbed experiments, ElasticFlow achieves a much higher deadline satisfactory ratio than all baselines.

Figure 8(b) compares ElasticFlow with the baselines except for Pollux on more traces and larger scales. We use the job traces collected from ten Microsoft ITP clusters [4] and one public trace of the Microsoft Philly cluster [29]. We observe that ElasticFlow consistently outperforms the baselines in all traces. The deadline satisfactory ratios of Gandiva and Tiresias do not change much across the traces because they are not deadline-aware and lack elasticity. In some traces (#9 and #10), EDF achieves a higher deadline satisfactory ratio than Gandiva, Tiresias, Themis, and Chronus. This is because when the cluster size is large enough for the jobs in these traces, EDF runs jobs with the earliest deadline first and uses adequate GPU resources to finish them as quickly as possible. However, EDF does not make the most efficient use of GPUs due to the sub-linear scaling of DL training jobs. Therefore, in other traces where the GPU resources in the cluster are not adequate, EDF



(a) Simulation of all baselines.

(b) Simulation on multiple traces.

Figure 8: Deadline satisfactory ratio under difference traces.

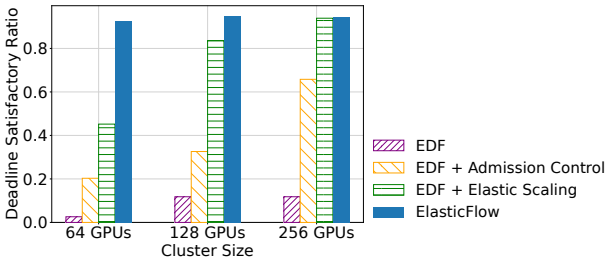


Figure 9: Sources of improvement in ElasticFlow.

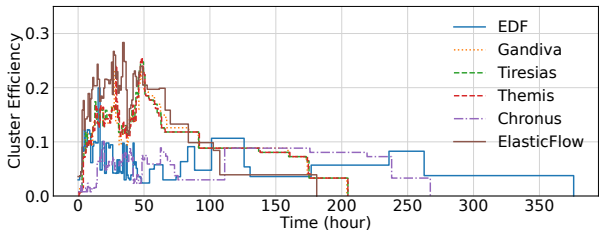
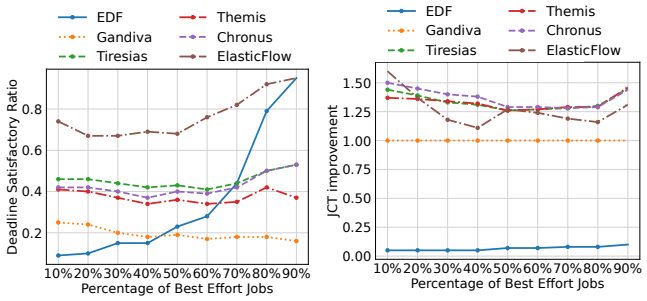


Figure 10: Cluster efficiency of different schedulers.



(a) (b)

Figure 11: Performance under a mix of SLO jobs (with deadlines) and best-effort jobs (without deadlines). (a) Deadline satisfactory ratio of SLO jobs. (b) Average JCT of best-effort jobs (normalized to Gandiva).

performs poorly. On average, ElasticFlow improves the deadline satisfactory ratio to 12.95×, 2.58×, 2.15×, 1.76×, and 1.68× compared to EDF, Gandiva, Tiresias, Themis, and Chronus, respectively.

