Tiresias: A GPU Cluster Manager for Distributed Deep Learning

Juncheng Gu, Mosharaf Chowdhury, and Kang G. Shin, *University of Michigan, Ann Arbor;* Yibo Zhu, *Microsoft and Bytedance;* Myeongjae Jeon, *Microsoft and UNIST;* Junjie Qian, *Microsoft;* Hongqiang Liu, *Alibaba;* Chuanxiong Guo, *Bytedance*

https://www.usenix.org/conference/nsdi19/presentation/gu

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Tiresias: A GPU Cluster Manager for Distributed Deep Learning

Juncheng Gu[1], Mosharaf Chowdhury[1], Kang G. Shin[1], Yibo Zhu[2,3]
Myeongjae Jeon[2,4], Junjie Qian[2], Hongqiang Liu[5], Chuanxiong Guo[3]

[1]University of Michigan, [2]Microsoft, [3]Bytedance, [4]UNIST, [5]Alibaba

Abstract

Deep learning (DL) training jobs bring some unique challenges to existing cluster managers, such as unpredictable training times, an all-or-nothing execution model, and inflexibility in GPU sharing. Our analysis of a large GPU cluster in production shows that existing big data schedulers cause long queueing delays and low overall performance.

We present Tiresias, a GPU cluster manager tailored for distributed DL training jobs, which efficiently schedules and places DL jobs to reduce their job completion times (JCTs). Given that a DL job's execution time is often unpredictable, we propose two scheduling algorithms – *Discretized Two-Dimensional Gittins index* relies on partial information and *Discretized Two-Dimensional LAS* is information-agnostic – that aim to minimize the average JCT. Additionally, we describe when the consolidated placement constraint can be relaxed, and present a placement algorithm to leverage these observations without any user input. Experiments on the Michigan ConFlux cluster with 60 P100 GPUs and large-scale trace-driven simulations show that Tiresias improves the average JCT by up to 5.5× over an Apache YARN-based resource manager used in production. More importantly, Tiresias's performance is comparable to that of solutions assuming perfect knowledge.

1 Introduction

Deep learning (DL) is gaining rapid popularity in various domains, such as computer vision, speech recognition, etc. DL training is typically compute-intensive and requires powerful and expensive GPUs. To deal with ever-growing training datasets, it is common to perform distributed DL (DDL) training to leverage multiple GPUs in parallel. Many platform providers have built GPU clusters to be shared among many users to satisfy the rising number of DDL jobs [1, 3, 4, 9]. Indeed, our analysis of Microsoft traces shows a 10.5× year-by-year increase in the number of DL jobs since 2016. Efficient job scheduling and smart GPU allocation (i.e., job placement) are the keys to minimizing the cluster-wide average JCT and maximizing resource (GPU) utilization.

Due to the unique constraints of DDL training, we observe two primary limitations in current cluster manager designs.

1. Naïve scheduling due to unpredictable training time. Although shortest-job-first (SJF) and shortest-remaining-time-first (SRTF) algorithms are known to minimize the average JCT [23, 24], they require a job's (remaining) execution time, which is often unknown for DL training jobs. Optimus [34] can predict a DL training job's remaining execution time by relying on its repetitive execution pattern and assuming that its loss curve will converge. However, such proposals make over-simplified assumptions about jobs having smooth loss curves and running to completion; neither is always true in production systems (§2.2).

Because of this, state-of-the-art resource managers in production are rather naïve. For example, the internal solution of Microsoft is extended from Apache YARN's Capacity Scheduler that was originally built for big data jobs. It only performs basic orchestration, i.e., non-preemptive scheduling of jobs as they arrive. Consequently, users often experience long queuing delays when the cluster is over-subscribed – up to several hours even for small jobs (Appendix A).

2. Over-aggressive consolidation during placement. Existing cluster managers also attempt to consolidate a DDL job onto the minimum number of servers that have enough GPUs. For example, a job with 16 GPUs requires at least four servers in a 4-GPUs-per-server cluster, and the job may be blocked if it cannot find four completely free servers. The underlying assumption is that the network should be avoided as much as possible because it can become a bottleneck and waste GPU cycles [31]. However, we find that this assumption is only partially valid.

In this paper, we propose Tiresias, a shared GPU cluster manager that aims to address the aforementioned challenges regarding DDL job scheduling and placement (§3). To ensure that Tiresias is practical and readily deployable, we rely on the analysis of production job traces, detailed measurements of training various DL models, and two simple yet effective ideas. In addition, we intentionally keep Tiresias transparent to users, i.e., all existing jobs can run without any additional user-specified configurations.

Our first idea is a new scheduling framework (2DAS) that aims to minimize the JCT when a DL job's execution time is unpredictable. We propose two scheduling algorithms under this framework: *Discretized 2D-LAS* and *Discretized 2D-Gittins index*. The Gittins index policy [8, 21] is known to be the optimal in the single-server scenario in minimizing the average JCT when JCT distributions are known. Similarly, the classic LAS (Least-Attained Service) algorithm [33] has been widely applied in many information-agnostic scenarios, such as network scheduling in datacenters [13, 17]. Both assign each job a priority – the former uses the Gittins index while the latter directly applies the service that job has received so far – that changes over time, and jobs are scheduled in order of their current priorities.

Adapting these approaches to the DDL scheduling problem faces two challenges. First, one must consider both the spatial (how many GPUs) and temporal (for how long) dimensions of a job when calculating its priorities. We show that simply considering one is not enough. Specifically, a job's total attained service in our algorithms jointly considers both its spatial and temporal dimensions.

More importantly, because relative priorities continuously change as some jobs receive service, jobs are continuously preempted. Although this may be tolerable in networking scenarios where starting and stopping a flow is simpler, preempting a DDL job from its GPUs can be expensive because data and model must be copied back and forth between the main memory and GPU memory. To avoid aggressive job preemptions, we apply *priority discretization* atop the two classic algorithms – a job's priority changes after fixed intervals.

Overall, when the cluster manager has the distribution of previous job execution times that may still be valid in the near future, our scheduling framework chooses the *Discretized 2D-Gittins index*. If no prior knowledge is available, *Discretized 2D-LAS* will be applied.

Our second idea is to use model structure to loosen the consolidated placement constraint whenever possible. We observe that only certain types of DL models are sensitive to whether they are consolidated or not, and their sensitivity is due to skew in tensor size distributions in their models. We use this insight to separate jobs into two categories: jobs that are sensitive to consolidation (high skew) and the rest. We implement an RDMA network profiling library in Tiresias that can determine the model structure of DDL jobs through network-level activities. By leveraging the profiling library and the iterative nature of DDL training, Tiresias can transparently and intelligently place jobs. Tiresias first runs the job in a trial environment for a few iterations, and then determines the best placement strategy according to the criteria summarized from previous measurements.

We have implemented Tiresias[1] and evaluated using unmodified TensorFlow DDL jobs on a 15-server GPU cluster (each server with four P100 GPUs with NVlink) using traces derived from a Microsoft production cluster. We further evaluate Tiresias using large-scale trace-driven simulations. Our results show that Tiresias improves the average JCT by up to 5.5× w.r.t. current production solutions and 2× w.r.t. Gandiva [41], a state-of-the-art DDL cluster scheduler. Moreover, it performs comparably to solutions using perfect knowledge of all job characteristics.

In summary, we make the following contributions:

- Tiresias is the first information-agnostic resource manager for GPU clusters. Also, it is the first that applies two-dimensional extension and priority discretization into DDL job scheduling. It can efficiently schedule and place unmodified DDL jobs without any additional information from the users. When available, Tiresias can leverage partial knowledge about jobs as well.
- Tiresias leverages a simple, externally-observable, model-specific criteria to determine when to relax worker GPU collocation constraints.
- Our design is practical and readily deployable, with significate performance improvements.

[1]https://github.com/SymbioticLab/Tiresias

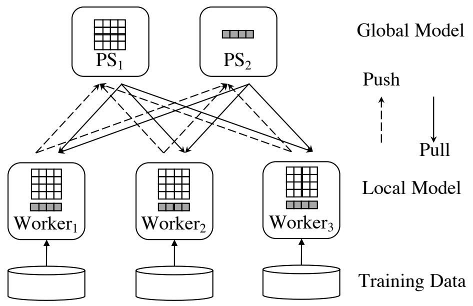


Figure 1: Data parallelism & parameter server architecture. This DDL job has two parameter servers (PS) and three workers.

2 Background and Motivation

2.1 Distributed Deep Learning (DDL)

As DL models become more sophisticated and are trained on larger datasets, distributed training is becoming more prevalent (see Appendix A). Here, we focus on *data parallelism*, which is the most common option for DDL training in popular DDL frameworks.[2] As Figure 1 shows, each worker occupies a GPU and works on its local copy of the DL model. The training dataset is divided into equal-sized parts to feed the workers. All jobs are trained in the *synchronous* mode which has been observed often to achieve faster convergence than asynchronous distributed training over GPUs [19].

Periodic iterations. DL training works in an iterative fashion. In each *iteration*, workers first perform *forward-backward* computation with one chunk of its training data (*minibatch*). Workers then aggregate local results to update the DL model with each other, which is referred to as *model aggregation*. Since the computation load and the communication volume are exactly the same across iterations, the iteration time of a DDL job is highly predictable.

