Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning

Shubham Chaudhary
t-shucha@microsoft.com
Microsoft Research India

Ramachandran Ramjee
ramjee@microsoft.com
Microsoft Research India

Muthian Sivathanu
muthian@microsoft.com
Microsoft Research India

Nipun Kwatra
nipun.kwatra@microsoft.com
Microsoft Research India

Srinidhi Viswanatha
srinidhi.viswanatha@microsoft.com
Microsoft Research India

Abstract

We present *Gandiva$_{fair}$*, a distributed, fair share scheduler that balances conflicting goals of efficiency and fairness in GPU clusters for deep learning training (DLT). *Gandiva$_{fair}$* provides performance isolation between users, enabling multiple users to share a single cluster, thus, maximizing cluster efficiency. *Gandiva$_{fair}$* is the first scheduler that allocates cluster-wide GPU time fairly among active users.

Gandiva$_{fair}$ achieves efficiency and fairness despite cluster heterogeneity. Data centers host a mix of GPU generations because of the rapid pace at which newer and faster GPUs are released. As the newer generations face higher demand from users, older GPU generations suffer poor utilization, thus reducing cluster efficiency. *Gandiva$_{fair}$* profiles the variable marginal utility across various jobs from newer GPUs, and transparently incentivizes users to older GPUs by a novel resource trading mechanism that maximizes cluster efficiency without affecting fairness guarantees of any user. With a prototype implementation and evaluation in a heterogeneous 200-GPU cluster, we show that *Gandiva$_{fair}$* achieves both fairness and efficiency under realistic multi-user workloads.

ACM Reference Format:
Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3342195.3387555

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387555

1 Introduction

~~Love~~ Resource only grows by sharing. You can only have more for yourself by giving it away to others.

- Brian Tracy

Several products that are an integral part of modern living, such as web search and voice assistants, are powered by deep learning. Companies building such products spend significant resources for deep learning training (DLT), managing large GPU clusters with tens of thousands of GPUs. Multiple teams compete for these GPUs to run DLT jobs. Partitioning GPUs statically across multiple users provides predictability and performance isolation, but results in poor cluster utilization.

A single shared cluster across all users is attractive for overall efficiency, but in order to be practical, such a cluster must guarantee that each user will get at least the same performance as they would have with a statically partitioned cluster. In other words, if a user *A* was entitled to a 20% global share of GPUs, regardless of other jobs/users running on the shared cluster, the effective performance of user *A* in the shared cluster must be at least the same as if *A* ran on a dedicated cluster with 20% of the GPUs. If user *A* is unable to utilize their quota, the unused capacity must be shared across other active users, thus, maximizing cluster efficiency.

An additional dimension that complicates sharing is hardware heterogeneity, a particularly stark problem in GPU clusters. As newer GPU generations get released at a rapid pace, large clusters, over time, typically have a mix of heterogeneous GPUs. Users prefer newer generations of GPUs for their higher performance, thus leaving the older generations under-utilized. With a single shared cluster, the scheduler needs to intelligently allocate GPUs of different generations across users, to maximize efficiency while ensuring fairness.

Unfortunately, while there have been several DLT job schedulers proposed in the literature [19, 33, 41], none of them support user fairness even in homogeneous clusters. In fact, none of them have a notion of users and operate at a job-level, either focusing on improving cluster efficiency [41] or job-completion time [19, 33]. Even schedulers used in companies today [25] simply divide a cluster *statically* into virtual clusters to isolate one group from another, resulting in poor efficiency.

In this paper, we present *Gandiva*$_{fair}$, a scheduler that guarantees cluster-wide fair share of GPU minutes across users in a shared cluster of heterogeneous GPUs, while ensuring cluster efficiency by distributing unused quota fairly among other active users. Like traditional fair-share schedulers [39], *Gandiva*$_{fair}$ uses the notion of *tickets* to provide fair-share of resources in a cluster to a particular user. *Gandiva*$_{fair}$ assumes that GPU is the only dominant resource for DLT jobs [33, 41]. However, if network/CPU can also become bottlenecks, one could extend *Gandiva*$_{fair}$ to incorporate a fairness metric like dominant resource fairness [14] for apportioning multiple resources fairly.

