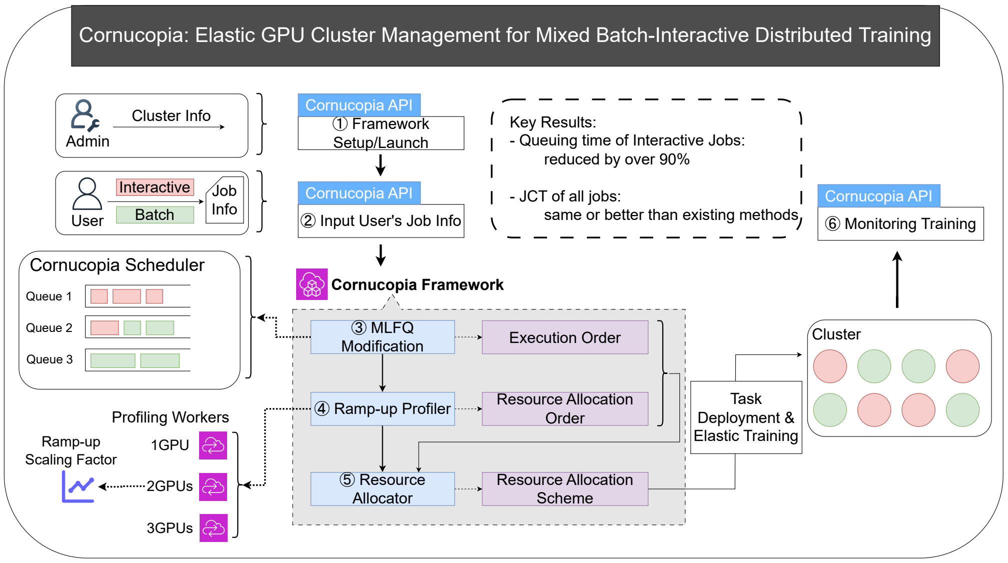
Graphical Abstract

Cornucopia: Elastic GPU Cluster Management for Mixed Batch-Interactive Distributed Training

Zimeng Huang, Shijian Li, Zihao Fan, Bo Jiang, Tian Guo



Highlights

Cornucopia: Elastic GPU Cluster Management for Mixed Batch-Interactive Distributed Training

Zimeng Huang, Shijian Li, Zihao Fan, Bo Jiang, Tian Guo

- We design *Cornucopia*, a priority-aware cluster manager for deep learning clusters. *Cornucopia* employs elastic training to achieve partial preemption/execution, thereby addressing resource utilization issues. The results show that *Cornucopia* can reduce the interactive queuing time by over 90% (t o within 1 second) while simultaneously ensuring that the JCT is same to or better than existing methods.

- We devise a resource allocation algorithm based on job resource acceleration ratios. For jobs with higher speedup ratios, we selectively allocate resources so as to meet the queuing time requirement of interactive jobs without compromising the job completing time (JCT) for all jobs.

Cornucopia: Elastic GPU Cluster Management for Mixed Batch-Interactive Distributed Training

Zimeng Huang[a,*], Shijian Li[b,*], Zihao Fan[a], Bo Jiang[a,**], Tian Guo[b,**]

[a]*Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, 201199, Shanghai, China*
[b]*Worcester Polytechnic Institute, 100 Institute Road, Worcester, 01609, Worcester, America*

Abstract

The ability to quickly train deep learning (DL) models plays an important role in accelerating the model design process. DL practitioners often need to submit trial-and-error jobs when designing new models or debugging existing models; once the model architectures and the hyperparameters are set, practitioners will then train those models to convergence for inference deployment. Therefore, these trial-and-error jobs often need to be interactive as DL practitioners are actively working on the model design. However, existing GPU cluster managers often assume that training jobs are batch-oriented, leading to very long queueing time for interactive training jobs. To cater to a mix of interactive and batch training workloads, optimizing for both queueing time and job completion time (JCT) simultaneously, we have designed a new cluster manager called *Cornucopia*. We leverage a new feature called elastic training, which allows a job to be partially preempted to release GPU resources to a higher-priority job to start execution. Specifically, *Cornucopia* consists of three main components: a scheduler that determines the job queue based on priorities, a profiler that assesses the resource acceleration ratios of jobs, and an allocator that makes decisions regarding resource allocation and preemption. Via large-scale trace-driven simulations, we show that *Cornucopia* reduces the queuing time of interactive jobs by up to 90%

*Both authors contributed equally to this work.
**Corresponding Authors.
Email addresses: `lukehuang@sjtu.edu.cn` (Zimeng Huang), `sli8@wpi.edu` (Shijian Li), `fzhsjtu2023@sjtu.edu.cn` (Zihao Fan), `bjiang@sjtu.edu.cn` (Bo Jiang), `tian@wpi.edu` (Tian Guo)

while ensuring JCT compared to several recent GPU schedulers.

Keywords: distributed training, GPU cluster, elastic training, job scheduling

1. Introduction

Given the relentless growth in the complexity and scale of deep neural networks (DNNs), distributed training leveraging multiple GPU workers has become a standard practice [1, 2, 3]. To scale DNN training across multiple GPUs, data parallelism and model parallelism are commonly used. For large models such as LLMs, more advanced strategies, such as tensor, pipeline, expert, and sequence parallelism, are adopted to scale training across more GPUs and further accelerate performance [4, 5]. In response to this development, the predominant preference among researchers and industry specialists is to employ GPU clusters to facilitate a range of distributed training jobs. These jobs are dispatched to the cluster manager, which is responsible for the allocation of hardware resources and job execution sequence [6, 7, 8, 9, 10, 11].

Drawing on insights from a recent user survey [6] and trace analysis [12, 13], it has been observed that user-interactive jobs occupy a substantial share of the workload in real-world production environments. Users often expect these jobs to be prioritized with minimal wait time for their execution. To better understand practical cluster scheduling, we differentiate two types of jobs: *(i) interactive jobs*: these jobs are of a debugging or exploratory nature, and users anticipate handling such jobs as soon as possible; *(ii) batch jobs*: these jobs pertain to regular model training, and users expect the job completion time (JCT) to be minimized.

However, existing cluster managers have not sufficiently taken into account the distinct scheduling requirements of interactive jobs. Such an oversight can result in delays in scheduling of interactive jobs, subjecting users to prolonged waiting periods. Concurrently, naive scheduling approaches also pose challenges: if a simplistic prioritization scheme is implemented, whereby interactive jobs are assigned a higher priority than batch jobs, this could inadvertently prolong the JCT for batch jobs. Consequently, there is a pressing need to devise an innovative cluster manager that is capable of accommodating the divergent requirements of batch and interactive jobs. Overall, we face two problems at the design level:

First, job-level granularity in resource scheduling can lead to wastage of resources. Due to the preemption and allocation of resources being conducted on per-job-level granularity, the residual resources post-allocation to high-priority jobs may prove insufficient to meet the execution demands of other tasks awaiting in the queue. This scenario could potentially result in the underutilization of GPU resources.

Second, low-priority jobs suffer JCT dilation when preempted by high-priority jobs. When a hierarchy of job priorities is established, jobs with lower priority invariably receive a subordinate position in the resource allocation sequence. This demotion in scheduling priority, relative to a non-priority approach devoid of such differentiation, may result in a deterioration of the JCT for those jobs. As we will demonstrate in §3.2, the average JCT of the cluster's workload can be increased by 7.2X.

Our key design to address the above problems is the utilization of *elastic training* to handle preemption and *scalability-based resource allocation* to ensure the JCT performance in priority job scheduling. Elastic training is a recent technique that enables dynamic adjustment of the resources employed *during* the training process. Currently, major DL training frameworks, such as PyTorch[14], Horovod[15], PaddlePaddle[16], and MXNet[17], all support elastic training. We address the issue of resource wastage by leveraging elastic training to refine the granularity of resource scheduling from the level of individual jobs to that of single GPU units. This is achieved by employing a strategy of *partial preemption*, where only a fraction of a job's resources are preempted, and *partial execution*, which allows jobs to execute with a dynamically adjustable subset of their total resource allocation. Furthermore, we have devised a resource allocation algorithm informed by observations of the resource-to-speedup ratio for different jobs, allowing for selective resource skewing, with the aim of enhancing the JCT performance.

Those key designs are manifested in the implementation of *Cornucopia*, a cluster manager for shared GPU clusters supporting elastic training. *Cornucopia* focuses on optimizing interactive queuing time and JCT during the scheduling process. *Cornucopia* consists of three key components: *(i)* a multi-level feedback queue (MLFQ) *(ii)* a ramp-up scalability profiler, and *(iii)* an elastic allocator that supports partial preemption and execution. The MLFQ is used to manage training jobs with different priorities, where low-priority jobs are subject to preemption and a potential degradation of JCT. To address the preemption and JCT issue, we design the ramp-up job-scalability profiler and implement the scaling-aware elastic resource allocating algorithm to favor resource allocations to jobs with better speedup.

We evaluate *Cornucopia* by trace-driven simulations using two real-world traces, i.e., Microsoft's Philly trace [18] and SenseTime's Helios trace [12]. In comparison with state-of-the-art methods, *Cornucopia* has achieved a reduction of over 90% in the queuing time for interactive jobs, while achieving the same or shorter average JCT for all jobs. We observed that using *Cornucopia* on the Philly trace could reduce the JCT by up to 77% compared to existing methods.