Parameter server architecture. The parameter server (PS) architecture [30] (Figure 1) is the most popular method for model aggregation. The parameter server hosts the master copy of the DL model. It is in charge of updating the model using the local results from all workers. The workers pull back the updated model from the parameter server at the beginning of each iteration. There can be multiple parameter servers in a single DDL job.

***Trial-and-error* exploration.** Training a DL model is not an one-time effort and often works in a *trial-and-error*

[2]There are other parallelization architectures, like model parallelism, which is job-specific and much less popular.

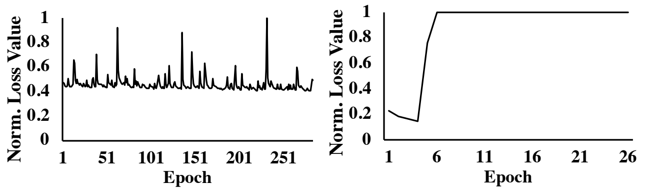


Figure 2: The training loss of two production jobs from Microsoft.

manner. DL model exposes many *hyperparameters* that express the high-level properties of the model. To get a high-quality model, the combinations of hyperparameters need to be explored in a very large search space; this is known as *hyperparameter-tuning* [41, 44]. Users can use AutoML [2] to perform this exploration efficiently and automatically by using some searching tools [14]. In AutoML, many DL jobs with different hyperparameter configurations are generated to train the same job. Most of those jobs will be killed because of random errors, or low quality of improvement. With the feedbacks from early trials, AutoML can search new configurations and spawn new jobs. Only a very small portion of those jobs with good qualities can run to completion.

2.2 Challenges

We highlight three primary challenges faced by DDL cluster managers in production. These challenges originate from the nature of DDL training and are not specific to the Microsoft cluster. See Appendix A for more details about the Microsoft cluster and its workload.

Unpredictable job duration. Current solutions that predict DL job training times [34] all assume DL jobs to (1) have smooth loss curves and (2) reach their training targets and complete. However, for many poor models during a *trial-and-error* exploration, their loss curves are not as smooth as the curves of the best model ultimately picked at the end of exploration. We show two representative examples from Microsoft in Figure 2. The spikes in the first example and the non-decreasing curve in the second example make it challenging to predict when the target will be hit. In addition to these proprietary models, popular public models sometimes also show non-smooth curves [25]. Additionally, the termination conditions of DL jobs are non-deterministic. In AutoML, most of the trials are killed because of quality issues which are determined by the searching mechanism. Usually, users also specify a maximum epoch number to train for cases when the job cannot achieve the training target. Therefore, a practical resource manager design should not rely on the accuracy/loss curve for predicting eventual job completion time.

Over-aggressive job consolidation. Trying to minimize network communication during model aggregation is a common optimization in distributed training because the network can be a performance bottleneck and waste GPU cycles [31]. Hence, many existing GPU cluster managers blindly follow a consolidation constraint when placing DDL jobs – specifically, they assign all components (parameter servers and workers) of the job to the same or the minimum number of servers. A DDL job will often wait when it cannot be consolidated, even if there are enough spare resources elsewhere in the cluster. Although this constraint was originally set for good performance, it often leads to longer queuing delays and resource under-utilization in practice.

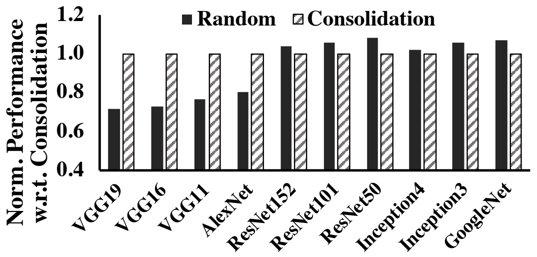


Figure 3: 4 concurrent 8-worker jobs with different placement schemes. The performance values are normalized by the value of the consolidation scheme. We use the median value from 10 (20) runs for consolidation (random) scheme.

To understand the importance of this constraint, we run four concurrent 8-GPU jobs using different placement (random and always-consolidate) strategies on eight 4-GPU servers. Similar to [45], each job uses eight parameter servers – the same as the number of workers. Figure 3 shows that the locality of workers mainly impacts the VGG family and AlexNet. Nevertheless, neither the cluster operator nor the users can tell which category a job belongs to.

Time overhead of preemption. The current production cluster does not preempt jobs because of large time overhead. To show this, we manually test pausing and resuming a DDL job on our local testbed. Upon pausing, the chief worker checkpoints the most recent model on a shared storage. The checkpointed model file will be loaded by all workers when the job is resumed. Figures 4 and 5 show the detailed numbers. Whenever Tiresias preempts a job, we must take this overhead into account.

2.3 Potential for Benefits

We can achieve large gains by mitigating two common myths.

Myth I: jobs cannot be scheduled well without exact job duration. Despite the fact that DDL job durations are often unpredictable, their overall distribution can be learned from history logs. The Gittins index policy [21], which is widely used for solving the classic multi-armed bandit problem [21], can decrease the average JCT as long as the job duration distribution is given. Even without that information, the LAS algorithm can efficiently schedule jobs based on their attained service.

Myth II: DDL jobs should always be consolidated. While it is true that consolidated placement of a job may minimize its commuication time, we find that some DDL jobs are insensitive to placement. We identify that the core factor is the model structure (§3.3).

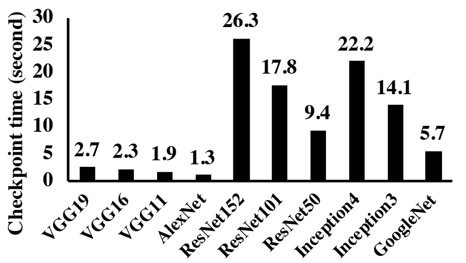


Figure 4: Time overhead of pausing a DDL job in Tensorflow. Only the chief worker checkpoints the most updated model.

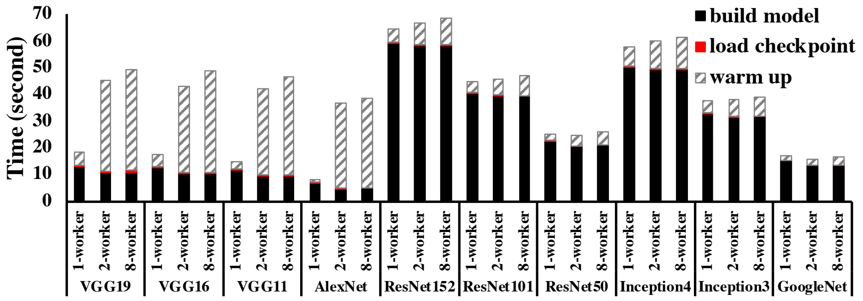


Figure 5: Time overhead of resuming a DDL job in Tensorflow. Each model is tested with different number of workers.

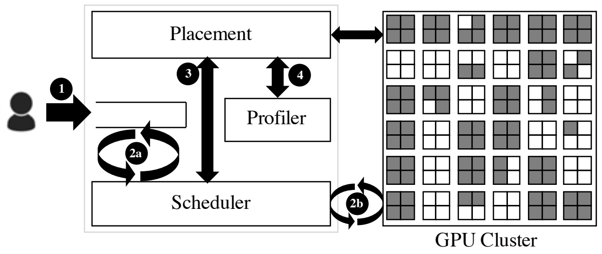


Figure 6: Tiresias components and their interactions. Job lifecycle under Tiresias is described in Section 3.1. In this figure, each machine has four GPUs; shaded ones represent GPUs in use.

In the rest of this paper, we demonstrate that Tiresias – using smarter job placement and scheduling strategies – can improve the average total job completion time by more than 5× when running the same set of jobs.

3 Tiresias Design

This section describes Tiresias's architecture, followed by descriptions of its two key components – scheduler and placement manager – and the profiler that learns the job characteristics during runtime.

3.1 Overall Architecture

Tiresias is a bespoke resource manager for GPU clusters, where the primary workload is DL training. It deals with both allocating GPUs to individual jobs (i.e., job placement) and scheduling multiple jobs over time. So, it has two primary objectives: one user-centric and the other operator-centric.

1. *Minimizing the average JCT:* Jobs should complete as fast as possible regardless of their requirements.
2. *High GPU utilization:* All the GPUs in the cluster should be utilized as much as possible.

Tiresias has an additional goal to balance between operator- and user-centric objectives.

3. *Starvation freedom:* Jobs should not starve for arbitrarily long periods.

Constraints and assumptions: Tiresias must achieve the aforementioned objectives under realistic assumptions highlighted in prior sections:

1. *Online job arrival:* Jobs are submitted by users (trial-and-error exploration mechanisms such as AutoML) in an online fashion. The resource requirements of a job J (i.e., the number of parameter servers PS_J and workers W_J) are given but unknown prior to its arrival. Model and data partitions are determined by the DL framework and/or the user [9, 16, 42]. Tiresias only deals with resource allocation and scheduling.
2. *Unknown job durations:* Because of non-smooth loss curves and non-deterministic termination in practice, a DL job's duration cannot be predicted. However, the overall distribution of job duration may sometimes be available via history logs.
3. *Unknown job-specific characteristics:* A user does not know and cannot control how the underlying DL framework(s) will assign tensors to parameter servers and the extent of the corresponding skew.
4. *All-or-nothing resource allocation:* Unlike traditional big data jobs where tasks can be scheduled over time [11], DL training jobs require all parameter servers and workers to be simultaneously active; i.e., all required resources must be allocated together.