While big-data schedulers like Yarn [38] also support user fairness, there are two key differences between big-data jobs and DLT jobs that make big-data schedulers unsuitable for the deep learning setting. First, unlike Yarn, DLT job scheduler needs to be *gang-aware*, i.e, they need to schedule all the GPUs required by a DLT job in an all-or-nothing manner. Second, big-data schedulers resort to job *preemption* during over-subscription. However, DLT jobs are typically long-running and their preemption can result in loss of hours/days of job state. Instead, *Gandiva*$_{fair}$ relies on *job migration* as a key primitive for enforcing fairness without forfeiting job state.

Gandiva$_{fair}$ achieves fairness and efficiency using three key techniques. First, it uses a novel split, gang-aware stride scheduler to enforce fairness at short time-scales (time-quantum of few minutes). A central scheduler is responsible for gang-aware scheduling of large jobs, jobs that require large number of GPUs and span multiple servers, while a local per-server gang-aware scheduler schedules small jobs, jobs whose GPU requirement fit within that server. Such a split design allows the central scheduler to coordinate multiple servers, necessary for scheduling a large job in a gang-aware manner, while the distributed, local schedulers managing small jobs allow scalability.

However, the split scheduler by itself cannot guarantee fairness as jobs arrive and depart. Thus, a second component of *Gandiva*$_{fair}$ is a load balancer that uses job migration as a key mechanism to distribute jobs, weighted by their tickets, evenly throughout the cluster.

Third, *Gandiva*$_{fair}$ addresses hardware heterogeneity by employing a novel technique of *automatic resource trading* across users. From a fairness perspective, *Gandiva*$_{fair}$ maps user tickets to a proportional share of each generation of GPUs in the cluster. For example, if the cluster had 5000 V100 and 10000 P100 GPUs, a user with a 20% share of the cluster would be entitled to 1000 V100s and 2000 P100s, and *Gandiva*$_{fair}$ guarantees that the user's performance would be at least the same as a dedicated cluster with 1000 V100s and 2000 P100s. While preserving such guarantee, *Gandiva*$_{fair}$ uses automatic trading to maximize cluster efficiency. The key insight behind *Gandiva*$_{fair}$'s resource trading is to leverage the *varying marginal utility* of faster GPUs for different DLT jobs. As an example, if jobs of user *A* achieve a speedup of 1.25x on V100 compared to a K80 GPU, while jobs of user *B* achieve a 5x speedup on V100, user *B* has a 4x higher marginal utility, and thus would be better off trading 4 of his K80 GPUs in exchange for 1 V100; both users benefit from this trade, and cluster efficiency also improves. As we show in Section 2, the factor speedup varies across a wide spread between jobs, enabling such win-win trades that maximize cluster efficiency. Furthermore, this mechanism prevents users from gaming these incentives since it leverages the incentive-compatible benefits of second-price auctions [32].

We have implemented *Gandiva*$_{fair}$ as a custom scheduler in Kubernetes [10], and evaluate it on a cluster of 200 GPUs, running a wide diversity of jobs across multiple users. We show that *Gandiva*$_{fair}$ achieves inter-user fairness at the cluster-wide level, while also maximizing cluster efficiency. We also demonstrate the efficacy of automatic resource trading in *Gandiva*$_{fair}$, that results in faster progress rate for user jobs compared to a fairness-only scheduler without trading.

We make the following contributions in the paper:

- We present the first cluster scheduler for deep learning training jobs that guarantees user-level fair share of cluster-wide GPU minutes, while maximizing cluster efficiency.
- We demonstrate that migration can be used as a first-class primitive to achieve cluster-wide fairness without resorting to preemption.
- We present an automated trading strategy to handle GPU heterogeneity, by leveraging the variable marginal utility of faster GPUs for various jobs, thus improving efficiency while ensuring fairness.
- Using a prototype implementation on a 200-GPU heterogeneous cluster, we show that *Gandiva*$_{fair}$ is able to achieve its goals of fairness and efficiency under large-scale multi-user workloads.

2 Motivation

In this section, we motivate the design of *Gandiva$_{fair}$* by highlighting the challenges involved in supporting fairness and heterogeneous GPUs for DLT jobs.

2.1 Fairness

In order to illustrate the various options available to a fair scheduler and motivate *Gandiva$_{fair}$*'s design choice, consider the simple example shown in Figure 1. In this example, the cluster consists of two servers with four GPUs each. All GPUs are of the same model (homogeneous). Let two 2-GPU jobs of two users, A and B, be running on the eight GPUs of the cluster.