We summarize our main contributions as follows:

- We design *Cornucopia*, a priority-aware cluster manager for deep learning clusters. *Cornucopia* employs elastic training to achieve partial preemption/execution, thereby addressing resource utilization issues. We will open-source *Cornucopia* upon acceptance.
- We devise a resource allocation algorithm based on job resource acceleration ratios. For jobs with higher speedup ratios, we selectively allocate resources so as to meet the queuing time requirement of interactive jobs without compromising the JCT for all jobs.
- We conduct extensive evaluation of *Cornucopia* by large-scale trace-driven simulations. The results show that *Cornucopia* can reduce the interactive queuing time by over 90% (to within 1 second) while simultaneously ensuring that the JCT is same to or better than existing methods.

The rest of the paper is organized as follows: §2 provides a comprehensive exploration of the background about distributed DL training and the current status of elastic DNN training. §3 elucidates the statement of the problem and expounds the corresponding challenges. §4 delineates the specific design details of *Cornucopia* and §5 presents the evaluation results under different workloads. §6 summarizes other works related to the research presented in our paper. §8 provides a conclusion for the entire paper.

2. Background

2.1. Distributed DL training

Distributed DL training accelerates the training process of large-scale DNN models by utilizing multiple GPUs[19, 20, 21]. When employing a distributed strategy for model training, users must delineate the specific distributed training approach from two dimensions: parallelism and synchronization.

Parallelism. Distributed training can be broadly categorized into *data parallelism* and *model parallelism*. In data parallelism, each GPU holds a full replica of the model and processes a distinct shard of the training dataset [19, 22]. In model parallelism, the model is partitioned across GPUs, with each GPU storing and computing only a subset of the parameters [23, 24, 25, 26]. For large-scale models such as LLMs, more advanced strategies, including tensor, pipeline, expert, and sequence parallelism, are often combined to scale training to thousands of GPUs and further improve throughput [4, 5, 27, 28].

Synchronization. During the distributed training process, parameter updates after each batch can be achieved using either synchronous[29, 23] or asynchronous[30, 24] methods. Synchronous distributed training synchronously aggregates the gradients from each worker after each iteration and updates model parameters based on the average gradient. Popular distributed frameworks often employ a Parameter-Server (PS) [31, 32] or Allreduce[33, 34] architecture to facilitate gradient communication during the training process. Asynchronous distributed training, on the other hand, does not require all workers to synchronize their operations during parameter updates to achieve faster training speeds but may come at the cost of reduced training accuracy. In this paper, we primarily focus on discussing synchronous distributed strategies, as it is the most commonly employed communication strategy nowadays[12].

2.2. Elastic DNN training

Elastic DNN training refers to the technique of dynamically adjusting the amount of computing resources (such as GPUs) during the training process of a DNN model. In recent years, some research efforts have been dedicated to the field of elastic DNN training[35, 36, 37], and thanks to these advancements, the application of elasticity in distributed training platforms has become feasible[14, 15, 17, 16]. One significant advantage of employing elasticity in cluster scheduling is the ability to efficiently utilize surplus resources generated by preemption or allocation within the cluster, thereby enhancing overall resource utilization [38, 39]. Additionally, elasticity permits the allocation of idle resources to certain tasks in low workload situations, accelerating their execution [10, 11]. Note that *Cornucopia* does not

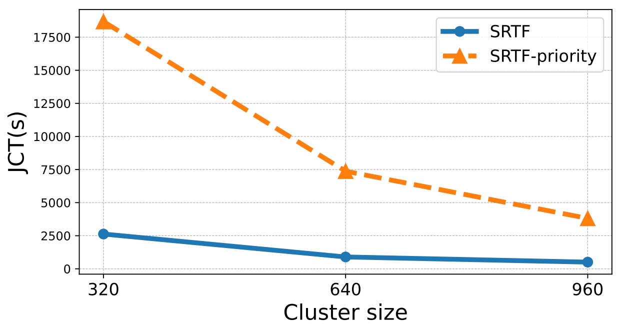


Figure 1: **JCT escalation due to priority preemption on Philly.** In SRTF-priority scheduling, we consistently preemptively prioritize the execution of interactive jobs.

primarily focus on accelerating specific tasks. Instead, *Cornucopia* is dedicated to mixed-type scheduling within a cluster and leverages elastic training to facilitate scheduling at a finer granularity, thereby enhancing the overall performance.

3. Problem Statement

This paper looks at the key problem of how to efficiently schedule a mixed batch-interactive DL workload in a shared GPU cluster. In particular, we consider a cluster scheduling scenario where both interactive and regular batch jobs co-exist and leverage elastic training to design tailored job scheduling and resource allocation strategies. We consider two key metrics: queuing time for interactive jobs that require prompt responses and completion time for all jobs.

3.1. Problem Scope

We consider a multi-tenant GPU training cluster, where users can submit DL training jobs at any given time. These jobs can be of a debugging or testing nature or require long training time to converge. Upon job submission, users are required to manually categorize the job type as either *interactive* or *batch* and provide the corresponding training script. To accommodate different parallel mode jobs incompatible with elastic training (such as model parallelism), we also permit users to submit a *non-elastic* label when submitting their jobs. For *non-elastic* jobs, the scheduler will allocate resources exactly according to user-specified resource requirements.

We further make the following assumptions regarding user input and training environment: *(i)* **There is no inter-job dependency for most jobs.** This means our work focuses on handling data-parallel jobs rather than model parallelism. The primary reason for this is the lack of support for model-parallel in elastic training. However, we can still handle model-parallel jobs by treating them as *non-elastic*. *(ii)* **Each submitted job is associated with a corresponding label.** The label is assumed to be provided by the user and *Cornucopia* does not accept unlabeled jobs. *(iii)* **Jobs run on a homogeneous set of GPUs.** *Cornucopia* supports deployment on heterogeneous GPU clusters; however, each job is restricted to run on a homogeneous set of GPUs. This follows the design of mainstream cloud PaaS services such as AWS SageMaker [40] and Alibaba Cloud PAI [41], where users specify the GPU type at job submission. There are two main reasons for this design: (1) running a single job across heterogeneous GPUs may lead to stragglers and synchronization delays due to performance imbalance; (2) in real-world large-scale clusters, different GPU types often reside in isolated interconnect domains without high-speed links. Cross-type scheduling typically falls back to low-bandwidth virtual private cloud paths, leading to poor efficiency. Therefore, elastic scaling and preemption in *Cornucopia* are constrained within the user-specified homogeneous GPU type. Extending support for heterogeneous execution is left for future work.

3.2. Challenges of DL Scheduling

In designing *Cornucopia*, we address the following three key challenges. We will elaborate on key designs of *Cornucopia* in §4.

Challenge #1: Unpredictable job duration. Existing scheduling algorithms either require precise job duration inputs [42, 43] or need to obtain this value through predictive methods [6, 7, 44]. Unfortunately, we often don't know DL job duration ahead of time. Further, it can be challenging to predict DL job duration due to the uncertain convergence time. Moreover, users can provide an early stopping condition, reducing the expected job duration. This means that the traditional scheduling policies such as Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) cannot apply. This also means that the using of performance models [44, 7, 45] to predict job length is not always feasible.

Challenge #2: Nonlinear job scalability. Previous studies have shown that the resource speedup ratio for deep learning jobs is often nonlinear [46]. On the one hand, there is lower communication overhead for GPUs

Trace Set	Linear Job Num	Sublinear Job Num	Sublinear GPU Num	Avg sub GPU per sub job
Philly	20764	176	872	4.95
Helios:Venus	10902	122	3752	30.75
Helios:Earth	63706	57	960	16.84
Helios:Saturn	69433	254	7992	31.46
Helios:Uranus	46259	259	6592	25.45

Table 1: **Number of sublinear jobs and GPUs in different trace sets.** We assume that GPU resources are allocated to jobs based on their requirement during the scheduling process. Jobs that do not scale proportionally with additional GPUs are termed sublinear, and the allocation exceeding the point of proportional performance gains is referred to as sublinear GPU nums.

on the same server node than inter-node setup. On the other hand, the scalability of models differs significantly due to different structures, which means that distinct models exhibit disparate scalability characteristics. However, the majority of existing scheduling algorithms do not consider the varying resource acceleration ratios of jobs in the actual scheduling process[6, 7, 9]. Instead, they merely allocate resources based on user-defined values, potentially resulting in a significant allocation of resources to sublinearly accelerating jobs, as illustrated in Table 1. Based on our statistical analysis, when disregarding the resource acceleration ratios of jobs, it becomes apparent that each sublinear job can lead to an average non-linear acceleration of 5 to 30 GPUs, which may result in potential resource wastage.

Challenge #3: Performance costs of priority preemption. As shown in Figure 1, preempting low-priority jobs may result in significant degradation of their JCT performance and hence the overall average JCT. Beyond the implications for individual job performance, the presence of potential preemptions at any given moment introduces a complex dynamic that can result in resource fragmentation within the cluster. This fragmentation, in turn, has broader repercussions, impacting the overall efficiency of resource utilization across the cluster environment and the JCT metric along all jobs. Thus, addressing the interplay between task priority, preemption, and resource allocation becomes paramount in achieving optimal performance and resource management in such scheduling systems.