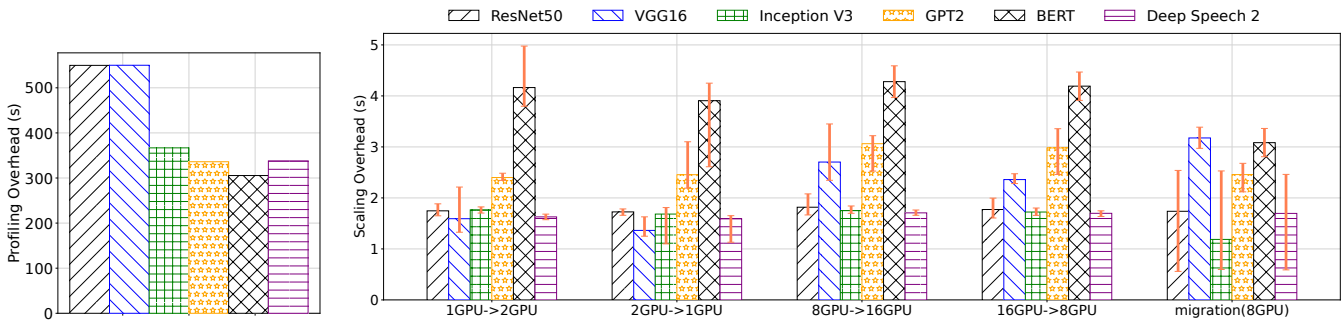
6.4 Sources of Improvement in ElasticFlow

Improvement in deadline satisfactory ratio. We analyze the sources of improvement in ElasticFlow. ElasticFlow has two key components, which are admission control and elastic scaling. To show that both components are important, we develop two variants on top of EDF, which are EDF + Admission Control and EDF + Elastic Scaling. We vary the cluster size and keep the same load. First, Figure 9 shows that both admission control and elastic scaling contribute to the improvement of ElasticFlow, as admission control drops the jobs with unachievable deadlines and elastic scaling adjusts the resource allocation in time according to the cluster status and jobs' deadlines. Only adding one of them to EDF performs worse than ElasticFlow. Second, we observe that the gap between EDF + Elastic Scaling and ElasticFlow decreases when the cluster size increases. This is because, when the load is kept the same, most jobs can be admitted with more GPUs, and elastic scaling is critical to the improvement. On the other hand, when the cluster size is small, admission control is important to avoid wasting resources on jobs where the deadlines can never be met.

Improvement in CE. We evaluate how well ElasticFlow utilizes cluster resources. We simulate the execution of a 100-job trace on a 16-node cluster where each node has eight GPUs. If ElasticFlow declines any job in the trace, the comparison is unfair to the baselines. Therefore, we set the deadlines to be loose enough to admit all jobs (1.5× a job's duration). This ensures that all solutions run the same set of jobs. Figure 10 shows that ElasticFlow achieves a higher CE than the baselines in the first 100 hours to save more GPU resources for later jobs. This is because the resource allocation module allocates the idle GPUs in the cluster in such a way that they are used efficiently. ElasticFlow also achieves the smallest makespan when finishing all jobs.

6.5 Handling Jobs without Deadlines

ElasticFlow can schedule both SLO jobs and best-effort jobs. We vary the percentage of best-effort jobs, and we measure the deadline satisfactory ratio of SLO jobs and the average JCT of best-effort



(a) Profiling overheads of different models. **(b) Scaling and migration overheads of different cases.**

Figure 12: System overheads of ElasticFlow.

jobs. Because the average JCT under EDF is too large compared to other solutions, we normalize the average JCT of each solution to that of Gandiva for illustration purposes. As shown in Figure 11, ElasticFlow achieves the highest deadline satisfactory ratio for jobs with deadline requirements among all the solutions. For traces with 10% of best-effort jobs, ElasticFlow achieves the smallest average JCT for best-effort jobs. In other traces, ElasticFlow reserves more GPU resources for jobs with deadline requirements and sacrifices JCT for deadline satisfactory ratio as expected.

6.6 System Overheads

Profiling overheads. Figure 12(a) shows the profiling overheads of different DL models. ElasticFlow profiles the throughput of each new DL model with different numbers of GPUs along with different batch sizes. ElasticFlow records the largest local batch size of each job that the GPU memory can hold, and records the largest and smallest number of GPUs for each job to avoid poor performance or memory overflow. If adding more GPU to a job with a specific batch size cannot increase the throughput of the given job, ElasticFlow stops the profiling procedure for that batch size. As DL training usually takes hours, days or even weeks, the profiling overhead is marginal. There is no need to profile known/repeated jobs.