Job lifecycle: Tiresias is designed to optimize the aforementioned objectives without making any assumptions about a job's resource requirements, duration, or its internal characteristics under a specific DL framework.

Figure 6 presents Tiresias's architecture along with the sequence of actions that take place during a job's lifecycle. As soon as a job is submitted, its GPU requirements become known, and it is appended to a WAITQUEUE (1). The scheduler (§3.2) periodically schedules jobs from the WAITQUEUE and preempts running jobs from the cluster to the WAITQUEUE (2a and 2b) on events such as job arrival, job completion, and changes in resource availability. When starting a job for the first time or resuming a previously preempted job, the scheduler relies on the placement module (§3.3) to allocate its GPUs (3). If a job is starting for the first time, the placement module first profiles it – the profiler identifies job-specific characteristics such as skew in tensor distribution – to determine whether to consolidate the job or not (4).

3.2 Scheduling

The core of Tiresias lies in its scheduling algorithm that must (1) *minimize the average JCT* and (2) *increase cluster utilization* while (3) *avoiding starvation*.

We observe that preemptive scheduling is necessary to satisfy these objectives. One must employ preemption to avoid head-of-line (HOL) blocking of smaller/shorter jobs by the larger/longer ones – HOL blocking is a known problem of FIFO scheduling currently used in production [41]. Examples of preemptive scheduling algorithms include time-sharing,[3] SJF, and SRTF. For example, DL jobs in Gandiva [41] are scheduled by time-sharing. However, time-sharing based algorithms are designed for isolation via fair sharing, not minimizing the average JCT. SJF and SRTF are also inapplicable because of an even bigger uncertainty: it is difficult, if not impossible, to predict how long a DL training job will run. At the same time, size-based heuristics (i.e., how many GPUs a job needs) are not sufficient either, because they ignore job durations.

3.2.1 Why Two-Dimensional Scheduling?

By reviewing the time- or sized-based heuristics, we believe that considering only one aspect (spatial or temporal) is not enough when scheduling DDL jobs on a cluster with limited GPU resources. In an SRTF scheduler, large jobs with short remaining time can occupy many GPUs, causing non-negligible queuing delays for many small but newly submitted jobs. If the scheduler is smallest-first (w.r.t. the number of GPUs), then large jobs may be blocked by a stream of small jobs even if they are close to completion.

To quantify the approaches, we ran trace-driven simulations on three different schedulers using the Microsoft production trace: (1) smallest-first (SF); (2) SRTF; and (3) shortest-remaining-service-first (SRSF). Of them, the first two are single-dimensional schedulers; the last one considers both spatial and temporal aspects. The remaining service in SRSF is the multiplication of a job's remaining time and the number of GPUs. For this simulation, we assume that job durations are given when needed.

Table 1 shows that SRSF outperforms the rest in minimizing the average JCT. SRSF has a much smaller tail JCT than the single-dimensional counterparts as well. Altogether, we move forward in building a DDL scheduler that considers both spatial and temporal aspects of resource usage.

Note that, among the three, SF is not a time-based algorithm; hence, it does not actively attempt to minimize the average JCT. As for the rest, SRTF is not always worse than SRSF, either. For example, large-and-short jobs that have many GPUs but short service time can mislead the SRSF scheduler and block many smaller jobs. However, in DL training, multiple GPUs are typically allocated to the jobs that have well-tuned hyperparameters and run to completion.

[3]Also known as processor-sharing.

Table 1: Normalized performance of single-dimensional schedulers w.r.t. SRSF.

	Avg. JCT	Med. JCT	95th JCT
Smallest-First (SF)	1.52	1.20	3.45
SRTF	1.03	1.01	1.55

Therefore, the fraction of large-and-short jobs is often small in practice.

3.2.2 Two-Dimensional Attained Service-Based Scheduler (2DAS)

We address the aforementioned challenges with the 2DAS scheduler, which schedules DL jobs without relying on their exact durations while taking their GPU requirements into consideration. 2DAS generalizes the classic least-attained service (LAS) scheduling discipline [33] as well as the Gittins index policy [21] to DL job scheduling by considering both the *spatial* and *temporal* aspects of such jobs as well as their all-or-nothing characteristic. At a high-level, 2DAS assigns each job a priority based on its attained service. The attained service of a job is calculated based on the number of GPUs it uses (W_J) and the amount of time it has been running so far (t_J). The former becomes known upon the job arrival, while the latter continuously increases.

The priority function in 2DAS can be changed based on different prior knowledge. When *no job duration information* is provided, the priority function applies the LAS algorithm where a job's priority is inverse to its attained service. If the cluster operator provides *the distribution of job duration* from previous experience, then a job's priority equals its Gittins index value (Pseudocode 1). In the Gittins index-based algorithm, the ratio (Line 11) is between (1) the probability that the job will complete within the service quantum of Δ (i.e., the possibility of reward when adding up to Δ overhead on all subsequent jobs) and (2) the expected service that Job J will require for completion.

Both LAS and Gittins index take job's attained service as their inputs. LAS prefers jobs that received less service. All jobs start with the highest priority, and their priorities decrease as they receive more service. The Gittins index value of job represents how *likely* the job that has received some amount of service can complete within the next service quantum. Higher Gittins index value means higher priority.

Example: Let us consider an example that illustrates both the algorithms and compares them against SRSF that has complete information (Figure 7). Three DL jobs arrive at a two-GPU machine at the same time. The resource requirement of each job is represented using (number of GPUs, duration) pairs. Only SRSF has prior knowledge of job duration. 2D-Gittins index knows the distribution, while 2D-LAS has no related information. The average JCTs in this example are 9.3, 10 and 11.7 units of time for SRSF, 2D-Gittins index, and 2D-LAS, respectively. In general, algorithms with more information perform better in minimizing the average JCT.

Pseudocode 1 Priority function in 2DAS

```
1:  procedure PRIORITY(Job $J$, Distribution $\mathbb{D}$)
2:    if $\mathbb{D}$ is $\varnothing$ then            ▷ w/o distribution, apply LAS
3:      $R_J = -W_J \times t_J$
4:    else                        ▷ w/ distribution, apply Gittins index
5:      $R_J = Gittins\_Index(J, \mathbb{D})$
6:    return $R_J$
7:  end procedure
8:
9:  procedure GITTINS_INIDEX(Job $J$, Distribution $\mathbb{D}$)
10:   $a_J = W_J \times t_J$
11:   $G_J = \sup_{\Delta>0} \frac{\mathbf{P}(S-a_J \le \Delta | S>a_J)}{\mathbf{E}[min\{S-a_J,\Delta\}|S>a_J]}$
12:       ▷ $\mathbf{P}$ is the probability and $\mathbf{E}$ is the mean, both of which are
      calculated from $\mathbb{D}$. $\Delta$ is the service quantum.
13:   return $G_J$
14: end procedure
```

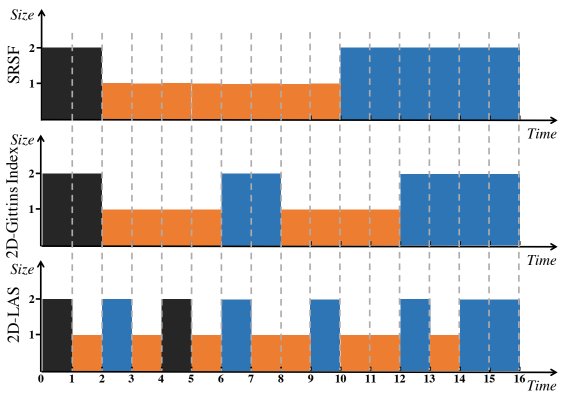


Figure 7: Time sequence of three jobs with three different two-dimensional scheduling algorithms. Job 1 (black) is $(2,2)$, job 2 (orange) is $(1,8)$, and job 3 (blue) is $(2,6)$. The first value in each tuple is the number of GPUs while the second is duration. The scheduling interval is one unit of time. The 2D-Gittins index values for this example are shown in Appendix C. Job index is used to break ties.

3.2.3 Priority Discretization

As observed in prior work [17], using continuous priorities can lead to a sequence of preemptions and subsequent resumptions for all jobs. Unlike preempting a network flow or a CPU process, preempting and resuming a DL job on GPU(s) can be time-consuming and expensive (§2.2). The excessive cost can make 2DAS infeasible. Furthermore, continuous preemption degenerates 2DAS to fair sharing by time-division multiplexing, which increases the average JCT.

We address these challenges by adopting the *priority discretization* framework based on the classic Multi-Level Feedback Queue (MLFQ) algorithm [12, 17, 18].

Discretized 2DAS: Instead of using a continuous priority spectrum, we maintain K logical queues $(Q_1, Q_2, \ldots, Q_K)$, with queue priorities decreasing from Q_1 to Q_K (Figure 8). The i-th queue contains jobs of attained service $(W_J t_J)$ values within $[Q_i^{\text{lo}}, Q_i^{\text{hi}})$. Note that $Q_1^{\text{lo}} = 0$, $Q_K^{\text{hi}} = \infty$, and $Q_{i+1}^{\text{lo}} = Q_i^{\text{hi}}$.