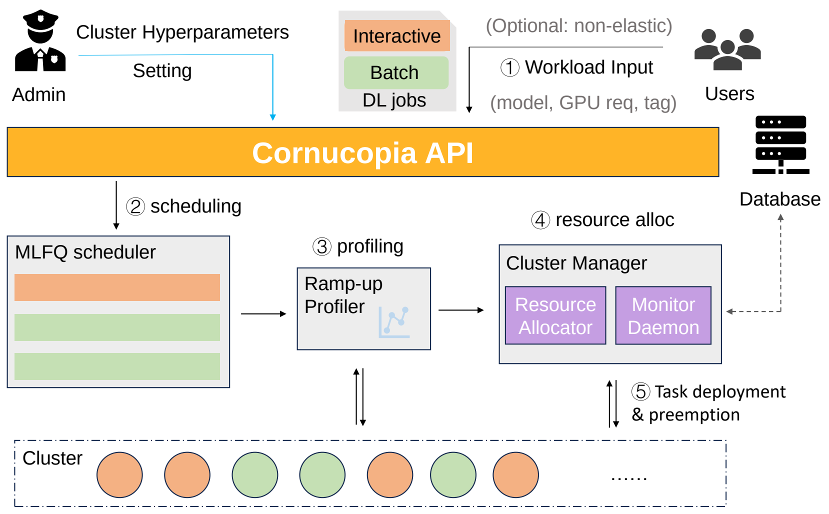


Figure 2: ***Cornucopia* architecture and workflow.** *Cornucopia* consists of three components: the scheduler is responsible for determining when jobs should run, the profiler provides resource acceleration ratio info for jobs, and the allocator performs the resource allocation for each job in the cluster. We also provide a monitor daemon that continuously monitors the overall cluster status in real time.

4. Cornucopia Design

In this section, we present a DL cluster management framework called *Cornucopia*, which consists of three key components Multi-level-Feedback-Queue (MLFQ)-based scheduler (§4.2), the elastic-training-enabled profiler (§4.3), and the scaling-sensitive resource allocator (§4.4).

4.1. Cornucopia *Overview*

Cornucopia is designed to allow users that share a GPU cluster to enjoy both short queuing time and good JCT by taking advantage of the recently introduced elastic training mechanism [14, 15]. To schedule a job, we consider the following key questions: *(i)* When should we run it? *(ii)* In what order should we allocate or preempt resources for it? *(iii)* How much resources should be allocated to it each time?

Cornucopia is designed with three main components to address the above questions, as shown in Figure 2. The *scheduler* takes inputs from the cluster users, including GPU requirements, model training scripts, corresponding job types, and the optional *non-elastic* tag to handle different workloads. It then schedules the jobs using a MLFQ-based approach. The *profiler* profiles

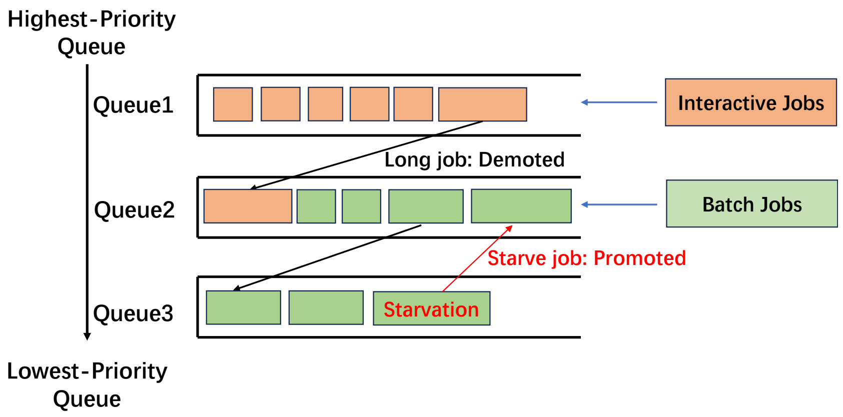


Figure 3: **The scheduling policy.** The policy is based on multi-level feedback queue. Different jobs are placed into different queues, according to their characteristics. We also implement dynamic job promotion and demotion strategies, as well as real-time preemption policies among different queues.

job throughput at runtime with low overhead thanks to the elastic training mechanism, and high profiling fidelity via a ramp-up approach. The throughput profiles will be used to model each training job's performance, which *Cornucopia* will leverage to decide how many GPUs to allocate for each job. The resource *allocator* coordinates the resource allocation for each job, guided by the scaling factor supplied by the profiler and the scheduling decisions offered by the scheduler.

Cluster administrators also need to configure a series of scheduler hyper-parameters, including the MLFQ task promotion and demotion thresholds. Moreover, *Cornucopia* conducts real-time monitoring and data collection of the cluster's status, allowing users to consistently track the comprehensive workload and resource utilization of the cluster. In the following sections, we will provide a detailed overview of the various components of *Cornucopia*.

4.2. Priority-based Scheduler

For scheduling, we adopted a multi-level feedback queue (MLFQ) based approach shown in Figure 3. We made this choice because MLFQ is a classic and valuable strategy used in priority scheduling scenarios, and our targeted cluster workload naturally comes with different priorities (i.e., interactive and batch training jobs). Another advantage of MLFQ is its ability to directly determine a job's scheduling queue based on the job's own priority, eliminating the need for continuous job duration prediction to decide the job's initial queue, which naturally addresses the first challenge mentioned in § 3.2. MLFQ is known for reducing the response time of jobs [47]. Moreover, the multiple queues in MLFQ provide the intuitive functionality to handle jobs with different characteristics.

To handle both interactive and batch jobs, we use three queues. The first two queues are for handling job priorities, and the last queue is for mitigating job starvation. Thus, for interactive jobs, we place them initially into the highest priority Queue 1. Queue 2 will be for the normal batch jobs.

To use MLFQ, we also need to configure three hyperparameters, corresponding to the demotion thresholds from Queue 1 to Queue 2, from Queue 2 to Queue 3, and the promotion threshold from Queue 3 to Queue 2. Departing from traditional MLFQ, we determine these hyperparameters by dynamically monitoring the actual job durations, and we set initial values for these hyperparameters as twice the average duration of interactive-type jobs, the average Runtime-to-failure (RTTF), and twice the average duration of batch-type jobs, respectively. We also devise online dynamic adjustment strategies for these parameters. Specifically, *Cornucopia* collects label information and completion time information for each job it processes and recalculates the mean values to update the hyperparameters every 1000 jobs.

In our MLFQ, we devise policies that govern aspects such as resource preemption and the promotion and demotion between the queues as follows:

Demotion policy: Queue 2 jobs will be demoted to Queue 3 once their accumulated executed time hits a threshold. We set this threshold to twice the RTTF based on the real-world traces [12], but the threshold can be updated in an online manner to adapt to new workloads. Furthermore, to prevent certain batch jobs from being mistakenly categorized as interactive jobs by users, we have designed a demotion strategy from Queue 1 to Queue 2. When an interactive job runs longer than twice the average interactive job duration, it is demoted from Queue 1 to Queue 2.

Promotion policy: Queue 3 jobs can be promoted to Queue 2 if they have accumulated enough queuing time to avoid starvation. This accumulated time only counts when they are completely stripped of all their resources, halted, and put back into the queue.

Preemption policy: When a new task is introduced into the high-priority queue, our scheduler enables task preemption among existing tasks. During the preemption process, Queue 1 also holds the highest priority, while Queue 3 has the lowest priority, implying that jobs in Queue 1 will not undergo preemption.

4.3. Elasticity-Aware Ramp-up Profiler

A practical problem for deep learning jobs is scalability, which determines the effective throughput of the training session. Distributed deep learning model training cannot scale infinitely with more GPU resources. As previous studies have pointed out, distributed deep learning jobs usually can achieve linear speedup before hitting a bottleneck [32]. However, after a distributed job reaches this *acceleration bottleneck*, the benefits of adding additional resources diminish. Therefore, in deployment, knowing how each job scales is beneficial. One way is to run pilot jobs for a period of time and measure the corresponding throughput. However, this method can be time-consuming in an elastic setting, as we will need to run as many pilot jobs for one model as the number of GPU resource combinations. Instead, we leverage the elasticity property and design a ramp-up profiler that imposes minimal measurement overhead, addressing the second challenge in § 3.2.

Concretely, instead of giving all the resources needed at once, we gradually add in resources to record the training throughput for this job. Because elastic training incurs less than 10 seconds of overhead when adding/removing resources based on our measurements of Horovod, the ramping-up overhead for a distributed training job with tens of GPUs is only a few minutes. Therefore, our ramp-up profiler provides the ability to profile ground truth training throughput in real-time with low overhead. Doing so is beneficial for two reasons: *(i)* avoiding model-based prediction errors, *(ii)* mitigating overhead that impacts training throughput or scheduling performance.

Since we collect a stable training throughput to establish the profiles, we need to make sure the training stabilizes. Our observation shows that the training time of the newly initiated GPU stabilizes after a few initial batches. This suggests that only a few batches are required to calculate the stable training throughput. Through empirical measurements, we have set the profiling duration to 5 batches: the first two batches are designated as warm-up batches, and the subsequent three batches offer reasonably reliable estimates. The total profiling overhead is directly proportional to the current GPU size in operation. Empirically, we observe that the per-GPU throughput remains highly stable during profiling, with fluctuations consistently under 1% relative to the mean, confirming the robustness of our measurement methodology.

Algorithm 1 Ramp-Up Throughput Profiler

```
1: Input $G_i$: number of GPUs allocated to job $i$
2: for $G_i^{current} \leftarrow 1$ to $2G_i$ do
3:     Train for 5 batches
4:     Average and cache the training throughput $T_i^{current}$ with $G_i^{current}$ number of GPUs
5: end for
6: Output $T_i$: vector for job $i$'s training throughput profiles
```

To better facilitate resource allocation and preemptive decision-making for each job during the resource allocation phase, we collect profile data up to twice the number of GPUs requested by the user. This gives the cluster manager the flexibility to increase the GPU resources for the training jobs when the conditions are right, e.g., when there are idle resources in the cluster or when a job with lower priority has better projected JCT and, therefore, can be a good candidate for preemption. The pseudocode for the ramp-up profiler is shown in Algorithm 1.