Scaling and migration overheads. The scaling and migration overhead is the interval between "*suspending a job*" and "*restarting the job on a new set of GPUs*". Figure 12(b) shows the scaling and migration overheads of different DL models for five cases. The first four cases change the number of GPUs for a job, and the fifth case changes the set of GPUs (i.e., migrating to eight GPUs on another machine). Our current prototype uses simple checkpointing/restoration available in PyTorch for scaling and migration. When the number of GPUs is large, the overheads of the internal implementation of checkpointing/restoration in PyTorch dominate. When we scale up or down from a single GPU, the ElasticFlow overheads are visible. Therefore, the overheads of different cases are similar; the variance is mostly from the internal implementation of checkpointing/restoration in PyTorch. This implementation can be further optimized by keeping the model weights in memory if the GPU devices will train the same DL model after a scaling event. The average scheduling interval of ElasticFlow on the real-world traces we collected in a production system is about 23 minutes. Compared to the scheduling interval, the scaling and migration overheads are marginal.

7 CONCLUSION

In this paper, we proposed ElasticFlow, an elastic serverless computing platform for distributed training. ElasticFlow performs admission control based on minimum satisfactory share to guarantee deadlines. We developed a scheduling algorithm that dynamically allocates resources to admitted jobs based on diminishing returns. ElasticFlow applies buddy allocation to job placement to eliminate the effect of topology. The evaluation showed that ElasticFlow increased the number of jobs for which deadlines can be met by 1.46–7.65× over existing solutions. This work does not raise any ethical issues.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under grant numbers 62172008 and 62102009, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), the Beijing Outstanding Young Scientist Program under grant number BJJWZYJH01201910001004, the Microsoft University Collaboration Program, and the PKU-Baidu Fund Project under grant number 2020BD007.

We sincerely thank our shepherd Thaleia Dimitra Doudali and the anonymous reviewers for their valuable feedback on this paper. We acknowledge the Singularity team from Azure for sharing the job traces. Diandian Gu, Yihao Zhao, Yinmin Zhong, Gang Huang, Xin Jin, and Xuanzhe Liu are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. Diandian Gu and Yihao Zhao are co-primary authors, and Xuanzhe Liu is the corresponding author, of this work.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact provides source code for the prototype of the proposed system ElasticFlow, including the main implementation of ElasticFlow, testbed experiment scripts (§6.2 & §6.6), and cluster simulation scripts (§6.3 & §6.4 & §6.5). We provide a docker image with