Inter-user fairness. One of the goals of *Gandiva$_{fair}$* is inter-user fairness. For simplicity, let us assume that all users have the same number of tickets. Then, a scheduler is inter-user fair if each active user receives a resource allocation of at least the total cluster GPU resources divided by the number of active users. In case an active user does not have sufficient number of jobs to make use of their fair-share, then the scheduler should allocate enough resources to satisfy that user and then recursively apply the fairness definition to the remaining resources and active users. In this example, the cluster has 8-GPUs and user A and user B have been allocated four GPUs each. Thus, the allocation is inter-user fair.

Let us now assume that a new 2-GPU job arrives for user C. User C's fair share in this example is 8 GPUs/3 active users = 2.66 GPUs (i.e., 2 GPUs + 2/3rd of time on one GPU) but since the user has only one 2-GPU job, user C's fair share is 2 GPUs. After allocating user C's fair share of 2-GPUs, for inter-user fairness, the remaining 6 GPUs needs to be divided equally between user A and user B, resulting in 3 GPUs for each of them in aggregate.

Now consider the various fairness options provided by current schedulers in this scenario. Schedulers like Optimus [33] or Tiresias [19] don't have inter-user fairness as a goal and optimize their scheduling decisions based on minimizing job completion time. So, they either allow user C's job to stay in the queue or move one of the existing jobs back to queue and schedule user C's job in its place. Apart from Optimus or Tiresias, any scheduler that does not time-share GPUs such as [9, 25] will also be left with only these two options. These options are shown in Figures 1(a) and (b), respectively. In either case, note that the scheduler is *not inter-user fair*. In the former case, user C does not get its fair share of 2 GPUs while in the latter case, one user (user A as shown) is allocated four GPUs while the other user is allocated two GPUs. Thus, these options point out the *need for time-sharing of GPU resources for supporting inter-user fairness*.

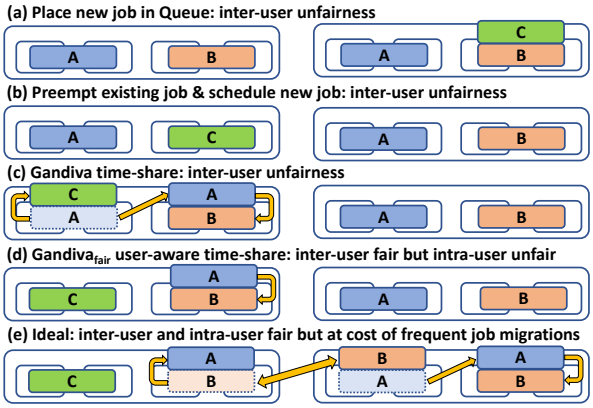

Figure 1. Two 4-GPU servers running two 2-GPU jobs of users A and B; A new 2-GPU job arrives for user C. Five scheduling options shown.

Gandiva [41] is a scheduler that uses time-sharing to schedule more jobs than GPUs. However, Gandiva is optimized for efficiency and not fairness. In this example, Gandiva would simply add user C's job to one of the nodes and time-slice all jobs equally on that node as shown in Figure 1(c). However, this approach also does not provide inter-user fairness as user C will receive only 4/3 GPUs when its fair-share is 2-GPUs.

As we will show in Section 3, *Gandiva$_{fair}$* allocates GPUs as shown in Figure 1(d). On one server, user C is allocated 2-GPUs for its job while user A and user B time-share equally on 2-GPUs. This allocation is *inter-user fair* since user C has been allocated 2-GPUs while user A and user B are allocated three GPUs in aggregate. However, this allocation is *not intra-user fair*, i.e., all jobs of a given user do not get equal resources. For example, one of user A's job gets the equivalent of one GPU through time-share while another of user A's job gets two GPUs.

If achieving intra-user fairness is necessary, then Figure 1(e) shows one possible way to achieve it. To support intra-user fairness, jobs of user A and user B are migrated periodically between the two servers so that each of the jobs of user A and B get a time-average of 1.5 GPU resources. While such a solution is intra-user fair, it will result in significant number of job migrations in a large cluster. Furthermore, it is not clear that strict intra-user fairness is a necessary requirement for deep learning since jobs that are part of multi-job are released at different times. Thus, *intra-user fairness is not a goal* for *Gandiva$_{fair}$*. Instead, *Gandiva$_{fair}$ balances efficiency and fairness by opting for only guaranteeing inter-user fairness*, as shown in Figure 1(d).