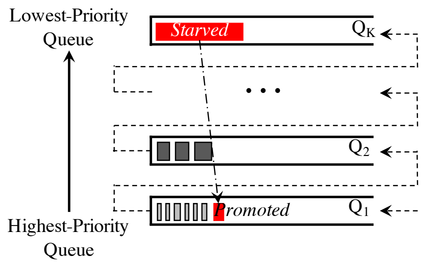


Figure 8: Discretized 2DAS with K queues. Starving jobs are periodically promoted to the highest priority queue.

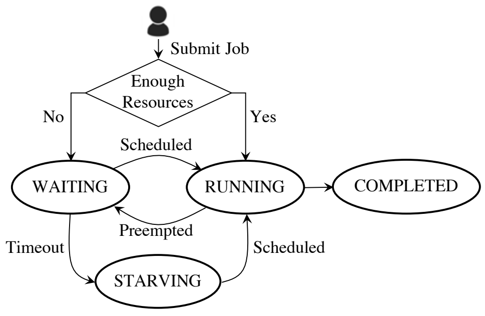


Figure 9: State transition diagram of a job in Tiresias.

Actions taken during four lifecycle events determine a job's priority (Figure 9).

- *Arrival:* If there are available resources, a new job enters the highest priority queue Q_1 when it starts.
- *Activity:* A job is demoted to Q_{i+1} from Q_i, when its $(W_J t_J)$ value crosses queue threshold Q_i^{hi}.
- *Starvation:* A job's priority is reset if it had been preempted for too long.
- *Completion:* A job is removed from its current queue upon completion.

The overall structure ensures that jobs with similar $(W_J t_J)$ values are kept in the same queue. Jobs with highly different $(W_J t_J)$ values are kept in different priority levels.

When LAS is used, jobs in the same queue are scheduled in a FIFO order of their start time (i.e., when they were first scheduled) without any risk of HOL blocking. Because of the all-or-nothing nature of DDL jobs, high-priority jobs without enough GPUs must be skipped over to increase utilization; as such, FIFO ordering on submission time instead of start time can lead to unnecessary preemptions.

The service quantum Δ in Gittins index is also discretized. For jobs in Q_i, Δ_i equals Q_i^{hi} which is the upper limit of Q_i. When a job consumes all its service quantum, it will be demoted to the lower priority queue. For Gittins index, jobs in the same queue are scheduled according to their Gittins index values. In the last queue, Q_K, Δ_K is set to ∞. In this extreme case, Gittins index performs similar to that of LAS, and jobs in the last queue are scheduled in the FIFO order.

Determining K and queue thresholds: While the discretization framework gives us the flexibility to pick K and corresponding thresholds, optimally picking them is an open problem [13, 17]. Instead of frequently solving an integer linear programming (ILP) [13] formulation or using a heavy-weight deep learning mechanism [15], we leverage the classic foreground-background queueing idea [33], which has been shown to perform well for heavy-tailed distributions. There are only two queues ($K = 2$) and only one threshold. Our sensitivity analysis shows that using $K = 2$ performs close to that of larger K values, ignoring preemption overheads. In practice, $K = 2$ limits the number of times a job can be preempted, which reduces job completion time.

Avoiding starvation: Using Discretized 2DAS, some jobs can starve if a continuous stream of small-and-short jobs keep arriving. This is because jobs in the same queue may be skipped over due to the lack of free GPUs. Similarly, jobs in lower priority queues may not receive sufficient GPUs either.

To avoid starvation, we promote a job to the highest-priority Q_1 if it has been WAITING for longer than a threshold: STARVELIMIT (Line 6).

This poses a tradeoff: while promotion can mitigate starvation, promoting too often can nullify the benefits of discretization altogether. To this end, we provide a single knob (PROMOTEKNOB) for the cluster operator to promote a job if its WAITING time so far (δ_J) is PROMOTEKNOB times larger than its execution time so far (t_J); i.e.,

$$\delta_J \geq \text{PROMOTEKNOB} * t_J$$

Setting PROMOTEKNOB $= \infty$ disables promotion and focuses on minimizing the average JCT. As PROMOTEKNOB becomes smaller, 2DAS becomes more fair, sacrificing the average JCT for tail JCT.

Note that both t_J and δ_J are reset to zero to ensure that a promoted job is not demoted right away.

3.3 Placement

Given a job J that needs PS_J parameter servers and W_J workers, if there are enough resources in the cluster, Tiresias must determine how to allocate them. More specifically, it must determine whether to consolidate the job's GPUs in as few machines as possible or to distribute them. The former is currently enforced in Microsoft production clusters; as a result, a job may be placed in the WAITQUEUE even if there are GPUs available across the cluster.

Taking this viewpoint to its logical extreme, we created an ILP formulation to optimally allocate resources in the cluster to minimize and balance network transfers among machines (see Appendix D). The high-level takeaways from such a solution are as follows. First and foremost, it is extremely slow to solve the ILP for a large-scale cluster with many DDL jobs. Second, from small-scale experiments, we found that explicitly minimizing and balancing the load of the network does not necessarily improve DL training performance.

Pseudocode 2 2DAS Scheduler

```
1: procedure 2D-LAS(Jobs $\mathbb{J}$, Queues $Q_1 \ldots Q_K$, Distribution $\mathbb{D}$)   ▷ §3.2
2:   $\mathbb{P} = \{\}$                                   ▷ Tracks jobs to preempt
3:   for all Job $J \in \mathbb{J}$ do
4:     if $J$ is RUNNING then
5:       $r_J$ = PRIORITY($J$, $\mathbb{D}$)               ▷ calculate job's priority
6:     if $J$ is WAITING longer than STARVELIMIT then
7:       Reset $t_J$
8:       Enqueue $J$ to $Q_1$                      ▷ Promote if $J$ is STARVING
9:   while Cluster has available GPUs do
10:    for all $i \in [1, K]$ do                       ▷ Prioritize across queues
11:      if $\mathbb{D}$ is not $\varnothing$ and $i \in [1, K-1]$ then
12:        Sort_Gittins_Index($Q_i$)                        ▷ Sort jobs in $Q_i$
13:      for all Job $J \in Q_i$ do       ▷ From the first $J$ in $Q_i$ to the end
14:        if Available GPUs $\geq W_J$ then                      ▷ $J$ can run
15:          Mark $W_J$ GPUs as unavailable
16:        else                                                ▷ $J$ cannot run
17:          $\mathbb{P} = \mathbb{P} \cup J$
18:          Preempt $J$ if it is already RUNNING
19:  for all Job $J \in \mathbb{J}$ and $J \notin \mathbb{P}$ do
20:    if $J$ is not already RUNNING then
21:      if $J$ was not profiled before then
22:        Profile $J$                                               ▷ §3.3.1
23:      Store $J$'s start time ▷ Used for FIFO in Discretized 2D-LAS
24:      Assign GPUs by comparing $S_J$ to PACKLIMIT             ▷ §3.3
25: end procedure
```

How important is consolidation? Given the infeasibility of an ILP-based formulation, we focused on developing a faster solution by asking a simple question: *which jobs benefit from consolidation?*

We found that the skew of the model structure (S_J) can be a good predictor. The DL models whose performance are sensitive to consolidated placement (Figure 3) have huge tensor(s); their largest tensor size dominates the whole model (Table 6). This is because messages sizes in model aggregation are closely related to the structure of the model. For example, a model in TensorFlow consists of many tensors. Each tensor is wrapped as a single communication message.[4] Therefore, the message size distribution in DDL depends on the tensor size distribution of the model. The tensor sizes are often unevenly distributed; sometimes there is a huge tensor which holds most of the parameters in those models. Hence, aggregating larger tensors suffers from network contention more severely, while transmissions of smaller tensors tend to interleave better with each other.

Leveraging this insight, we design Tiresias profiler that finds out the skew level of each model, which is then used by the Tiresias placement algorithm.

3.3.1 Profiler

For a given job J, Tiresias's profiler identifies the amount of skew in tensor distributions across parameter servers (S_J)

[4]Other frameworks may split each tensor into multiple messages, but still, these messages are sent out in clear batches for each tensor.

Table 2: Comparison of DL cluster managers.

	YARN-CS	Gandiva [41]	Optimus [34]	Tiresias(Gittins index)	Tiresias(LAS)
Prior Knowledge	None	None	JCT prediction	JCT distribution	None
Scheduling Algorithm	FIFO	Time-sharing	Remaining-time-driven	Gittins index	LAS
Scheduling Input	Arrival time	N/A	Remaining time	Attained service	Attained service
Schedule Dimensions	Temporal	None	Temporal	Spatial & temporal	Spatial & temporal
Job Priority	Continuous	Continuous	Continuous	Discretized queues	Discretized queues
Job Preemption	N/A	Context switch	Model checkpoint	Model checkpoint	Model checkpoint
Minimizing Average JCT	No	No	Yes	Yes	Yes
Starvation Avoidance	N/A	N/A	Dynamic resource	Promote to Q_1	Promote to Q_1
Job Placement	Consolidation	Trial-and-error	Capacity-based	Profile-based	Profile-based

without user input and in a framework-agnostic manner. The skew is a function of the tensor size distribution of the DL job and tensor-to-parameter server mapping of the DL framework (e.g., TensorFlow assigns tensors in a round-robin fashion). Instead of forcing users to design DL models with equal-sized tensors or making assumptions about the tensor assignment algorithm of a given DL framework, we aim to automatically identify the skew via profiling.