pre-installed prerequisites to simplify the testbed experiment workflow. Users can also use a script to install all software dependencies from scratch. Please refer to the documents in our repository for more details.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** The experiments evaluate the effectiveness of the ElasticFlow algorithm. The baseline scheduling algorithms include EDF, Gandiva, Tiresias, Themis, Chronus, and Pollux.
- **Program:** `ElasticFlow/ElasticFlow/scheduler/scheduler.py` is the main program for evaluation. For more details, please refer to the documents in our repository.
- **Compilation:** The experiments require the compilation of protobuf and gRPC. Please refer to the README of our public code repository.
- **Data set:** We collect two-month real-world traces from ten Microsoft's internal ITP clusters. The collected traces are publicly available at https://github.com/microsoft/elasticflow-traces.
- **Run-time environment:** The simulation scripts require Python3.8. The testbed experiments require Docker. We provide a docker image with other dependencies pre-installed.
- **Hardware:** The simulation experiments can be conducted on a server or a laptop. The testbed experiments require up to 16 servers, each with eight NVIDIA A100 40 GB SXM GPUs, 96 AMD Epyc 7V12 (Rome) CPU cores, 900 GB RAM, and eight NVIDIA Mellanox HDR InfiniBand HCAs. NVMe is required to speed up the I/O process. The provided scripts can be executed on Azure Standard_ND96asr_A100 VMs [6].
- **Execution:** Scripts and commands for execution are provided in our repository. Please refer to the document.
- **Metrics:** The main metric of our experiments is Deadline Satisfactory Ratio, which is the ratio of jobs that can meet their deadlines.
- **Output:** The final results are printed in stdout. The details are recorded in CSV files.
- **How much disk space required (approximately)?:** The simulation experiments require about 5 GB of disk space for the logs. The testbed experiments require at least 160 GB NVMe storage on each node.
- **How much time is needed to prepare workflow (approximately)?:** A few hours.
- **How much time is needed to complete experiments (approximately)?:** The main results of simulation experiments (Figure 8(a)) take about half an hour. The rest of the simulation results take a few days. The testbed experiments of the Pollux baseline take about one day, and the testbed experiments of each of the other scheduling algorithms take a few hours. In total, it takes about a week to finish all of the experiments.
- **Publicly available?:** The code is publicly available at https://github.com/pkusys/ElasticFlow and the job traces are publicly available at https://github.com/microsoft/elasticflow-traces.
- **Code licenses:** Apache License 2.0.
- **Data licenses:** MIT License.
- **Archived:** https://doi.org/10.5281/zenodo.7481637

A.3 Description

A.3.1 How to Access. Clone the git repository at https://github.com/pkusys/ElasticFlow, including its submodules. An archived copy is also available [22].

A.3.2 Hardware Dependencies. The simulation experiments can be conducted on a server or a laptop. The testbed experiments can be executed on Azure Standard_ND96asr_A100 VMs [6]. Please adjust the scripts accordingly if the testbed has a different hardware configuration.

A.3.3 Software Dependencies. The simulation experiments require Python3.8. The testbed experiments require Docker. We provide a docker image with other dependencies pre-installed.

A.3.4 Data Sets. The job traces that we collected are publicly available [4]. Scripts are provided to parse the traces. The job traces for some other experiments can be automatically generated by the scripts that we provided. To download the datasets for DL training in testbed experiments, please refer to our documents.

A.4 Installation

The artifact can be downloaded and accessed as -

```
$ git clone --recursive https://github.com/pkusys/
ElasticFlow.git
$ cd ElasticFlow
```

A.5 Experiment Workflow

The detailed workflow, including configuring environments, executing the experiments, parsing the results, and plotting figures, is described in the documents of our repository.

A.6 Evaluation and Expected Results

The numbers of accepted jobs are printed in stdout, and the detailed results are recorded in CSV files. The log files can be found in the `ElasticFlow/plot_figure/logs/` directory. The reports can be matched against the figures reported in the paper.

REFERENCES

[1] 2019. NCCL. https://developer.nvidia.com/nccl. Retrieved on July 3, 2022.
[2] 2021. AWS EC2 pricing. https://aws.amazon.com/ec2/pricing. Retrieved on December 23, 2022.
[3] 2022. Amazon SageMaker. https://aws.amazon.com/sagemaker/?nc1=h_ls. Retrieved on July 3, 2022.
[4] 2022. ElasticFlow Traces. https://github.com/microsoft/elasticflow-traces. Retrieved on December 22, 2022.
[5] 2022. gRPC. https://grpc.io. Retrieved on July 3, 2022.
[6] 2022. ND A100 v4-series. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series. Retrieved on December 23, 2022.
[7] 2022. TorchElastic. https://pytorch.org/elastic/0.2.0/index.html. Retrieved on July 3, 2022.
[8] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016*, Vol. 48. 173–182.
[9] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. 2022. Varuna: Scalable, Low-cost Training of Massive Deep Learning Models. In *Proceedings of 17th European Conference on Computer Systems, EuroSys 2022*. 472–487. https://doi.org/10.1145/3492321.3519584
[10] Dimitri P. Bertsekas and Robert G. Gallager. 1992. *Data Networks, Second Edition*. Prentice Hall.
[11] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, K Jayaram, Michael Kalantar, Vinod Muthusamy, Priya Nagpurkar, and Florian Rosenberg. 2017. Scalable Multi-framework Multi-tenant Lifecycle Management of Deep Learning Training Jobs. In *Workshop on ML Systems, NeurIPS 2017*.
[12] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters For Deep Learning. In *Proceedings of the 15th European*