While this simple example motivates the various design choices, providing inter-user fairness gets challenging as we consider the following three realistic needs.

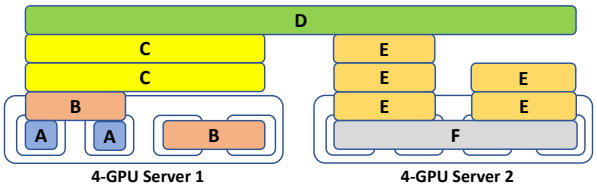

Figure 2. Gang scheduling with a mix of 1/2/4/8-GPU jobs from six users (A–F) on two 4-GPU servers. Job sizes depict their GPU requirements.

Inter-server gang scheduling. The above scenario assumes that all user jobs require the same number of GPUs. In reality, user jobs can require anywhere from one GPU to 128 GPUs or more. Satisfying inter-user fairness with a mix of job sizes with gang-scheduling is challenging for several reasons.

Consider two servers with a mix of job sizes shown in Figure 2. How do we ensure that all the jobs shown in the figure get their fair-share without sacrificing efficiency. In fact, Gandiva [41] avoids such scenarios by job-size based partitioning such that each server has jobs of only a given size. Since jobs of different sizes vary significantly [25], such partitioning will result in unbalanced load, causing unfairness and inefficiency. Thus, time-sharing a mix of job types on a single server is necessary for inter-user fairness.

In the above figure, first consider scheduling small jobs that span only Server 1. In a given time quantum, say we need to schedule user A and user C's jobs from a inter-user fairness perspective. However, since the sum of their requirements is 5 GPUs, this assignment is infeasible. One option to meet this fairness constraint is to schedule user-A's 1-GPU job in one time quantum and user C's 4-GPU job in the next time quantum but this is inefficient since in the first time quantum, three of the four GPUs will remain idle. We could allocate these three GPUs to 1-GPU and 2-GPU jobs of users A and B but, as we shall see in Section 3, such backfilling [13] without careful fairness consideration can result in unfairness. How then do we keep all GPUs utilized while ensuring inter-user fairness? We show how $Gandiva_{fair}$ balances efficiency and fairness to handle such scenarios using a gang-aware scheduling algorithm in Section 3.

Second, how do we ensure that the 8-GPU job of user D that spans multiple servers get scheduled? We would like each of the servers to make independent scheduling decisions for scalability reasons but we also need to ensure that they make coordinated choices so that user D's 8-GPU job gets scheduled at the same time quantum across servers 1 and 2. In Section 3, we present $Gandiva_{fair}$'s distributed, split scheduler that addresses this challenge.

Job	K80 (ms)	P40 (%)	P100 (%)	V100 (%)
VAE [26]	11.5	117	119	125
SuperResolution [37]	207.5	143	173	187
DCGAN [35]	183.4	434	431	642
GRU [12]	48.4	300	358	481
LSTM [6]	48.9	310	358	481
ResNet-50 [20]	134	317	334	514
ResNext-50 [42]	2005.7	370	412	633

Table 1. Performance speedup of various DLT models on different GPU generations compared against time per mini-batch in milliseconds on K80 GPU

Third, we have assumed so far that GPUs are homogeneous. However, a cluster will over time accrue a variety of GPU models since new GPU models are released every year while depreciation of older hardware typically takes three years or longer. Given heterogeneity, users prefer the newer GPUs as they are able to run their models faster. How then can a scheduler incentivize users to use older GPUs? We discuss this issue next.

2.2 GPU Heterogeneity

Table 1 lists the time per mini-batch in milliseconds for different deep learning training jobs for the older K80 GPU model and the speedup ratios for newer GPUs like P40, P100, and V100. The jobs span a variety of deep learning training tasks including Variational Auto-Encoders (VAE), Image Super Resolution, Deep Convolutional Generative Adversarial Networks (DCGAN), Gated Recurrent Units (GRU), Long Short-Term Memory (LSTM) and well-known convolution network models such as ResNet and ResNext. The speedup ratio shown is mini-batch time in K80 divided by mini-batch time in newer GPU, expressed as a percentage. These measurements were obtained using PyTorch v0.4 (more details in Section 5), and the mini-batch sizes were chosen as default values specified by the author of the model. The speedup for VAE and SuperResolution tasks in V100 are much lower than the other tasks because these are smaller models that end up significantly under-utilizing the powerful GPUs like V100.