After obtaining the training throughput profile T_i^j for job i under j GPUs, we calculate the scaling factor S_i^j for this job under j GPUs allocation using the following formula:

$$S_i^j = \begin{cases} 1, & \text{if } j = 1 \\ \frac{T_i^{j+1} - T_i^j}{T_i^2 - T_i^1}, & \text{if } j > 1 \end{cases} \tag{1}$$

which quantifies the differential gains in throughput experienced by various jobs when incremental resources are allocated. Note that if a job is marked as *non-elastic*, its throughput will not be evaluated during the profiling process. To enable the comparison of scaling factors for such jobs with those of other jobs, we set the scaling factor for *non-elastic* jobs to an initial value of 1.

Profiling Overhead. With using elastic ramp-up profiling, even if the job is running hundreds of GPUs, the overhead would still be similar to or smaller than restarting the training session. For example, with batch size 32, ImageNet dataset, and ResNet-101 used in the empirical experiment, profiling based on our method for a 64-GPU job's total time cost is about 314 seconds. This is similar to restarting the training session 6 times, as reported by prior work [36]. This means if using a checkpoint-based approach, we could only collect 6 data points in this period of time. Moreover, since our training session keeps running throughout the ramping period, the actual loss of work is much smaller. This overhead is only needed for the first time the job is profiled (up until twice the actual number of GPUs in use) and will not require repeated profiling. Moreover, the overhead of hundreds of seconds is small compared to the total execution time of training jobs, which typically ranges from tens of thousands to hundreds of thousands of seconds.

4.4. Resource Allocator

In the resource allocator module within the cluster manager, we will integrate the scaling factor obtained from the profiler and the scheduling priorities provided by the scheduler to consider resource allocation and job preemption in a holistic job-centric perspective. In addition to the conventional on-demand resource allocation, we have leveraged the elasticity training mechanism to implement *partial preemption* and *partial execution* of jobs. This contributes to the enhancement of the cluster's overall resource utilization. Furthermore, judiciously allocating GPU resources to jobs with linear speedup ratios will further enhance JCT for low-priority jobs, addressing the third challenge mentioned in § 3.2.

If there are still available resources in the cluster, our first consideration is how to allocate resources for jobs in priority queues from §4.2. One straightforward approach is to greedily allocate GPUs based on the job's acceleration ratio under the current resources. That is, we always allocate resources to the job corresponding to the maximum scaling factor among all jobs within the same priority level. However, considering that interactive jobs often have a testing nature, we initially attempt to allocate GPUs for interactive jobs as requested by users. For batch jobs, we employ the aforementioned greedy allocation approach. Intuitively, this greedy algorithm helps the batch jobs to be completed sooner and then releases corresponding GPU resources for other jobs. The overall allocation pseudocode is outlined in Algorithm 2. Specifically, the allocator iteratively traverses the job list sorted by priority. For each interactive job, it attempts to fulfill its full GPU request if sufficient resources are available; otherwise, it assigns the remaining GPUs. For batch jobs, it identifies the job with the highest scaling factor within the same priority level and allocates one GPU at a time to that job, aiming to accelerate jobs with better parallel efficiency. The loop terminates when all GPUs are allocated.

If the current cluster has no idle resources, when higher-priority tasks are added, we will resort to resource stealing for these jobs. In particular, we

Algorithm 2 Greedy-Based Allocation Policy

```
1: Inputs: a set of jobs $J$ in priority order with GPU requirement $R$ and
   scaling factor $S$, remain GPU num $W$
2: while $W > 0$ do
3:   if $W == 0$ then
4:     break
5:   end if
6:   for job $j_i$ in $J$ do
7:     if $j_i$ is interactive then
8:       if $R_i \leq W$ then
9:         Allocate $R_i$ GPUs to $j_i$
10:        Remove $j_i$ from $J$
11:      else
12:        Allocate $W$ GPUs to $j_i$
13:        Update $R_i \leftarrow R_i - W$
14:      end if
15:    else if current scaling factor of $j_i$: $S_i$ is maximum in $J$ then
16:      Allocate 1 GPU to $j_i$
17:    end if
18:    Update $S_i, W$
19:  end for
20: end while
```

enable resource stealing by preempting resources assigned to existing jobs to different degrees. Thanks to the robustness of data parallelism, we can reduce the GPU resource consumption during the training process without having an adverse effect. Thus, we adopted the approach of *fine-grained GPU resource preemption* instead of the traditional job-level preemption. For a job, we can take away part of its resources without terminating them entirely. This can be done via the support of elastic deep learning frameworks such as Elastic Horovod [15]. However, when jobs with model parallelism are submitted to *Cornucopia*, we still need to adjust the resources at the job level since the current elasticity training mechanism does not support these parallel strategies.

To select the job to preempt, we consider the impact of stripping away resources. We choose the job whose training throughput will have the least impact. This impact is determined by the scaling factor of the jobs. For

Algorithm 3 Scaling-Aware Preemption Policy

```
1: Inputs: a high priority job $j_{high}$ with GPU requirement $req_{high}$, a set of
   low priority jobs $J$
2: while $req_{high} > 0$ do
3:   for job $j_i$ in $J$ do
4:     if Current scaling factor of $j_i$: $S_i$ is minimum in $J$ then
5:       Preempt 1 GPU from $j_i$
6:       Update $S_i, req_{high}$
7:       Provision 1 GPU to $j_{high}$
8:     end if
9:   end for
10: end while
```

mixed scaling factors where sub-linear and linear jobs both are present, we choose the sub-linear jobs first. This impact is based on the profiled scaling factors of the jobs. The pseudocode for the scaling-aware preemption policy is detailed in Algorithm 3. The algorithm iteratively selects the low-priority job with the lowest scaling factor, revokes one GPU, and reallocates it to the high-priority job. This process repeats until the high-priority job's demand is met. In the special case of linear scaling jobs, the preemption is sorted in descending order of GPU resources.

5. Evaluation

In this section, we first present the interactive queuing time and end-to-end JCT results of *Cornucopia* by comparing them with state-of-the-art cluster schedulers(§5.2 and §5.3). We then evaluate *Cornucopia*'s robustness by studying the impacts of mislabeled job types.

5.1. Evaluation Setup

We evaluate the performance of *Cornucopia* with a discrete trace-driven simulator written in Python. The scheduler part is based on the implementation of the simulator in Tiresias [7].

Workloads. To reflect the characteristics of deep learning jobs in a realistic production cluster, we use two popular real-world deep learning traces, namely the Helios trace from SenseTime [12] and the Philly trace from Microsoft [18]. Since they are large traces that expand for a long period, we

Method	Tiresias [7]	ElasticFlow [10]	Synergy [9]	Chronus [6]	SRTF	Cornucopia (Ours)
Cluster Environment	GPU cluster	Serverless	GPU cluster	GPU cluster	GPU cluster	GPU cluster
Prior Knowledge	None	None	None	None	Job Duration	None
Scheduling Algorithm	2D-LAS	MSS	LP optim	Lease based	SRTF	MLFQ
Elastic Feature	×	✓	×	×	×	✓
Interactive-aware	×	×	×	✓	×	✓

Table 2: **Cluster scheduling baselines *Cornucopia* is compared against.** *Cornucopia* does not require prior knowledge about training jobs and is the only work that leverages the elastic feature and handles interactive jobs.

use subsamples of these traces from shorter time windows. We adopt the same time windows used for these two traces in a recent work Chronus [6]. Specifically, the time window for the Helios trace is April 14th to April 27th, 2020, and October 12th to October 25th, 2017 for the Philly trace. [1]

We label the jobs as *interactive* or *batch* based on the following strategy for both Philly and Helios traces. For *Completed* jobs, we label them as normal *batch* jobs; for *Cancelled* jobs, we label them as *"interactive"* jobs; for *Failed* jobs with a duration of less than 20 minutes in Helios or 10 minutes in Philly, we label them as *interactive* jobs[2]; and for *Timeout* and *Node_fail* jobs in the Helios trace set, we label them as *batch* jobs.

Cluster setup. We use variable cluster sizes to measure scenarios where potential under-provisioning could happen. In particular, we chose four sets of commonly used GPU cluster sizes: 256, 320, 640, and 960. Each cluster consists of 32, 40, 80, and 120 nodes, with each node equipped with 8 GPUs, in line with the prevalent configuration of existing clusters[48, 20]. Considering the job arrival frequencies and workload sizes of different trace sets, we use different settings for them. For the Philly and Saturn trace set, we conducted evaluations using three cluster sizes: (320, 640, and 960). In the case of the Venus trace set, we conducted evaluations using three cluster sizes: (256, 320, and 640).

Baselines. We compare the performance of *Cornucopia* to four recently published GPU scheduler works and a classic scheduling algorithm. We chose these five works because they exhibit different characteristics, as detailed in

[1]We observed that there are discrepancies between the real traces and the ones that are used and shown in Chronus. In the subsequent evaluation, we use the Philly trace from Chronus directly and reproduce the Helios trace used according to the details described in their paper.

[2]The time threshold here is obtained through the average execution time profile of different types of jobs.

Table 2. Due to the requirement of deadline-aware baselines for the job's deadline input, we set the corresponding inputs of these baselines to infinity for comparison with others.

1. Tiresias[7]: designs a two-dimensional scheduling algorithm based on spatial and temporal aspects to minimize average JCT. However, Tiresias does not consider elastic scaling and job priority.