Because each parameter server periodically sends out its portion of the updated model to each worker (§2.1), observing these network communications can inform us of the skew. Given that most production DL jobs use RDMA (e.g., InfiniBand in Microsoft) for parameter server-worker communication, and to the best of our knowledge, there exists no RDMA-level traffic monitoring tool, we have built one for Tiresias.

Tiresias's profiler intercepts communication APIs – including the low-level networking APIs like RDMA `ibverbs` – in each machine to collect process-level communication traces. Whether a DDL job uses RDMA directly or through GPUDirect, Tiresias can capture detailed meta-data (e.g., message sizes) about all RDMA communications.

During the profiling run of a job, Tiresias aggregates information across all relevant machines to determine S_J for job J. Because each iteration is exactly the same from a communication perspective, we do not have to profile for too many iterations. This predictability also enables us to identify a job's iteration boundaries, model size, and skew characteristics. Tiresias's placement algorithm uses this information to determine whether the GPU allocation of a job should be consolidated or not.

3.3.2 The Placement Algorithm

Tiresias's placement algorithm compares S_J with a threshold (PACKLIMIT); if S_J is larger than PACKLIMIT, Tiresias attempts to consolidate the job in as few machines as possible. As explained above, a job with a large skew performs worse due to a skewed communication pattern if it is not consolidated. For the rest, Tiresias allocates GPUs in machines to decrease fragmentation. Albeit simple, this algorithm is very effective in practice (§5). It performs even better than the previous ILP-based design because the ILP cannot capture the different effects of consolidation on different models.

Determining PACKLIMIT: We rely on job history to periodically update PACKLIMIT. Currently, we use a simple linear classifier to periodically determine the PACKLIMIT value using a job's placement and corresponding performance as features. More sophisticated mechanism to dynamically determine PACKLIMIT can be an interesting future work.

3.4 Summary

Compared to Apache YARN's Capacity Scheduler (YARN-CS) and Gandiva, Tiresias aims to minimize the average JCT. Unlike Optimus, Tiresias can efficiently schedule jobs without or with partial prior knowledge (Table 2). Additionally, Tiresias can smartly place DDL jobs based on the model structure automatically captured by the Tiresias profiler.

4 Implementation

We have implemented Tiresias as a centralized resource manager. The Discretized 2DAS scheduler, the placement algorithm, and the profiler are integrated into the central master, and they work together to appropriately schedule and place DDL jobs. Similar to using current DDL clusters, users submit their DDL jobs with the resource requirements, primarily the number of parameter servers (PS_J) and the number of GPUs/workers (W_J). The resource manager then handles everything, from resource allocation when a job starts to resource reclamation when it completes.

As mentioned earlier, Tiresias makes job placement decisions based on profiling via a network monitoring library. This library is present in every server of the cluster and communicates with the central profiler so that Tiresias can determine the skew of each new DDL job.

Central master: In addition to starting new jobs and completing existing ones, a major function of the master is to preempt running jobs when their (GPU) resources are assigned to other jobs by the scheduler. Because of the iterative nature of DL jobs, we do not need to save all the data in GPU and main memory for job preemption. Currently, we use the checkpoint function provided by almost every DL framework and just

Table 3: DL jobs are put into bins by their number of GPUs (**S**mall and **L**arge) and their training time ($\mathbb{S}$hort and $\mathbb{L}$ong)

Bin	1 (**S**$\mathbb{S}$)	2 (**S**$\mathbb{L}$)	3 (**L**$\mathbb{S}$)	4 (**L**$\mathbb{L}$)
% of Jobs	63.5%	12.5%	16.5%	7.5%

save the most updated model for the preempted job. When a preemption is triggered, the job is first paused; then its chief worker checkpoints its model to a cluster-wide shared file system. When a paused job is resumed again by the scheduler, its most recent checkpoint will be loaded before it is restarted. The central master also determines a job's placement using the placement algorithm and the profiler.

Distributed RDMA monitoring: Because RDMA is widely used in GPU clusters for DDL jobs, we implement the profiler as a loadable library that intercepts RDMA `ibverbs` APIs. Therefore, it can record all the RDMA activities on each server, such as building connections, sending and receiving data. The RDMA-level information of all relevant workers and parameter servers are then aggregated at the central profiler. Based on the aggregated information (e.g., message size and the total amount of traffic), Tiresias can resolve the detailed model information of a given DDL job, including its skew. Though implemented for RDMA networks, the profiler can easily be extended to support TCP/IP networks by intercepting socket APIs.

5 Evaluation

We have deployed Tiresias on a 60-GPU cluster and evaluated it using experiments and large-scale simulations using production traces from Microsoft. The highlights are:

- In testbed experiments, Tiresias improves the average JCT by up to 5.5× and the makespan by 1.21× compared to YARN-CS. It also performs comparably to SRTF, which uses complete prior information (§5.2). Tiresias's benefits are due to job placement benefits for skewed DDL jobs and reduction in queueing delays during scheduling.
- Tiresias's benefits hold for large-scale simulation of the production trace from Microsoft (§5.3).
- Tiresias is robust to various configuration parameters and workload variations (§5.4).

In this section, Tiresias-G (Tiresias-L) represents Tiresias using the *Discretized 2D-Gittins index* (*Discretized 2D-LAS*).

5.1 Experimental Setup

Testbed. Our testbed consists of 15 4-GPU PowerNV 8335-GTB machines from IBM in the Michigan ConFlux cluster. Each machine has 4 NVIDIA Tesla P100 GPUs with 16 GB GPU memory, two 10-core (8 threads per core) POWER8 CPUs, 256 GB DDR4 memory, and a 100 Gbps EDR Mellanox InfiniBand adapter. There is also a high-performance cluster file system, GPFS [35], shared among those machines. In Tiresias, the checkpoint files used in job preemptions are written to and read from GPFS. The read and write throughput of GPFS from each machine is 1.2 GB/s.

Simulator. We developed a discrete-time simulator to evaluate Tiresias at large scale using a real job trace from Microsoft. It simulates all job events in Tiresias, including job arrival, completion, demotion, promotion, and preemption. However, it cannot determine job training time with the dynamic cluster environment; instead, it uses actual job completion times.

Workload. Given the scale of our GPU cluster, we generate our experimental workload of 480 DL/DDL jobs by scaling down the original job trace. Job requirements (number of GPUs, and training time) in our workload follow the distributions of the real trace. Half of these jobs are single-GPU DL jobs; the rest are DDL ones (40 2-GPU jobs, 80 4-GPU jobs, 90 8-GPU jobs, 25 16-GPU jobs, and 5 32-GPU jobs). The number of parameter servers in each DDL job is the same as its GPU number. Each model in Table 6 has 48 jobs. Each job has a fixed number of iterations to run. The training time of jobs varies from 2 mins to 2 hours. Jobs arrive following a Poisson process with an average inter-arrival time of 30 seconds. We run the jobs in synchronous data parallelism mode using TensorFlow 1.3.1 with RDMA extension and using model files from the TensorFlow benchmark [5].

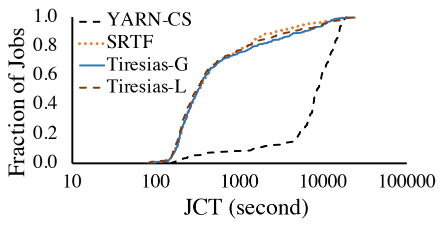
Job Bins. We category our jobs based on both their spatial (number of GPUs) and temporal (job training time) characteristics (Table 3). For the original trace, we consider a job to be *small* if it does not need more than 8 GPUs (Microsoft uses 8-GPU machines) and *short* if its training time is less than 4 hours. After scaling down, we consider a job to be small if it needs at most 4 GPUs (we are using 4-GPU machines) and short if it trains for less than 800 seconds.

Baselines. We compare Tiresias to an Apache YARNs capacity scheduler (YARN-CS) used in Microsoft (Appendix A). For comparison, we also implement an *SRTF* scheduler that has complete information, i.e., the eventual training time that in practice, cannot be obtained before running the job. Note that job durations are unknown to both Tiresias and YARN-CS. SRTF uses Tiresias's placement mechanism. We also comapre Tiresias with the time-sharing scheduler in Gandiva [41] in the large-scale simulation.

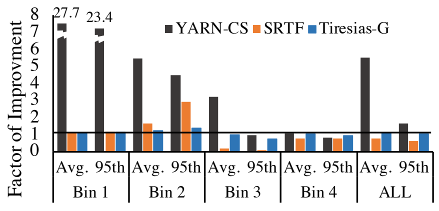
Metric. Our key metric is the improvement in the average JCT (i.e., time from submission to completion):

$$\text{Factor of Improvement} = \frac{\text{Duration of an Approach}}{\text{Duration of Tiresias-L}}$$

To clearly present the performance of Tiresias, unless otherwise specified, the results of all schedulers (including Tiresias-G) are normalized by that of Tiresias-L. Factor of improvement (FOI) greater than 1 means Tiresias-L is performing better, and vice versa.