For these measurements, the model parameters were trained using the typical 32-bit floating point (FP32) format. However, there is increasing realization in the deep learning community that many models with 16-bit parameters train as well as models with 32-bit parameters and newer GPUs have advanced support for FP16. Thus, the latest generation V100 GPU has specialized tensor cores designed specifically to support FP16 models. Compared to FP32, running ResNext-50 on a V100 with FP16 results in a further speedup of 10-15% while running the

same on a K80 results in a slowdown of 20%, thus, exacerbating the already large 6.33x gap between K80 and V100 to over 8x!

Heterogeneous GPU performance observations. There are four interesting observations that one can make from these measurements. First, it is clear that all models benefit from newer generation GPUs compared to older GPUs. Thus, users preferring newer GPU models is natural from their individual perspective of reducing the training time for their models. Second, we can see that the marginal utility of using a newer GPU varies significantly across models. For example, while running VAE on a V100 provides only a 25% gain in time relative to a K80, a ResNext-50 model sees a time gain of 533%. The third observation is that it is hard to predict a model's performance on different GPUs. For example, in the case of DCGAN, P40 and P100 performance is similar while V100 performs significantly better than P40 and P100. On the other hand, for SuperResolution, P40, P100 and V100 each provide incremental gains over each other. To handle this unpredictability, *Gandiva$_{fair}$* uses job migration across GPU generations along with profiling to measure the performance of each model with different GPU generations to make its trading decisions. Finally, we note that models that utilize FP16 format for its parameters gain even more speedup compared to a K80, attesting to the specialized support for this format in newer GPUs and greater marginal utility of newer GPUs for models that specifically use FP16 parameters. We discuss in Section 3.4, how *Gandiva$_{fair}$* leverages these observations to design an automated GPU trading strategy that maximizes efficiency while ensuring fairness.

3 Design

We assume each DLT job is assigned a number of tickets which represents its share of resources. Our goal is to provide proportional-share of resources based on job tickets like classic schedulers [39].

To achieve this goal, *Gandiva$_{fair}$* scheduler consists of three key components. First, *Gandiva$_{fair}$* uses a gang-aware, split stride scheduler to schedule DLT jobs of various sizes. Second, *Gandiva$_{fair}$* uses a ticket-adjusted height-based load balancer that uses migration to ensure that jobs across the cluster are balanced. Balanced load in conjunction with the split stride scheduler results in a fair and efficient service across a cluster of servers. Third, *Gandiva$_{fair}$* transparently handles GPU heterogeneity and implements an automated GPU trading strategy to improve efficiency across heterogeneous clusters while maintaining fairness. We discuss each of these components next and then present the full *Gandiva$_{fair}$* system.

Scheme	User	Job1 (%)	Job2 (%)
Lottery plus backfilling	A (1-GPU)	71.8	71.8
	B (2-GPU)	36.7	36.7
	C (4-GPU)	12.4	11.4
Gang-Aware Lottery (100 intervals)	A (1-GPU)	66	66.5
	B (2-GPU)	46	24
	C (4-GPU)	20	12
Gang-Aware Stride (6 intervals)	A (1-GPU)	66.7	66.7
	B (2-GPU)	33.3	33.3
	C (4-GPU)	16.7	16.7

Table 2. Scheduling in a 4-GPU server with 3 users, each with two 1/2/4-GPU jobs (Figure 2). Percentage of GPU-minutes job was scheduled.

3.1 Split Stride Gang-scheduling

Small jobs. Lottery [40] and Stride [39] are classic ticket-based scheduling algorithms. Let us first consider if we can utilize these for scheduling *small* jobs, i.e., jobs whose GPU needs can be served by a single server.

In Lottery scheduling, at every time-quantum the scheduler performs a lottery with the total number of tickets of all active jobs and schedules the job with the winning ticket. To support multi-GPU gang scheduling, we first normalize the tickets with their respective job sizes (divide tickets by number of GPUs) and draw winning tickets. Consider a 4-GPU server with a mix of small jobs. Let the first winning ticket be that of a 1-GPU job and we schedule it. Say the next winning ticket is for a 4-GPU job which does not fit. To maintain proportional-share, one option is to leave three GPUs idle for this time quantum and schedule the 4-GPU job in the next time quantum but this is inefficient. To improve efficiency, one option is to simply skip this 4-GPU job and generate another winning ticket. If the new ticket is for a 1 or a 2 GPU job, we allocate it, else we skip that job as well and we iterate until all GPUs are allocated or full allocation is infeasible. This approach is called backfilling [13] and can improve efficiency but at the cost of fairness.