2. ElasticFlow[10]: leverages elastic mechanisms to implement the scheduling of ddl-aware DL jobs on serverless platforms. It is elastic but does not consider priority.

3. Synergy[9]: performs joint scheduling optimization for DL tasks involving GPU, CPU, and memory resources. It is not elastic and not priority-aware. We solely compare the results of Synergy on the Philly set since it has only modeled memory usage for the Philly trace set.

4. Chronus[6]: designs a deadline-aware scheduling algorithm for jobs with different priorities. It is not elastic.

5. Shortest-Remain-Time-First (SRTF): is a classic preemptive job scheduling algorithm but the duration of jobs is often challenging to predict in DL scenarios. In the evaluation, we assume that the SRTF algorithm has prior knowledge of the duration of all jobs and use it as a performance reference. It is not elastic.

Metrics. Recall that *Cornucopia* has two primary design objectives: *(i)* to provide prompt responsiveness for interactive jobs and *(ii)* to expedite the completion of all jobs as much as possible. We choose the following metrics to evaluate the performance of job scheduling in *Cornucopia*:

- *Queuing time for interactive jobs*: Queuing time measures the total time a job waits in the queue without being executed. In our work, we don't consider a partially running job to be waiting. This metric evaluates the effectiveness of the scheduler in handling interactive jobs. The shorter the queuing time is, the better the performance is. We conducted a detailed analysis of the variations in this metric under different scheduling algorithms in §5.2.

- *Job completion time (JCT)*: JCT measures the length of a job from entering the queue to the time it is completed. We use this metric to evaluate the efficiency of the scheduling. The shorter the JCT is, the better the performance is.

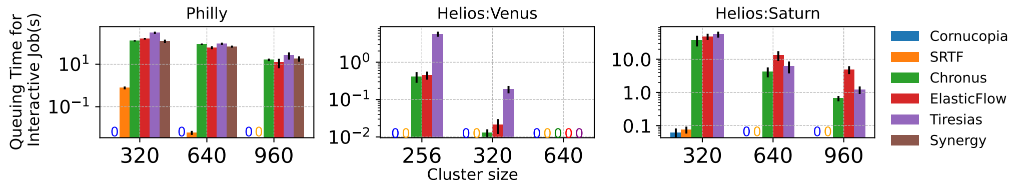


Figure 4: **The average queuing time for interactive jobs.** Note that we employ Synergy as the baseline only for the Philly traces, as Synergy only models the job memory consumption within Philly. A value of 0 in the figure indicates that the corresponding method achieves zero average queuing time.

- *JCT inflation*: To visually compare the scheduling outcomes with the job execution time, we also use the *JCT inflation* as our metric, which is defined as the ratio of the post-scheduling JCT to the job's own duration, as shown in Equation (2).

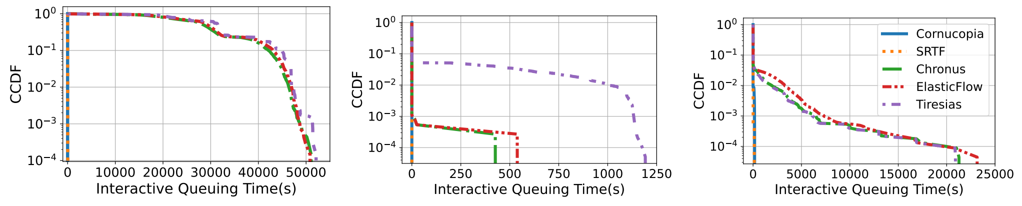
$$JCT\ inflation = \frac{JCT}{job\ duration} \tag{2}$$

  In an ideal scenario without elasticity, the optimal value for JCT inflation is 1. However, in cases of resource abundance and the utilization of elastic training capabilities, this value can be less than 1. The smaller the JCT inflation is, the better the performance is. We analyzed the differences in JCT inflation under various scheduling algorithms in §5.3.

5.2. Queuing time for Interactive Jobs

We first focus on the differences in interactive job queuing time under different scheduling methods. As shown in Figure 4, we compare the queuing time results for interactive jobs under *Cornucopia* and various baseline scheduling algorithms in Philly and Helios. It is evident that the utilization of *Cornucopia* significantly reduces queuing time for interactive jobs, and in most scenarios, it ensures that the waiting time for interactive jobs is reduced to zero.

Our analysis indicates that current scheduling algorithms often overlook the queuing time of interactive jobs, leading to excessive waits, especially when resources are scarce. For example, under a 320 GPU configuration, the 2DAS algorithm used with Tiresias resulted in interactive job queues up to 300 seconds. Despite variations across different datasets, our method, Cornucopia, consistently outperforms others, substantially reducing interactive



(a) Philly (b) Venus (c) Saturn

Figure 5: **CCDF of queuing time for interactive jobs.** During this evaluation, we employed 256 GPUs for the Venus set and 320 GPUs for the Philly and Saturn sets.

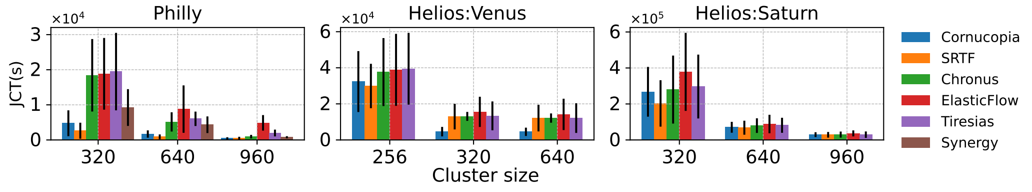
job queuing times. To capture the tail behavior, we plot the complementary cumulative distribution function (CCDF) of queuing time in Figure 5, which shows the probability that the queuing time exceeds a given threshold.

We also find that *Cornucopia* achieves comparable performance to SRTF, which possesses prior knowledge of the durations of all job executions. This can be explained by the generally shorter durations of interactive-type jobs, as in the SRTF scheduling policy, they still manage to attain higher execution priority. It is worth noting that, although Chronus has designed priority differentiation for various job types, its specific priority scheduling strategy differs significantly from ours. Specifically, Chronus prioritizes the scheduling and resource allocation for jobs with impending deadlines and only considers the scheduling of interactive-type jobs first when it is confirmed that one job cannot meet its deadline. This also contributes to its shortcomings in the queuing time metric. Other schedulers do not separately account for interactive-type jobs in the scheduling process, which is also one of the main reasons for their poor performance in queuing time metrics for such jobs.

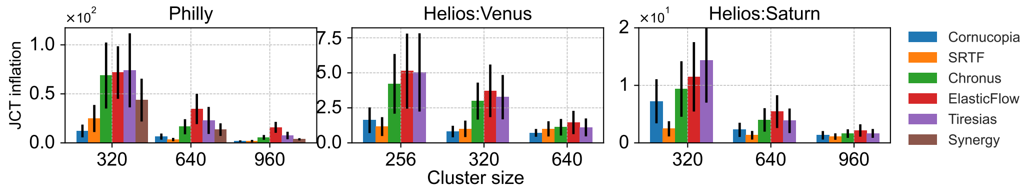
5.3. End-to-End JCT Results

Next, we compare the end-to-end JCT results between *Cornucopia* and all baseline algorithms, as shown in Figure 6. Generally, *Cornucopia* exhibits comparable JCT and better JCT inflation in most test cases, while showing superior performance over the existing design in terms of both JCT and JCT inflation metrics in other cases. In contrast to existing approaches across various cluster sizes, *Cornucopia* demonstrates the capability to reduce the JCT by up to 77% and the JCT inflation by up to 68% under different trace sets, and this improvement is most pronounced on the Philly trace set.

For Venus and Saturn, *Cornucopia* exhibits JCT improvements of about 10% compared to existing methods while demonstrating an improvement of



(a) JCT



(b) JCT Inflation

Figure 6: **The average JCT performance using different schedulers.** The above figure displays the JCT metric, while the below figure depicts the JCT inflation metric. Note that we employ Synergy as the baseline solely within the context of Philly, as it exclusively models the job memory consumption within Philly.

19% to 77% in the JCT inflation metric. We provide the following comparison for the performance improvements observed in *Cornucopia* with respect to JCT and JCT inflation. Compared to Tiresias and Synergy, their lack of elasticity hinders them from effectively utilizing cluster resources during scheduling and preemption processes, leading to higher JCT inflation. While Chronus considers differences in job priorities, its profiling incurs significant time overhead, which partially impacts job JCT performance. Moreover, it does not incorporate elasticity. The results demonstrate that upon adjusting the deadline to infinity, the performance of ElasticFlow significantly deteriorates. This conclusion has also been corroborated in ElasticFlow's paper[10].