(a) Individual job completion times



(b) Summarized results

Figure 10: Improvements in the average JCT using Tiresias w.r.t. YARN-CS and SRTF.

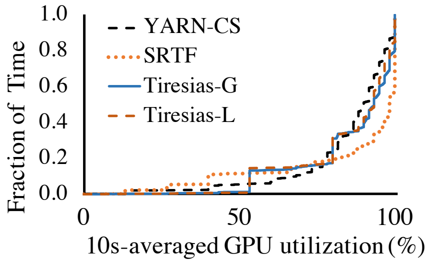


Figure 11: Cluster-wide GPU utilization.

5.2 Tiresias in Testbed Experiments

In testbed experiments, we compare the performance of YARN-CS, SRTF, Tiresias-G and Tiresias-L. For Tiresias, there are two priority queues with a threshold of 3200 GPU seconds. The PROMOTEKNOB for avoiding starvation is disabled in Testbed experiments.

5.2.1 JCT Improvements

Tiresias-L achieves 5.5× improvement in terms of the average JCT w.r.t. to YARN-CS (Figure 10). If we look at the median JCT, then Tiresias-L is 27× better than YARN-CS. Tiresias-G has almost the same performance as Tiresias-L (1.06× in average, 1.05× in median). Its negligible performance loss is due to more job preemptions (§5.2.4). Half of all jobs avoid severe queueing delays using Tiresias. Moreover, Tiresias is not far from SRTF either.

The key idea of Tiresias's scheduler is avoiding queueing delays to small or short jobs, thus saving them from large or long jobs. When using Tiresias-L (Tiresias-G), the average JCT of jobs in Bin1 (**S**$\mathbb{S}$) is just 300 (330) seconds, which is 27.6× (25.2×) better than that using YARN-CS. On the other hand, jobs in Bin4 (**L**$\mathbb{L}$) have almost the same average JCT in both Tiresias and YARN-CS.

5.2.2 Cluster-Wide GPU Utilization

Figure 11 shows the averaged GPU utilizations of our cluster over time. While there are some small variations, overall utilizations across solutions look similar. However, Tiresias reduces the makespan compared to YARN-CS. The makespan of Tiresias-L (27400 seconds) was 1.21× smaller than that of YARN-CS (33270 seconds), and it was similar to Tiresias-G (27510 seconds) and SRTF (28070 seconds).

5.2.3 Sources of Improvements

Smaller queueing delays. Tiresias's scheduler can reduce the average queueing delay of all jobs (Table 4), especially for small and short jobs. The average queueing delay is reduced from over 8000 seconds to around 1000 seconds when comparing YARN-CS and Tiresias. More importantly, half of the jobs are just delayed for less than or equal to 13 (39) seconds in Tiresias-L (Tiresias-G), which is negligible compared to the median delay in YARN-CS. Note that while Tiresias's average queueing delay is higher than SRTF, smaller jobs actually experience similar or shorter delays.

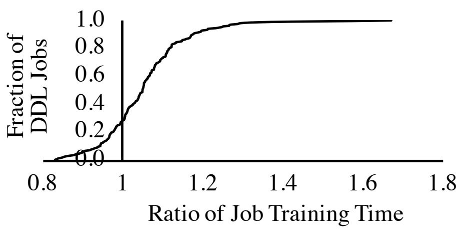


Figure 12: Performance improvement from job placement in Tiresias-L. We pick all the DDL jobs and compare their training times when Tiresias-L is running with and without placement.

Table 4: Queueing delays for DDL jobs for different solutions.

	Average	Median	95th
YARN-CS	8146s	7464s	15327s
SRTF	593s	32s	3133s
Tiresias-G	1005s	39s	7933s
Tiresias-L	963s	13s	7755s

Faster training. Albeit smaller, another source of performance improvement is Tiresias's job placement algorithm. To illustrate this, we rerun the experiment using Tiresias-L but without its profiler; i.e., jobs are randomly placed on the cluster. We compare the training time of DDL jobs in Tiresias-L without job profiling versus in the original Tiresias-L. We use the ratio of training time in random placement to Tiresias-L as the factor of improvement. In Figure 12, large ratio means random placement slows down the training, and vice versa; single-GPU jobs are excluded from the figure. Tiresias-L achieves up to 1.67× improvement w.r.t. random placement, because it can identify sensitive jobs and place them on minimal number of machines for better performance. Fewer than 30% of DDL jobs experience limited performance loss.

Because of the highly-skewed job distribution and the variety of model types, the major improvement comes from the job scheduling by avoiding HOL blocking of small/short jobs by the large/long ones.

5.2.4 Overheads

Because Tiresias uses preemption in its scheduling algorithm, its major overhead comes from preempting DDL jobs. The Discretized 2DAS scheduler in Tiresias provides *discretized* priority levels to jobs. Hence, two cases trigger job preemptions in Tiresias: job arrivals/promotions and demotions that change the priority queue of a job. In our experiments,

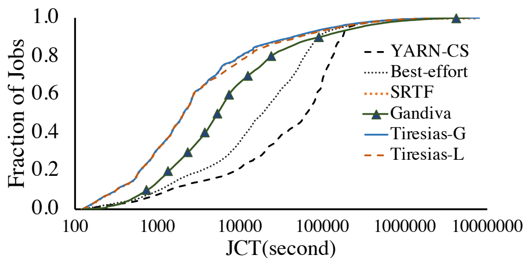


Figure 13: JCT distributions using different solutions in the trace-driven simulation. The x-axis is in logarithmic scale.

Table 5: Improvements in JCT using Tiresias in simulation. Numbers are normalized by that of Tiresias-L.

	Average	Median	95th
YARN-CS	2.41×	30.85×	1.25×
Best-effort	1.50×	9.03×	1.08×
SRTF	1.00×	1.00×	0.84×
Gandiva	2.00×	2.59×	2.08×
Tiresias-G	0.97×	1.00×	0.85×

Tiresias-L spent 13724 seconds performing 221 preemptions; Tiresias-G triggered 297 preemptions with 17425 seconds overhead in total. There are more preemptions in Tiresias-G because jobs in the same queue are sorted based on their Gittins index value at every event. In Tiresias-L, jobs will not be re-sorted because of FIFO ordering.

In contrast, job priorities in SRTF are *continuous*. Whenever short jobs come in, jobs that with longer remaining time (lower priorities) may be preempted due to lack of resources. Overall, SRTF spent 18057 seconds for 316 preemptions.

Note that the exact overhead of each preemption depends on the specific job and cluster conditions.

5.3 Tiresias in Trace-Driven Simulations

Here we evaluate Tiresias's performance on the Microsoft job trace. We compare it against YARN-CS, SRTF, and Best-effort, where Best-effort is defined as YARN-CS but without HOL blocking – i.e., it allows small jobs to jump in front of large jobs that do not have enough available GPUs.

5.3.1 Simulator Fidelity

We replayed the workload used in our testbed experiments in the simulator to verify the fidelity of our simulator. We found the simulation results to be similar to that of our testbed results – 5.11× (1.50×) average (95th percentile) improvement w.r.t. YARN-CS, 0.74× (0.55×) w.r.t. SRTF, and 1.01× (1.13×) w.r.t. Tiresias-G. Because the simulator cannot capture overheads of preemption, the impact of placement, or cluster dynamics, the results are slightly different.

5.3.2 JCT Improvements

We then simulated the job trace from Microsoft to identify large-scale benefits of Tiresias. Tiresias-L improves the average JCT by 2.4×, 1.5×, and 2× over YARN-CS, Best-effort, and Gandiva, respectively (Table 5). In addition, Tiresias-L reduces the median JCT by 30.8× (9×) w.r.t. YARN-CS (Best-effort). This means half of the Microsoft jobs would experience significantly shorter queueing delays using Tiresias. Compared to Tiresias-L, Tiresias-G has almost the same (median JCT) or slightly better (average and 95th percentile JCT) performance. More importantly, Tiresias performs similar to SRTF that uses complete knowledge.

5.4 Sensitivity Analysis

Here we explore Tiresias's sensitivity to K (number of priority queues), queue thresholds (server quantum Δ), and PROMOTEKNOB. By applying the Discretized 2D-LAS algorithm, Tiresias relies on K and corresponding thresholds to differentiate between jobs. In this section, we use (K, threshold1, threshold2, ...) to represent different settings in Tiresias. For example, (2, 1h) means Tiresias has 2 priority queues and the threshold between them is 1 hour GPU time.

5.4.1 Impact of Queue Thresholds

We use Tiresias with K=2 and increase the threshold between the two priority queues (Figure 14a and 15a). We observe that (2, 0.5h) is slightly worse than others who have larger thresholds in terms of the average JCT in Tiresias-L. When the threshold is larger than or equal to 1 hour, Tiresias-L's performance almost does not change. For Tiresias-G, different Δ values have almost the same performance. These are because 1h GPU time can cover more than 60% of all the jobs.