Conference on Computer Systems, EuroSys 2020. 1:1–1:16. https://doi.org/10.1145/3342195.3387555

[13] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. 2020. Elastic Parameter Server Load Distribution in Deep Learning Clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC 2020.* 507–521. https://doi.org/10.1145/3419111.3421307

[14] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A Comprehensive Study on Challenges in Deploying Deep Learning Based Software. In *Proceedings of 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020.* 750–762. https://doi.org/10.1145/3368089.3409759

[15] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012.* 1232–1240.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A Large-scale Hierarchical Image Database. In *Proceedings of 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2009.* 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019.* 4171–4186.

[18] Anis Elgabli, Jihong Park, Amrit S Bedi, Mehdi Bennis, and Vaneet Aggarwal. 2020. GADMM: Fast and Communication Efficient Framework for Distributed Machine Learning. *Journal of Machine Learning Research* 21, 76 (2020), 1–39.

[19] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2021. Chronus: A Novel Deadline-aware Scheduler for Deep Learning Training Jobs. In *Proceedings of the 12th ACM Symposium on Cloud Computing, Seattle, SoCC 2021.* 609–623. https://doi.org/10.1145/3472883.3486978

[20] Laurent George, Nicolas Rivierre, and Marco Spuri. 1996. *Preemptive and non-preemptive real-time uniprocessor scheduling.* Ph. D. Dissertation. Inria.

[21] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proceedings of 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019.* 485–500.

[22] gudiandian. 2022. *gudiandian/ElasticFlow: update traces.* https://doi.org/10.5281/zenodo.7481637

[23] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R Ganger, and Phillip B Gibbons. 2017. Proteus: Agile ML Elasticity through Tiered Reliability in Dynamic Resource Markets. In *Proceedings of the 12th European Conference on Computer Systems, EuroSys 2017.* 589–604. https://doi.org/10.1145/3064176.3064182

[24] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H. Campbell. 2019. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019.*

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition, CVPR 2016.* 770–778. https://doi.org/10.1109/CVPR.2016.90

[26] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of 33rd Annual Conference on Neural Information Processing Systems, NeurIPS 2019.* 103–112.

[27] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *Proceedings of 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021.* 721–739.

[28] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the Computation and Communication Abstraction Barrier in Distributed Machine Learning Workloads. In *Proceedings of 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2022.* 402–416. https://doi.org/10.1145/3503222.3507778

[29] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of 2019 USENIX Annual Technical Conference, ATC 2019.* 947–960.

[30] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. 2018. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research* (2018).

[31] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019.* 47–62. https://doi.org/10.1145/3341301.3359630

[32] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019.*

[33] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020.* 463–479.

[34] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019). https://doi.org/10.48550/arXiv.1902.03383

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of 26th Annual Conference on Neural Information Processing Systems, NeurIPS 2012.* 1106–1114. https://doi.org/10.1145/3065386

[36] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014.* 583–598.

[37] Mu Li, David G. Andersen, Alexander J. Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Proceedings of 28th Annual Conference on Neural Information Processing Systems, NeurIPS 2014.* 19–27.