To observe the JCT variations of each job more clearly, we also compared the individual job JCT results under different scheduling algorithms, as depicted in Figure 7. We selected SRTF, Synergy, and *Cornucopia* as three representative algorithms and measured the JCT values of each job in the Philly trace set under a cluster with 960 GPUs. It can be observed that, compared to other methods, the average JCT and JCT inflation improvement of *Cornucopia* mainly comes from enhancing shorter-duration jobs, but at the cost of slowing down some longer-duration jobs. This can be explained by the queue degradation mechanism in *Cornucopia*: as jobs continue to run, long-duration jobs eventually fall into the lowest-priority queue, which

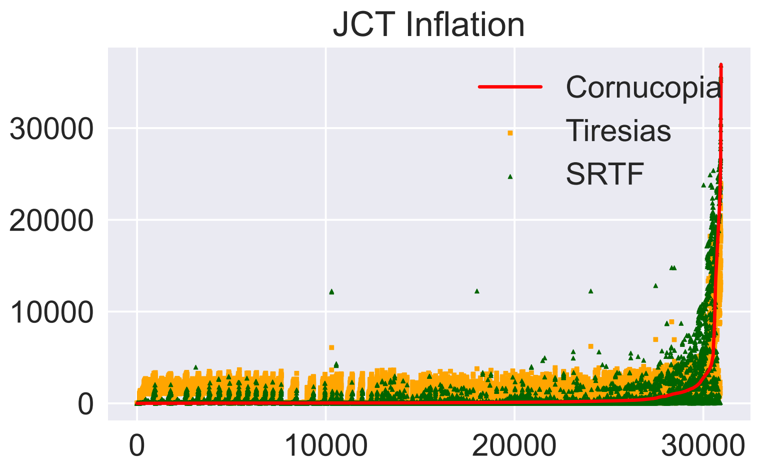


Figure 7: **Individual JCT comparison for different methods.** Experiments are done on a 960-GPU cluster with Philly trace set in our simulator.

prolongs their processing time compared to other methods.

It is worth noting that the similarity in results between *Cornucopia* and SRTF highlights the effectiveness of *Cornucopia*. Specifically, in resource-constrained scenarios, such as running Philly jobs on 320 and 640 GPU clusters or Venus jobs on a 256 GPU cluster, although *Cornucopia* does not achieve lower JCT inflation than SRTF, which possesses prior knowledge of job durations, the difference in their JCT inflation values are small. In resource-abundant settings, one significant advantage of *Cornucopia* over SRTF is its ability to leverage elastic mechanisms to accelerate existing tasks using available surplus resources, resulting in JCT inflation values below 1. As such, when the cluster size increases, the gap in JCT inflation between *Cornucopia* and SRTF gradually narrows, and *Cornucopia* may even surpass the performance of SRTF due to the use of its elastic training mechanism. Under resource-rich conditions, *Cornucopia* leverages elastic training to allocate more GPUs than a job's minimum requirement, accelerating execution via parallelism. In contrast, SRTF assigns a fixed number of GPUs (equal to the job's declared demand), and cannot benefit from surplus resources. This fundamental difference allows Cornucopia to better utilize idle capacity and shorten job execution time when the cluster is not saturated. As shown in Figure 6, this advantage becomes increasingly evident as cluster size grows. Across all three traces, Cornucopia demonstrates a steeper decrease in aver-

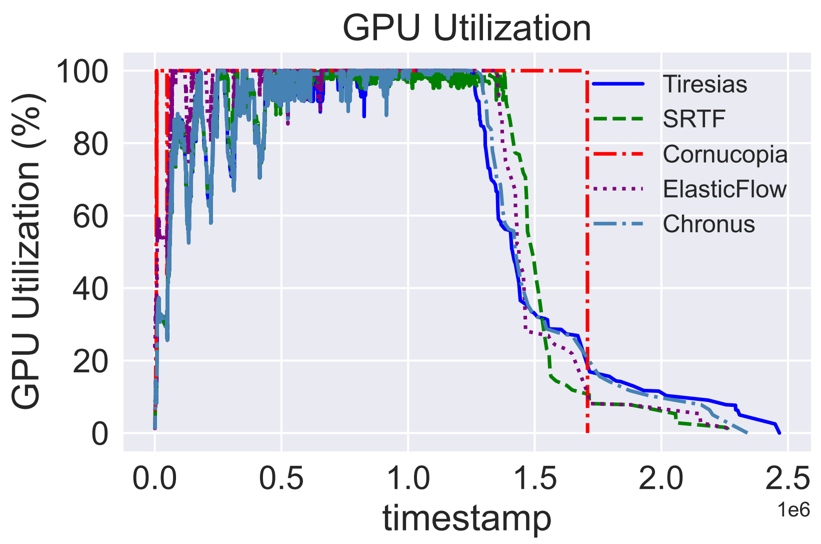


Figure 8: **GPU utilization for different methods.** Experiments are done on a 640-GPU cluster with Venus trace set in our simulator.

age JCT than SRTF when scaling up the number of GPUs. This confirms that Cornucopia is more effective at exploiting resource abundance via elastic scaling.

5.4. Results for cluster resource utilization

It is expected that as long as there are waiting jobs in the queue, there should be no idle resources in the cluster in an ideal scenario. However, if without elasticity, resource gaps often occur in the cluster, which are insufficient to meet the resource demands of any job. Existing non-elastic scheduling strategies such as Tiresias and Chronus are unable to effectively utilize these resources, resulting in low cluster resource utilization, as shown in Figure 8. Due to the elastic nature of ElasticFlow and *Cornucopia*'s design, they can leverage these resource gaps to accelerate existing jobs or partially execute pending jobs, thereby achieving a resource utilization advantage over other solutions. However, as ElasticFlow is a scheduler designed specifically for DDL jobs, its shortcomings in the context of priority job scheduling hinder its overall runtime reduction effectively.

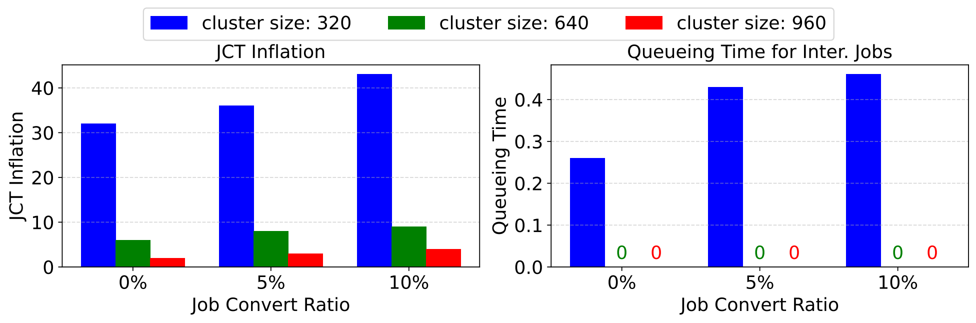


Figure 9: Mislabeled jobs impact on *Cornucopia* performance.

Trace Name	Convert Ratio	Inter. Num	Batch. Num	Inter. JGT	Batch. JGT
Philly	0	6829	24111	187421	22551101
Philly_C5	5	8034	22906	1151599	21586923
Philly_C10	10	9240	21700	2257214	20481308

Table 3: **Synthetic trace sets modified from Philly.** *JGT* means the total Job GPU Time, and *Convert Ratio* refers to the ratio of converting Batch-type jobs into Interactive-type jobs. When it equals 0, it signifies that no conversion is performed, indicating the utilization of the original Philly set.

5.5. Impacts of Mislabled Jobs

The scheduling in *Cornucopia* assigns the highest initial priority to interactive-type jobs by default. If a job is mislabeled, *Cornucopia* might make sub-optimal decisions. This section discusses the impacts of the scenario in which a user incorrectly labels a batch-type job as interactive.

To verify the impact of job types on scheduling, we generated a synthetic trace set based on Philly. In these trace sets, the jobs are identical to those in Philly, except for potential variations only in their types. Specifically, we randomly selected different proportions of Batch jobs from the original Philly set and labeled them as *Interactive*. The trace set we utilized is presented in Table.3.

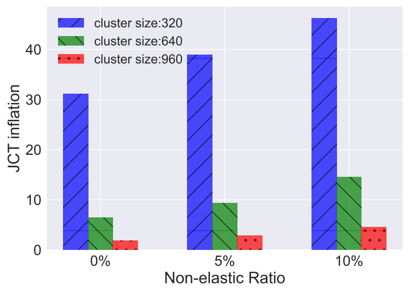
The results are shown in Fig.9. We observe that, although the JCT inflation of the *Cornucopia* increases with a higher job type conversion ratio, *Cornucopia*'s JCT inflation still remains smaller than that of the current state-of-the-art method, Synergy, when the error ratio remains below 10%. Furthermore, the interactive queuing time in *Cornucopia* continues to maintain a low level (the mean value is consistently lower than 1s). In scenarios with cluster sizes of 640 and 960, *Cornucopia* still ensures that interactive jobs have no waiting time.

5.6. Impacts of Non-elastic Parallelism

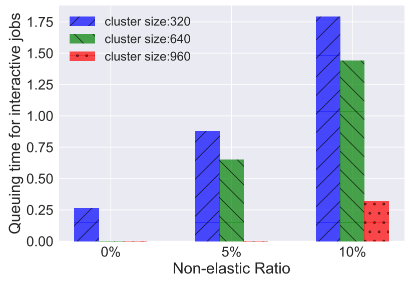
As mentioned in §4, the design of *Cornucopia* heavily leverages the characteristics of elastic training. However, existing elastic training techniques still only support data parallelism, and there is currently no mature elastic technology support for model parallelism and tensor parallelism, mainly due to the challenges associated with reallocating resources when changing the number of resources. As a result, we label such jobs as non-elastic in *Cornucopia* and exclude them from profiling and elastic scaling. Nevertheless, these jobs are still scheduled as regular batch jobs using the MLFQ-based scheduler. This section presents a set of experiments to demonstrate the utility of *Cornucopia*'s scheduling in scenarios where there exist instances of model parallelism and tensor parallelism. In order to make the evaluation more well-directed, we randomly added the *non-elastic* label to jobs from different GPU demand categories in a certain proportion. Specifically, we grouped jobs in the original Philly trace set into three categories based on their GPU demands: jobs requesting 2 to 4 GPUs, 5 to 8 GPUs, and more than 8 GPUs. Then, within these jobs, we selected a certain proportion to be labeled as *non-elastic*, indicating the use of model parallelism or tensor parallelism. We conducted separate tests for the scenarios with 5% and 10% non-elastic jobs, and the corresponding JCT and queuing time results for *Cornucopia* are shown in Figure. 10. It can be observed that as the proportion of non-elastic jobs increases, the JCT and queuing time results of *Cornucopia* also deteriorate continuously. When there are 10% non-elastic jobs present, the JCT metric of *Cornucopia* may be weaker than Synergy, but its interactive queuing time remains significantly superior to other methods.