5.4.2 Impact of K (number of priority queues)

Next, we examine Tiresias's sensitivity to K. We evaluate Tiresias with K set to 2, 3 and 4, and pick the best thresholds in each of them. The number of priority queues does not significantly affect Tiresias (Figure 14b and 15b). The 3- and 4-queue Tiresias only improves the average JCT by 1% in comparison to the 2-queue Tiresias-L.

5.4.3 Impact of PROMOTEKNOB

This simulation is based on (2, 1h). We pick the initial PROMOTEKNOB as 1, and increase it by the power of 2. When PROMOTEKNOB is infinite, Tiresias does not promote. Smaller PROMOTEKNOB means more frequent promotions of long-delayed jobs back to the highest priority queue. For Tiresias-G, the maximal JCT is cut down by PROMOTEKNOB (Figure 15c). However this trace is not sensitive to the different value of PROMOTEKNOB. In Figure 14c, the 95th JCT minutely changes when we use smaller PROMOTEKNOB in Tiresias-L – the key reason PROMOTEKNOB has little impact for this trace is due to its heavy-tailed nature [33].

6 Discussion and Future Work

Formal analysis. Although Discretized 2DAS has advantages in minimizing the average JCT of DL jobs, formal analyses are still needed to precisely present its applicable boundaries (in terms of cluster resources and DL jobs' requirements). This will simplify Tiresias configuration in practice.

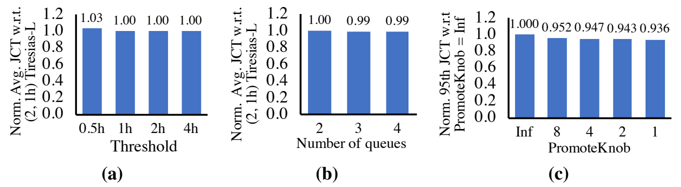


Figure 14: Sensitivity analysis of Tiresias-L. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

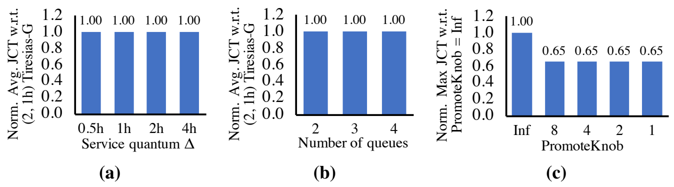


Figure 15: Sensitivity analysis of Tiresias-G. The queue settings in (b) are (2, 1h), (3, 1h, 2h), and (4, 1h, 2h, 4h) for each bar.

Lightweight preepmption. Existing preemption primitives for DL jobs are time-consuming. To reduce the number of preemptions, Tiresias adopts priority discretization using MLFQ (§3.2.3). A better way of preempting DL jobs has been proposed in Gandiva [41]. However, that approach requires DL framework modifications. At the same time, its overhead is still non-negligible. With lightweight preemption mechanisms, many classic and efficient algorithms in network flow and CPU scheduling can be applied for DDL scheduling.

Fine-grained job placement. Tiresias's profile-based placement scheme coarsely tries to avoid network transfers when necessary. However, there can be interferences within the server (e.g., on the PCIe bus) when too many workers and parameter servers are collocated. Further investigations on how placement can affect job performance are required. To this end, possible approaches include topology-aware schemes [10] and fine-grained placement of computational graphs in DL jobs [32].

7 Related Work

Cluster Managers and Schedulers. There are numerous existing resource managers and schedulers for CPU-based clusters for heterogenous workloads [20, 22–24, 26, 29, 38–40, 46] or for traditional machine learning jobs [27, 37, 44]. As explained in Section 1, these frameworks are not designed to handle the unique characteristics of DDL jobs – e.g., all-or-nothing task scheduling, and unpredictable job duration and resource requirements – running on GPU clusters.

Resource Management in DDL Clusters. Optimus [34] is an online resource scheduler for DDL jobs on GPU clusters. It builds resource-performance model on the fly and dynamically adjusts resource allocation and job placement for minimizing the JCT. It is complementary to Tiresias in terms of job placement, because the latter focuses on the efficiency of the initial job placement based on job characteristics, while the former performs online adjustment according to a job's realtime status. However, Optimus assumes that the remaining time of a DL job is predictable, which is not always true in practice (§2.2). Tiresias can schedule jobs without any or with partial prior knowledge, and it does not rely on such assumptions. Gandiva [41] is a resource manager for GPU clusters that gets rid of the HOL blocking via GPU time sharing. However, the time-slicing scheduling approach in Gandiva brings limited improvement in terms of the average JCT.

Resource Management with Partial or No Information. To the best of our knowledge, Tiresias is the first cluster scheduler for DDL training jobs that minimizes the average JCT with partial or no information. While similar ideas exist in networking [13, 17] and CPU scheduling [12, 18], GPU clusters and DDL jobs provide unique challenges with high preemption overheads and all-or-nothing scheduling. There exist all-or-nothing gang schedulers for CPU, but they are not information-agnostic. While fair schedulers do not require prior knowledge [6, 7, 43], they cannot minimize the average JCT. Similar to the Gittins index policy, shortest-expected-remaining-processing-time (SERPT) [36] just needs partial knowledge of job durations. However, the Gittins index policy is proven to be better because it prioritizes a larger number of potentially shorter jobs [36].

8 Conclusion

Tiresias is a GPU cluster resource manager that minimizes distributed deep learning (DDL) jobs' completion times with partial or no a priori knowledge. It does not rely on any intermediate DL algorithm states (e.g., training loss values) or framework specifics (e.g., tensors-to-parameter server mapping). The key idea in Tiresias is the 2DAS scheduling framework that has two scheduling algorithms (*Discretized 2D-LAS* and *Discretized 2D-Gittins index*). They can respectively minimize the average JCT with no and partial prior knowledge. Additionally, Tiresias's profile-based job placement scheme can maintain the resource (GPU) utilization of cluster without hurting job performance. Compared to a production solution (Apache YARN's Capacity Scheduler) and a state-of-the-art DDL cluster scheduler (Gandiva), Tiresias shows significant improvements in the average JCT.

Acknowledgments

Special thanks go to the ConFlux team from the University of Michigan, especially Karthik Duraisamy and Todd Raeker, for reserving enough GPU servers to make Tiresias experiments possible. We would also like to thank the anonymous NSDI reviewers, our shepherd, KyoungSoo Park, and SymbioticLab members, especially Yiwen Zhang, for their constructive comments and feedback that helped improve the paper. This work was supported in part by NSF grants CCF-1629397, CNS-1563095, and CNS-1617773. Computing resources were provided by the NSF via OAC-1531752.

References

[1] Amazon EC2 Elastic GPUs. https://aws.amazon.com/ec2/elastic-gpus/.

[2] AutoML. http://www.ml4aad.org/automl/.

[3] GPU-Accelerated Microsoft Azure. https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/.

[4] GPU on Google Cloud. https://cloud.google.com/gpu/.

[5] TensorFlow Benchmark Code. https://github.com/tensorflow/benchmarks.

[6] YARN Capacity Scheduler. http://goo.gl/cqwcp5.

[7] YARN Fair Scheduler. http://goo.gl/w5edEQ.

[8] S. Aalto, U. Ayesta, and R. Righter. On the gittins index in the m/g/1 queue. *Queueing Systems*, 63(1-4):437, 2009.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[10] M. Amaral, J. Polo, D. Carrera, S. Seelam, and M. Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *SC*, 2017.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[12] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Scheduling: The multi-level feedback queue. In *Operating Systems: Three Easy Pieces*. 2014.

[13] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *NSDI*, 2015.

[14] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[15] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.

[16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[17] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[18] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Spring Joint Computer Conference*, pages 335–344, 1962.

[19] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *EuroSys*, 2016.

[20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[21] J. Gittins, K. Glazebrook, and R. Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[22] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *OSDI*, 2016.

[23] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[24] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[25] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE CVPR*, 2016.

[26] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[27] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, 2015.

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

[29] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[30] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[31] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *arXiv preprint arXiv:1805.07891*, 2018.

[32] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[33] M. Nuyens and A. Wierman. The Foreground–Background queue: A survey. *Performance Evaluation*, 65(3):286–307, 2008.

[34] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[35] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[36] Z. Scully, M. Harchol-Balter, and A. Scheller-Wolf. Soap: One clean analysis of all age-based scheduling policies. In *SIGMETRICS*, 2014.

[37] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *IEEE Smart Computing (SMARTCOMP)*, 2017.

[38] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.

[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC*, 2013.

[40] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[41] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[42] D. Yu, A. Eversole, M. Seltzer, K. Yao, O. Kuchaiev, Y. Zhang, F. Seide, Z. Huang, B. Guenter, H. Wang, J. Droppo, G. Zweig, C. Rossbach, J. Gao, A. Stolcke, J. Currey, M. Slaney, G. Chen, A. Agarwal, C. Basoglu, M. Padmilac, A. Kamenev, V. Ivanov, S. Cypher, H. Parthasarathi, B. Mitra, B. Peng, and X. Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Research, October 2014.

[43] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[44] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *ACM SoCC*, 2017.

[45] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *USENIX ATC*, 2017.

[46] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. In *VLDB*, 2014.