Consider the 4-GPU Server 1 in Figure 2 that has six jobs from three users (ignore the 8-GPU job for now). Users A, B, and C, have two jobs each that require 1/2/4-GPUs, respectively. If all users have the same number of tickets, we should expect to see each of the 1/2/4-GPU jobs be scheduled for 66.7/33.3/16.7% of the available GPU-minutes, so that each of three user's two jobs in aggregate get 33.3% of the 4-GPU server. Table 2 shows the time share using lottery scheduling with backfilling. We can see that this scheme is unfair. This unfairness is because 4-GPU jobs are more likely to be skipped compared to 1/2-GPU jobs.

We can correct for the unfairness above as follows. Instead of drawing multiple lottery tickets, we conduct multiple lotteries to schedule jobs in a given time-quantum

(i.e., use sampling with replacement vs sampling without replacement). Each lottery winner gets scheduled in the time-quantum if it fits; if not, the job gets placed in the waiting queue. Jobs in the waiting queue always get scheduling priority at the beginning of every time quantum, ensuring that lottery winners get their fair-share. For the three user example (Table 2), while this scheme improves fairness over backfilling, due to lottery's probabilistic nature, it still results in unbalanced allocations over 100 time intervals. In this example, we find that the above gang-aware lottery scheduling achieves fair-share over intervals of only size 1000 or more (not shown). However, due to GPU context-switching overhead [41], each time-quantum in $Gandiva_{fair}$ is on the order of a minute. Since guaranteeing fairness over intervals of 1000 minutes is too long to be useful, *we conclude that gang-aware lottery scheduling is not suitable for* $Gandiva_{fair}$.

Let us now consider Stride [39]. A job's stride is inversely proportional to its tickets and denotes the interval between when it's scheduled. Strides are represented in virtual time units called passes. A new job's pass value is set to the minimum pass value of all jobs in the node. At each time quantum, the job with the minimum pass value is scheduled. The job's pass value is then updated by the job's stride.

Algorithm 1: Gang-Aware Stride

```
   Data: Set of running jobs.
   Result: Jobs scheduled this time-quantum.
1  begin
2      sortBy(jobs, λx : x.pass)
3      freeGPUs ← numGPUs
4      scheduled ← ∅
5      i ← 1
6      while freeGPUs > 0 and i ≤ |jobs| do
7          job ← jobs[i]
8          if job.size ≤ freeGPUs then
9              freeGPUs ← freeGPUs − job.size
10             scheduled ← scheduled ∪ {job}
11             job.pass ← job.pass + job.stride
12         i ← i + 1
13     return scheduled
```

We extend Stride to be gang-aware (Algorithm 1). The algorithm is called every time quantum and returns the jobs that are scheduled for the next time quantum out of the list of jobs available to be scheduled in the queue. Just as in classic Stride, jobs are chosen to be scheduled by the minimum pass value (line 2) but an additional check is performed to make sure they fit within the available resources (line 8). If the job fits, we schedule the job and update its pass value (line 9 – 11). If the job does not fit, we skip the job but note that the job retains it pass value; in the next time quantum, the job will have the minimum pass value and thus higher priority in being scheduled. For example, assume a 1-GPU job has minimum pass value and it gets scheduled on a 4-GPU server. Say a 4-GPU job has the next minimum pass value but we cannot schedule it. In gang-aware Stride, *we skip this job but retain its pass value*. We continue this process until all GPUs are allocated or there is no feasible allocation. Since we do not update the pass value of skipped jobs, they (e.g., the 4-GPU job) are guaranteed to have minimum pass values in the next time quantum and will get scheduled then. Thus, from a fairness perspective, the gang-aware Stride results in a service delay of at most 1-time quantum compared to the classic Stride algorithm. Finally, because of Stride's deterministic nature, the time interval required for providing the fairness guarantee in gang-aware Stride is significantly shorter than for the probabilistic gang-aware Lottery scheduling algorithm.