6. Related Work

GPU cluster schedulers. Apart from the methods discussed in this paper, other GPU scheduling methods have similar processing strategies. Optimus[44] predicts the execution duration of jobs through modeling and employs a heuristic algorithm to minimize JCT. Lyra[11] simultaneously considers the joint scheduling of both training and inference clusters, utilizing the idle time



(a) JCT inflation



(b) Queuing time for interactive jobs.

Figure 10: **Other parallelism support for *Cornucopia*.** The term *non-elastic ratio* refers to the proportion of different GPU-demand jobs that are transformed into *non-elastic* jobs. For example, we selected 5% and 10% of jobs from those with GPU demands of 2-4, 4-8, and larger than 8 for conversion, rather than randomly selecting jobs from the entire set for conversion.

of the inference cluster to accelerate training jobs. Pollux[8] implements dual-end optimization scheduling at both the job and cluster levels through the use of elastic training mechanisms. SJF-BSBF[49] allows multiple jobs to share the same set of GPUs to enhance the resource utilization and processing efficiency of GPU clusters. Heet[37] is a heterogeneity-aware elastic DL scheduler that uses lightweight profiling and price-based scheduling to optimize GPU allocation and reduce job completion time. Crux [39] is a communication scheduler for multi-tenant DL clusters that improves GPU utilization by prioritizing GPU-intensive jobs through contention-aware path selection and priority assignment. In contrast, *Cornucopia* is a partial preemption scheduler designed for mixed interactive-batch workloads. Its core strategy is to use a MLFQ to prioritize interactive jobs. It applies a scaling-aware policy to intelligently preempt a small number of GPUs from batch jobs with low scaling efficiency, thereby ensuring fast response times for interactive tasks.

Systems that support elastic scaling. In recent years, elastic training, as an emerging feature, has been receiving increasing attention and adoption by numerous training systems. Elastic Horovod[15], Pytorch[14] and Mxnet[17] have all integrated elastic training features into their training frameworks. In large-scale production clusters such as PAI[48] and PaddlePaddle[50], elastic training features have also been widely adopted. AntMan[51] provides an extension mechanism that allows fine-grained management of computations and GPU memory during training. These efforts provide the foundational implementation for elastic training, offering *Cornucopia* a concrete deployment environment.

Stop-free elastic training techniques. Some works have made improvements in resource adjustment overhead during the elastic process. EDL[38] significantly reduces elasticity overhead by overlapping worker initialization with regular training processes. Scale-Train[52] introduces the gradient eavesdrop scheme to further enhance the synchronization cost of nodes during elasticity adjustments. Elan[35] optimizes transmission links for node synchronization based on the deployment information of newly added workers during elasticity. These works are orthogonal to *Cornucopia* and can directly benefit it.

7. Discussion and Future Work

Training Consistency. *Cornucopia* preserves training consistency from two perspectives: (i) Statistical convergence consistency. Dynamic resource scaling changes the global batch size, which may influence convergence dynamics. However, this is a well-studied challenge in elastic training. Best practices such as the linear scaling rule can be applied at the framework or user level to maintain convergence quality. *Cornucopia* 's role is to expose elasticity; ensuring statistical equivalence is delegated to the training framework or user configuration. Our goal is to achieve comparable final model accuracy, not to guarantee identical training trajectories. (ii) Deterministic execution consistency. For use cases that require strict reproducibility (e.g., debugging, academic benchmarks), *Cornucopia* allows users to mark jobs as *non-elastic*. These jobs are scheduled with fixed resources and are excluded from preemption, providing a deterministic fallback path.

Memory State Retention. Preemption in *Cornucopia* is not a forceful termination (e.g., kill -9), but a graceful and coordinated process mediated by the cluster manager (e.g., Ray). When a job is partially preempted, a catchable signal (such as SIGTERM) is sent to the affected workers. Elastic training frameworks are designed to handle such signals: Before the process starts, it finalizes the current batch and writes a consistent checkpoint to persistent storage. A well-designed checkpoint typically includes not only the model weights, but also the optimizer state (e.g., momentum and variance for Adam), the learning rate scheduler state, epoch and batch counters, and even random number generator seeds. When the job is resumed under a new resource configuration, training continues seamlessly from the interruption point, with all key memory states preserved.

Fault Recovery and System Reliability. *Cornucopia* adopts a unified mechanism to handle both controlled preemptions and unexpected system faults. The same checkpoint-and-resume workflow is used in both cases, ensuring consistency across failure types. Unlike systems where fault recovery paths are rarely exercised, *Cornucopia* embeds fault handling into routine operation by treating preemption as a common case. This design choice ensures that recovery logic is continuously validated under realistic workloads, thereby strengthening robustness in production. In this way, *Cornucopia* enhances system resilience through routine reinforcement of its fault-handling mechanisms.

Implementation Feasibility and System Integration. *Cornucopia* is designed as a control-plane system that makes high-level resource scheduling decisions, while delegating job execution to elastic training frameworks such as Elastic Horovod or Elastic PyTorch. The control and data planes are decoupled and coordinated through a distributed computing substrate such as Ray. In our implementation, *Cornucopia* manages a one-to-one mapping between logical workers and Horovod processes via Ray actors. This enables precise control over which workers to add or remove at runtime. When *Cornucopia* decides to shrink the GPU allocation of a running job, it sends a command to the Ray head node to terminate specific actors. Ray executes this instruction by gracefully removing the corresponding actors, which effectively eliminates the associated Horovod workers. Elastic Horovod responds by detecting membership changes, triggering a re-rendezvous to reconstruct the communication topology, and re-synchronizing training state from rank-0 to the remaining workers. This workflow ensures that training can resume seamlessly from the last completed iteration. Overall, this design leverages the native elasticity support in Horovod and the actor-based resource abstraction in Ray to achieve practical, low-overhead integration without modifying user training logic.

Resource Reallocation Overhead. Our simulation does not explicitly model the overhead of resource reallocation, such as GPU memory transfers and model reinitialization. We justify this simplification based on the following analysis.

GPU Memory Transfer. *Cornucopia* targets data-parallel training jobs, where model parameters typically fit within the memory of a single mainstream GPU (e.g., A100/H100 with 80GB, H200 with 141GB). Even for small-scale

LLMs (e.g., 7B), the worst-case memory transfer volume is thus bounded by GPU memory capacity. With RDMA-enabled high-speed interconnects (e.g., ConnectX-6/7 at 200–400 Gbps), the transfer time remains within a few seconds. Compared to the JCT in our evaluation, which ranges from tens of thousands to more than a hundred thousand seconds, this overhead is negligible.

Model reinitialization. This overhead includes loading model checkpoints, initializing the training framework, and re-establishing collective communication (e.g., NCCL[53]). Recent system studies BootSeer [54] show that even for large-scale LLM training, initialization accounts for only 3.5% of total training time. For smaller models such as ResNet152 or VGG16 and moderate cluster sizes (under 100 GPUs, which is *Cornucopia* 's primary target), the relative overhead is even lower, typically under tens of seconds. Even for extreme cases involving thousands of GPUs, the startup latency remains below a few hundred seconds, accounting for 1% of JCT. We therefore consider these costs sufficiently minor to be excluded from simulation without altering the conclusions of our study.

Towards Automatic Job Type Inference. One promising direction for future work is the development of an automatic job type inference mechanism to replace or supplement user-provided labels. While *Cornucopia* relies on users to specify whether a job is elastic or *non-elastic*, this introduces the risk of misclassification, which may affect profiling accuracy and scheduling decisions. A potential solution is to dynamically infer a job's elasticity based on early-stage profiling signals—such as scaling behavior, resource utilization trends, or convergence characteristics, and adapt the job classification over time. This would improve system robustness and minimize reliance on external annotations.

Heterogeneous GPU Scheduling and Execution. While *Cornucopia* currently constrains each job to run on a homogeneous set of GPUs, which consistent with mainstream cloud platforms, supporting heterogeneous GPU scheduling and execution remains an important direction for future work. This extension would enable better resource utilization in heterogeneous clusters by allowing jobs to span across different GPU types. Achieving this goal requires revisiting placement, scaling, and scheduling strategies to account for device heterogeneity and interconnect limitations.

8. Conclusion

In this paper, we proposed *Cornucopia*, a priority-aware cluster manager for deep learning clusters with elastic training. In modern GPU clusters, the submission of trial-and-error jobs by users is prevalent, which requires shorter queuing times to facilitate faster responses. However, existing cluster schedulers are primarily designed for traditional batch jobs and do not account for the queuing time requirements of these jobs. Our main objective can be succinctly summarized as how to minimize the queuing time for interactive jobs without adversely affecting the overall JCT performance while considering a mixed workload. With three key components-*(i)* the MLFQ-based scheduler which processes jobs with different priorities, *(ii)* the ramp-up profiler which provides job scalability information and *(iii)* the resource allocator which utilizes elasticity to implement partially preemptive/running for jobs, *Cornucopia* achieved a reduction in interactive queuing time while maintaining the overall job workload's JCT levels. Through extensive large-scale simulation testing, we found that *Cornucopia* is capable of reducing interactive queuing time by up to 90% while maintaining JCT levels compared to existing state-of-the-art methods.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grant No. 62072302, the US National Science Foundation under Grants 1755659, 1815619, 2105564, and VMWare.