A Characteristics of Production Cluster

We describe Project Philly [28], the cluster manager in one of Microsoft internal production clusters, referred as *P*. *P* is shared by several production teams that work on projects related to a search engine. It is managed by an Apache YARN-like resource manager, which places and schedules DDL jobs submitted by users via a website/REST API front end. *P* supports various framework jobs, including TensorFlow, Caffe and CNTK. In 2016, *P* consisted of around 100 4-GPU servers. In 2017, due to the surging demand of running DDL, *P* is expanded by more than 250 8-GPU servers. So, the total number of GPUs has grown by 5×. Servers in P are interconnected using a 100-Gbps RDMA (InfiniBand) network.

We collect traces from *P* over a 10-week period from Oct. 2017 to Dec. 2017. This cluster runs *Ganglia* monitoring system, which collects per-minute statistics of hardware usage on every server. Since some jobs are quickly terminated because of bugs in user's job configuration, we only show the data of jobs that run for at least one minute. Also, we collect the per-job logs output by the DL framework which include the time for each iteration and the model accuracy along the running time. The network-level activities are monitored by Tiresias profiler which is explained in Section 3.3.1, that logs every RDMA network operation, e.g., the send and receive of every message,and their timestamps. In addition, we add hooks that intercept the important function calls in a DDL framework, e.g., the start of an iteration or aggregation, and log their timestamps.

Although we cannot disclose the details of proprietary DL models in *P*, we present the results of several public and popular models, some of which are also run in *P*.

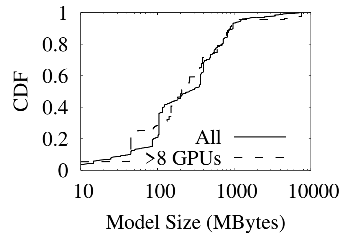


Figure 16: The CDF of DL model sizes being trained.

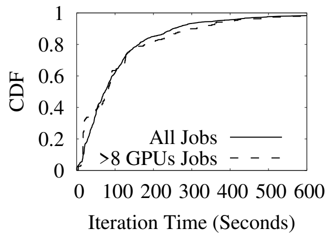


Figure 17: The CDF of average iteration time per job.

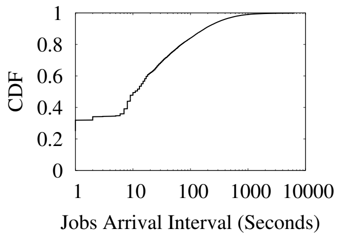


Figure 18: The CDF of DDL job arrival intervals.

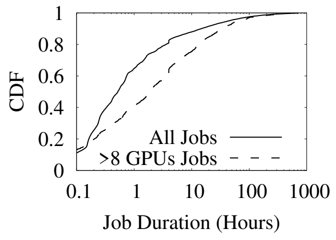


Figure 19: The CDF of DDL job duration.

Large and different DL model sizes. As shown in Figure 16, production DL models range from a few hundreds of megabytes to a few gigabytes. The model size distribution is rather independent from the number of GPUs used. According to the cluster users, the number of GPUs used often depends more on the training data volume and the urgency of jobs, and less on model sizes. Larger model sizes mean heavier communication overhead per iteration in distributed training. The largest one is 7.5GB. It may cause network congestion even with 100 Gbps network and greatly hurt the job-level performance. [5] To minimize this overhead, existing job placement strategy intuitively consolidates DDL jobs as much as possible.

A staggering increase in the number DDL jobs. We compare the number of DDL jobs (with at least two GPUs) during ten weeks from Oct. 2017 to Dec. 2017, and the number of DDL jobs during the same ten weeks in 2016. The total number of DDL jobs has grown by 10.5× year over year. We refer to jobs using more than 8 GPUs as "large jobs," since such jobs have to run on multiple servers (8 GPUs per server in *P*). Large jobs have grown by 9.4×. The largest job run on 128 GPUs in 2017, while the number was 32 GPUs in 2016. We expect this trend to continue as DL jobs are trained on ever larger data sets.

Long job queuing time in production clusters. We also observe that the number of DDL jobs is increasing faster than the speed of cluster expansion. As a result, some jobs have to wait in a queue when the cluster is overloaded. From the trace, we see the average queuing delay of all jobs is 4102 seconds! A brute-force solution is to add GPUs as fast as the demand. However, this poses significant monetary costs – each 8-GPU server in P costs around 100K US Dollars based on public available GPU price. Thus, the DDL cluster service providers are seeking ways to improve job completion time (including the queuing time) given limited GPU resources.

Unpredictable job arrivals. Since the cluster is shared by multiple teams and jobs are submitted on demand, the job arrival intervals are naturally unpredictable. Figure 18 shows that the job arrival interval is mostly less than one hour. Many arrival intervals are less than one second, suggesting that they are generated by AutoML to sweeping hyperparameters.

[5]Section 3.3 shows that in fact it mostly depends on the model structure.

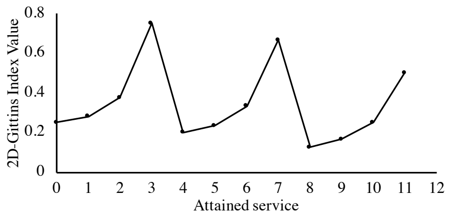


Figure 20: 2D-Gittins index value in §3.2.2. Jobs have required service 4, 8, and 12, each with probability $1/3$.

Various aggregation frequency depending on algorithm demands. The communication overhead also depends on how frequently aggregations are performed, which depends on the minibatch sizes. The size of minibatches is determined by the model developers – the larger the minibatches, the larger the learning step, which may help the learning process avoid local optimas but risk final convergence due to too-large steps. Thus, it is usually chosen by the users based on the requirements of specific models.

Figure 17 shows that the per iteration time varies significantly across jobs. However, the distribution of large jobs is very close to all jobs. This means that users probably do not choose minibatch sizes based on how many GPUs are used in each job.

B Characteristics of Popular DNN models

In Table 6, we pick 10 popular DNN models and present the details of their model structures for their TensorFlow implementations [5]. For the VGG family and AlexNet, the size of each model is dominated by its largest tensor. For the rest, their tensor size distributions are less skewed.

C 2D-Gittins Index Value in Section 3.2.2

When using 2D-Gittins index scheduling algorithm, the priorities of the jobs is determined by their corresponding 2D-Gittins index value mapped to their attained service. The three jobs in Figure 7 follow the same 2D-Gittins index in Figure 20.

Table 6: Characteristics of 10 popular DNN models in TensorFlow

Model	Model size (MB)	#Tensors	#Large tensors ($\geq$ 1MB)	Largest tensor size (MB)	Largest tensor ratio
VGG19	548.1	39	15	392.0	71.5%
VGG16	527.8	33	12	392.0	74.3%
VGG11	506.8	23	9	392.0	77.3%
AlexNet	235.9	17	7	144.0	61.0%
ResNet152	230.2	778	48	9.0	3.9%
ResNet101	170.4	523	35	9.0	5.3%
ResNet50	97.7	268	18	9.0	9.2%
Inception4	162.9	599	81	5.9	3.6%
Inception3	91.0	397	21	7.8	8.6%
GoogleNet	26.7	117	7	3.9	14.6%

D ILP Formula for DDL Placement

When placing a DDL job on to a shared GPU cluster, the network traffic generated by that job affects not only itself, but also all the jobs that share the same machines or network links. The existing network status can affect the newly-placed DDL job as well. Therefore, the objective of placing a DDL job is to maximize the overall performance of the entire cluster. To achieve this, we have to minimize the total network traffic and also balance the network load on individual machines in the cluster. In our ILP formulation, the objective function is to minimize the maximal network load of machines when placing a new DDL job onto the cluster.

By default, we assume all DDL jobs have the same number of parameter servers (PS) and GPU worker, which is a common practice [45]. Actually, changing number of parameter servers does not affect the total amount of data in aggregation in the parameter server architecture. There are N GPU nodes in the cluster. N_i is the i-th node whose network traffic from existing DDL jobs is t_i. And N_i has g_i free GPUs before placing any new jobs. We assume a new DDL job J with model size M is going to be placed. There are W GPU workers and K parameter servers in it. The total size of tensors hosted by the j-th parameter server is s_j. For J, w_i is the number of GPU workers placed on N_i. p_{ji} is a binary variable. It will be 1 if the j-th parameter server is placed on N_i, and vice versa.

The total network traffic of N_i comes from three parts: (1) existing traffic, (2) traffic from the workers of J on it, and (3) traffic from the parameter servers of J on it. For collocated parameter servers and workers, the traffic between them has to be deducted. Therefore, the total network traffic T_i is:

$$T_i = t_i + w_i \cdot (M - \sum_{j \in K} p_{ji} \cdot s_j) + \sum_{j \in K} p_{ji} \cdot s_j \cdot (W - w_i)$$

The overall objective can then be expressed as:

$$\text{minimize} \quad \max_{i \in N} \{T_i\}$$

The corresponding constraints are the following:

$$\forall_{i \in N} w_i \leq g_i \tag{1}$$

$$\sum_{i \in N} w_i = W \tag{2}$$

$$\forall_{j \in K} \sum_{i \in N} p_{ji} = 1 \tag{3}$$

The first one is GPU resource constraints on all nodes. The second one requires the consistency of total number of GPU workers in J. The last one means every parameter server must have exactly one host machine. Of course, more constraints, such as CPU and host memory limitations, can be added into this ILP formulation.