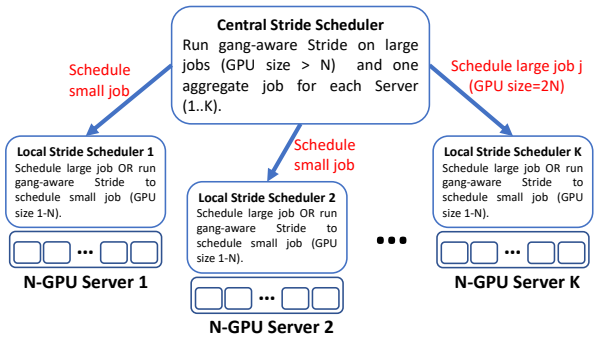

Figure 3. Split Stride Scheduler: Central scheduler for large jobs and local per-server scheduler for small jobs

Large jobs. Now consider large jobs that require GPUs across multiple servers (e.g., 8-GPU job in Figure 2). While we can run the gang-aware stride algorithm over the entire cluster, it will be inefficient as it will result in too many migrations (in each time quantum, jobs can be scheduled in any GPU in the cluster).

A key requirement for scheduling large jobs that span multiple servers is coordination for gang-awareness, i.e., all the GPUs across multiple servers for a given large job has to be allocated in the same time quantum. On the other hand, for scalability and efficiency reasons, we would like to run the gang-aware stride algorithm independently on each of the servers for small jobs.

To balance these conflicting goals, $Gandiva_{fair}$ uses a Split Stride scheduler, as depicted in Figure 3. A central scheduler maintains pass values of all large jobs and one aggregate job/pass value for each of the servers. The aggregate pass value for the aggregate job is inversely proportional to the cumulative number of tickets

of all small jobs in that server. When the central scheduler runs gang-aware stride, it selects the aggregate jobs and/or large jobs based on minimum pass value; in the former case, it simply instructs the respective server to run its own gang-aware Stride and schedule from its local pool of small jobs while in the latter case, it instructs the corresponding servers to run the large job. In this way, $Gandiva_{fair}$ achieves the coordination necessary for gang-aware scheduling of large jobs while simultaneously allowing independent servers to schedule small jobs.

The fairness provided by the Split Stride scheduler depends on how the job tickets are balanced across servers. If all servers are load balanced, ie., have equal aggregate tickets, the fairness provided by the split Stride scheduler can be shown to be identical to running a single cluster-level gang-aware Stride scheduler (but without the inefficiency of constant migrations). To see this, consider Figure 2. Assume all the small jobs in each of the servers have 200 tickets in total and the 8-GPU job has 100 tickets. In this case, the small jobs in aggregate will get 2 out of every 3 slots while the 8-GPU job will get 1 out of every 3. Thus, balancing the load among servers is crucial for fairness and efficiency. We discuss this next.

3.2 Load balancing

The key idea in $Gandiva_{fair}$ for ensuring fair share, is to distribute the *ticket load* across nodes as evenly as possible, and then use the Split Stride scheduler to schedule jobs in each node, proportional to the ticket load of each job.

Formally, let t_i, be the tickets of the i'^{th} user, and let $j_{i1}, j_{i2}, ..., j_{in_i}$ be the jobs of this user. Let the number of GPUs required by the job j_{ab} be r_{ab}. We can then define the $ticketsPerGPU_i$ for user i to be:

$$ticketsPerGPU_i = \frac{t_i}{\sum_{k=1}^{k=n_i} r_{ik}}, \tag{1}$$

where n_i is the number of jobs of user i. $ticketsPerGPU$ can be thought of as the number of tickets the user will be utilizing per GPU for its workload.

We can now define the ticket load per GPU on the i'^{th} node to be

$$ticketLoadPerGPU_l = \frac{\sum_{j_{ab} \in A_l} r_{ab} * ticketsPerGPU_a}{g_l}, \tag{2}$$

where g_l is the number of GPUs on node l, and A_l is the set of jobs scheduled on node l. $ticketLoadPerGPU$ can be thought of as the number of tickets each GPU on a particular node has to service. Now, if we achieve an equitable distribution of $ticketLoadPerGPU$ across all nodes, then it will ensure that all nodes service similar number of tickets. We can then achieve a fair distribution (in proportion of tickets) of GPU compute across jobs, by simply making sure that each node locally does a fair scheduling proportional to the local tickets.