[38] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. 2019. HyperSched: Dynamic Resource Reallocation for Model Development on a Deadline. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019.* 61–73. https://doi.org/10.1145/3357223.3362719

[39] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. 2020. Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2020.* 401–416. https://doi.org/10.1145/3373376.3378499

[40] Qinyi Luo, Jinkun Lin, Youwei Zhuo, and Xuehai Qian. 2019. Hop: Heterogeneity-aware Decentralized Training. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019.* 893–907. https://doi.org/10.1145/3297858.3304009

[41] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, ACL 2022.* 142–150.

[42] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *Proceedings of 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020.* 289–304.

[43] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter R. Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020.* 937–954.

[44] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020.* 481–498.

[45] Andrew Or, Haoyu Zhang, and Michael J. Freedman. 2020. Resource Elasticity in Distributed Deep Learning. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020.*

[46] Andrew Or, Haoyu Zhang, and Michael None Freedman. 2022. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022.* 126–140.

[47] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. In *Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015.* 5206–5210. https://doi.org/10.1109/ICASSP.2015.7178964

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of 33rd Annual Conference on Neural Information Processing Systems, NeurIPS 2019.* 8024–8035.

[49] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the 13th European Conference on Computer Systems, EuroSys 2018.* 1–14. https://doi.org/10.1145/3190508.3190517

[50] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2021. DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters. *IEEE Trans. Parallel Distributed Syst.* 32, 8 (2021), 1947–1960. https://doi.org/10.1109/

TPDS.2021.3052895

[51] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. 2018. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *Proceedings of 2018 USENIX Annual Technical Conference, ATC 2018*. 631–644.

[52] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *Proceedings of 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021*. 1–18.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI blog* 1, 8 (2019), 9.

[54] John E. Shore. 1975. On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *Commun. ACM* 18, 8 (1975), 433–440. https://doi.org/10.1145/360933.360949

[55] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of 3rd International Conference on Learning Representations, ICLR 2015*.

[56] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. 2019. Astra: Exploiting Predictability to Optimize Deep Learning. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019*. 909–923. https://doi.org/10.1145/3297858.3304072

[57] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. 2019. BLAS-on-flash: An Efficient Alternative for Large Scale ML Training and Inference?. In *Proceedings of 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019*. 469–484.

[58] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016*. 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[59] Jianyu Wang and Gauri Joshi. 2019. Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019*.

[60] Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. 2019. Neural Network Acceptability Judgments. *Trans. Assoc. Comput. Linguistics* 7 (2019), 625–641. https://doi.org/10.1162/tacl_a_00290

[61] Jinfeng Wen, pengpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An Empirical Study on Challenges of Application Development in Serverless Computing. In *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. 416–428. https://doi.org/10.1145/3468264.3468558

[62] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2022. Elastic Deep Learning in Multi-Tenant GPU Clusters. *IEEE Trans. Parallel Distributed Syst.* 33, 1 (2022), 144–158. https://doi.org/10.1109/TPDS.2021.3064966

[63] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. 595–610.

[64] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. 2020. Elan: Towards Generic and Efficient Elastic Training for Deep Learning. In *Proceedings of 40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020*. https://doi.org/10.1109/ICDCS47774.2020.00018

[65] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *Proceedings of 2017 USENIX Annual Technical Conference, ATC 2017*. 181–193.

[66] Xin Zhang, Jia Liu, Zhengyuan Zhu, and Elizabeth S Bentley. 2019. Compressed Distributed Gradient Descent: Communication-efficient Consensus over Networks. In *Proceedings of 2019 IEEE Conference on Computer Communications, INFOCOM 2019*. 2431–2439. https://doi.org/10.1109/INFOCOM.2019.8737489

[67] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*. 515–532.

[68] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of ACM SIGCOMM 2022 Conference, SIGCOMM 2022*. 428–440. https://doi.org/10.1145/3544216.3544224

[69] Yiren Zhao, Ilia Shumailov, Robert D. Mullins, and Ross Anderson. 2019. To Compress Or Not To Compress: Understanding The Interactions Between Adversarial Attacks And Neural Network Compression. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019*.

Received 2022-07-07; accepted 2022-09-22