References

[1] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, B. Guo, Swin transformer: Hierarchical vision transformer using shifted windows, in: Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 10012–10022.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.

[3] J. Ye, X.-C. Wen, Y. Wei, Y. Xu, K. Liu, H. Shan, Temporal modeling matters: A novel temporal emotional modeling approach for speech emotion recognition, in: ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2023, pp. 1–5.

[4] J. Duan, S. Zhang, Z. Wang, L. Jiang, W. Qu, Q. Hu, G. Wang, Q. Weng, H. Yan, X. Zhang, et al., Efficient training of large language models on distributed infrastructures: a survey, arXiv preprint arXiv:2407.20018 (2024).

[5] H. Liao, B. Liu, X. Chen, Z. Guo, C. Cheng, J. Wang, X. Chen, P. Dong, R. Meng, W. Liu, et al., Ub-mesh: a hierarchically localized nd-fullmesh datacenter network architecture, arXiv preprint arXiv:2503.20377 (2025).

[6] W. Gao, Z. Ye, P. Sun, Y. Wen, T. Zhang, Chronus: A novel deadline-aware scheduler for deep learning training jobs, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 609–623.

[7] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: A gpu cluster manager for distributed deep learning, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019, pp. 485–500.

[8] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, E. P. Xing, Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning, in: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), 2021.

[9] J. Mohan, A. Phanishayee, J. Kulkarni, V. Chidambaram, Looking beyond gpus for dnn scheduling on multi-tenant clusters, in: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 579–596.

[10] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, X. Liu, 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, 2023, pp. 266–280.

[11] J. Li, H. Xu, Y. Zhu, Z. Liu, C. Guo, C. Wang, Lyra: Elastic scheduling for deep learning clusters, in: Proceedings of the Eighteenth European Conference on Computer Systems, 2023, pp. 835–850.

[12] Q. Hu, P. Sun, S. Yan, Y. Wen, T. Zhang, Characterization and prediction of deep learning workloads in large-scale gpu datacenters, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

[13] D. Shahmirzadi, N. Khaledian, A. M. Rahmani, Analyzing the impact of various parameters on job scheduling in the google cluster dataset, Cluster Computing 27 (6) (2024) 7673–7687.

[14] Elastic pytorch, `https://pytorch.org/elastic/0.2.0rc1/distributed.html#module-torchelastic.distributed.launch` (2021).

[15] Horovod, `https://horovod.readthedocs.io/en/latest/elastic_include.html` (2019).

[16] Paddlepaddle, `https://www.paddlepaddle.org.cn/en` (2020).

[17] Mxnet, `https://mxnet.apache.org/versions/1.9.1/` (2022).

[18] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, F. Yang, Analysis of large-scale multi-tenant gpu clusters for dnn training workloads, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), 2019, pp. 947–960.

[19] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al., Pytorch distributed: Experiences on accelerating data parallel training, arXiv preprint arXiv:2006.15704 (2020).

[20] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-lm: Training multi-billion parameter language models using model parallelism, arXiv preprint arXiv:1909.08053 (2019).

[21] S. Rajbhandari, J. Rasley, O. Ruwase, Y. He, Zero: Memory optimizations toward training trillion parameter models, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–16.

[22] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, G. E. Dahl, Measuring the effects of data parallelism on neural network training, arXiv preprint arXiv:1811.03600 (2018).

[23] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al., Gpipe: Efficient training of giant neural networks using pipeline parallelism, Advances in neural information processing systems 32 (2019).

[24] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, M. Zaharia, Pipedream: Generalized pipeline parallelism for dnn training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.

[25] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, et al., Efficient large-scale language model training on gpu clusters using megatron-lm, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

[26] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al., Pytorch fsdp: experiences on scaling fully sharded data parallel, arXiv preprint arXiv:2304.11277 (2023).

[27] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, B. Catanzaro, Reducing activation recomputation in large transformer models (2022). `arXiv:2205.05198`.
URL `https://arxiv.org/abs/2205.05198`

[28] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, Z. Chen, Gshard: Scaling giant models with conditional computation and automatic sharding (2020). `arXiv:2006.16668`.
URL `https://arxiv.org/abs/2006.16668`

[29] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, et al., Dapple: A pipelined data parallel approach for training large models, in: Proceedings of the 26th ACM SIGPLAN

Symposium on Principles and Practice of Parallel Programming, 2021, pp. 431–445.

[30] W. Zhang, S. Gupta, X. Lian, J. Liu, Staleness-aware async-sgd for distributed deep learning, arXiv preprint arXiv:1511.05950 (2015).

[31] S. Li, O. Mangoubi, L. Xu, T. Guo, Sync-switch: Hybrid parameter synchronization for distributed deep learning, in: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), IEEE, 2021, pp. 528–538.

[32] S. Li, R. J. Walls, T. Guo, Characterizing and modeling distributed training with transient cloud gpu servers, in: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2020, pp. 943–953.

[33] P. Patarasuk, X. Yuan, Bandwidth efficient all-reduce operation on tree topologies, in: 2007 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2007, pp. 1–8.

[34] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, Journal of Parallel and Distributed Computing 69 (2) (2009) 117–124.

[35] L. Xie, J. Zhai, B. Wu, Y. Wang, X. Zhang, P. Sun, S. Yan, Elan: Towards generic and efficient elastic training for deep learning, in: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2020, pp. 78–88.

[36] Z. Bian, S. Li, W. Wang, Y. You, Online evolutionary batch size orchestration for scheduling deep learning workloads in gpu clusters, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

[37] Z. Mo, H. Xu, C. Xu, Heet: Accelerating elastic training in heterogeneous deep learning clusters, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 499–513. doi:10.1145/3620665.3640375.
URL https://doi.org/10.1145/3620665.3640375

[38] Y. Wu, K. Ma, X. Yan, Z. Liu, Z. Cai, Y. Huang, J. Cheng, H. Yuan, F. Yu, Elastic deep learning in multi-tenant gpu clusters, IEEE Transactions on Parallel and Distributed Systems 33 (1) (2021) 144–158.

[39] J. Cao, Y. Guan, K. Qian, J. Gao, W. Xiao, J. Dong, B. Fu, D. Cai, E. Zhai, Crux: Gpu-efficient communication scheduling for deep learning training, in: Proceedings of the ACM SIGCOMM 2024 Conference, 2024, pp. 1–15.

[40] Amazon Web Services, Amazon SageMaker, `https://aws.amazon.com/sagemaker/`, accessed: 2025-07-26.

[41] Alibaba Cloud, Machine Learning Platform for AI (PAI), `https://www.aliyun.com/product/pai`, accessed: 2025-07-26.

[42] J. Ru, J. Keung, An empirical investigation on the simulation of priority and shortest-job-first scheduling for cloud-based software systems, in: 2013 22nd Australian Software Engineering Conference, IEEE, 2013, pp. 78–87.

[43] C. Gao, V. C. Lee, K. Li, D-srtf: Distributed shortest remaining time first scheduling for data center networks, IEEE Transactions on Cloud Computing 9 (2) (2018) 562–575.

[44] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Guo, Optimus: an efficient dynamic resource scheduler for deep learning clusters, in: Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 1–14.

[45] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, G. R. Ganger, 3sigma: distribution-based cluster scheduling for runtime uncertainty, in: Proceedings of the Thirteenth EuroSys Conference, 2018, pp. 1–17.

[46] P. Jogalekar, M. Woodside, Evaluating the scalability of distributed systems, IEEE Transactions on parallel and distributed systems 11 (6) (2000) 589–603.

[47] M. Thombare, R. Sukhwani, P. Shah, S. Chaudhari, P. Raundale, Efficient implementation of multilevel feedback queue scheduling, in: 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), IEEE, 2016, pp. 1950–1954.

[48] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, Y. Ding, Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters, in: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 945–960.

[49] Y. Luo, Q. Wang, S. Shi, J. Lai, S. Qi, J. Zhang, X. Wang, Scheduling deep learning jobs in multi-tenant gpu clusters via wise resource sharing, in: 2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS), IEEE, 2024, pp. 1–10.

[50] Y. Ao, Z. Wu, D. Yu, W. Gong, Z. Kui, M. Zhang, Z. Ye, L. Shen, Y. Ma, T. Wu, et al., End-to-end adaptive distributed training on paddlepaddle, arXiv preprint arXiv:2112.02752 (2021).

[51] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, Y. Jia, Antman: Dynamic scaling on gpu clusters for deep learning, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 533–548.

[52] K. Kim, H. Lee, S. Oh, E. Seo, Scale-train: A scalable dnn training framework for a heterogeneous gpu cloud, IEEE Access 10 (2022) 68468–68481.

[53] NVIDIA, NCCL: Optimized primitives for collective multi-gpu communication, `https://github.com/NVIDIA/nccl` (2025).

[54] R. Li, X. Zhi, J. Chi, M. Yu, L. Huang, J. Zhu, W. Zhang, X. Ma, W. Liu, Z. Zhu, D. Luo, Z. Song, X. Yin, C. Xiang, S. Wang, W. Xiao, G. Cooperman, Bootseer: Analyzing and mitigating initialization bottlenecks in large-scale llm training (2025). `arXiv:2507.12619`.
URL `https://arxiv.org/abs/2507.12619`