For equitable distribution of load across all nodes, we follow the greedy heuristic of assigning a new job to the node with minimum value of $ticketLoadPerGPU$. Note that since the new job will change the $ticketsPerGPU$ (eq 1), we recalculate the $ticketLoadPerGPU$ for each node as per the updated $ticketsPerGPU$, and then choose the node with the least value.

For scheduling jobs within a node, we first calculate the tickets to be utilized for each job. For a job j_{ik}, this is simply given by

$$jobTickets_{ik} = ticketsPerGPU_i * r_{ik}, \tag{3}$$

where i is the job's user and r_{ik} is the number of GPUs required by the job. To ensure fair share locally, the scheduler then assigns time quantum to each job proportional to its $jobTickets$ using the Split gang-aware stride scheduling algorithm described in the previous section.

Note that, due to a burst of job departures, the load of the servers may get imbalanced. $Gandiva_{fair}$ fixes such imbalances using job migration as discussed in Section 3.5. Also, we have assumed so far, for simplicity, that each user has enough jobs to completely utilize their share of GPUs. However, it may be that a user submits much fewer jobs than needed to utilize his full share. In this case, we satisfy the needs of such users first, remove them from the list, recalculate the weights for the remaining users and iterate to determine each subsequent user's fair share. In this way, users with sufficient load get a *bonus* share from users who under-utilize their shares.

In order to implement this iteration in an optimized manner, we use the weight readjustment algorithm in [11] to calculate each users' *effective* tickets and then follow the above algorithm. Finally, care must be taken during placement as well as migration to ensure that jobs are "packed" in servers as far as possible to avoid non-work conserving scenarios, as discussed in the next section.

3.3 Scheduler Efficiency

A key aspect determining the efficiency of a scheduler is whether the scheduler is *work-conserving*, i.e., does the scheduler leave resources idle when there are active jobs that could be scheduled on them. The combined requirement of being fair while also handling variable-sized jobs, makes it challenging for the scheduler to be work-conserving in all scenarios.

For example, consider the case where only one job each of 1-GPU and 4-GPU sizes are active on a 4-GPU

server, each with equal number of tickets. To ensure fair-share, three GPUs will have to go idle in every alternate time interval.

Gandiva$_{fair}$ addresses this apparent conflict between fairness and efficiency by leveraging two domain-specific customizations. First, *Gandiva$_{fair}$* leverages the *mechanism* to migrate a job on-demand at a low cost (detailed in § 4), that allows jobs to be moved across servers. Second, the specific workload of DLT jobs in large clusters is particularly well-suited to a migration *policy* that performs intelligent packing of jobs to avoid such pathological scenarios.

For example, consider the job size distribution from the deep learning training job traces in Microsoft's Philly cluster [24]. Out of the over 110K jobs submitted in that trace, about 86.6% are 1-GPU jobs, 2.3% are 2-GPU jobs, 4.9% are 4-GPU jobs and 6.2% use more than 4-GPUs. The dominance of 1-GPU jobs implies that migration can be an effective mechanism to ensure that sufficient number of 1-GPU jobs are "packed" in servers to avoid the above non-work conserving scenario.

3.4 Handling GPU heterogeneity transparently

So far in the scheduler design, we have assumed that GPUs are homogeneous. We now discuss how *Gandiva$_{fair}$* handles GPU heterogeneity transparently to users. This has two components, first, allocation of jobs to GPUs of a particular model transparently and second, allowing two users to automatically trade their assigned GPUs to benefit each other.

Assigning jobs in a heterogeneous GPU cluster. Consider a cluster with a mix of V100s and K80s. Based on a given user's tickets, let us say their fair-share allocation is 4 V100 GPUs and 4 K80s.

If a user wants a particular GPU model, the user can specify that along with the job and scheduler simply 'pins' the job to that GPU and the rest of the section is moot for this job. However, we expect that most users will not want to pin GPUs, since, as we will see below, automated trading allows users to get higher throughput than pinning. In this case, when a job for the given user arrives, where do we allocate this job – on a V100 or a K80?

Gandiva$_{fair}$ assumes a strict priority order among various GPU models, with the newer GPUs like V100 having higher priority over older GPUs like K80. If a new job arrives, the scheduler automatically picks the newer GPU (V100), assigns the job there and profiles its performance. If the user is performing hyper-parameter tuning, many more similar jobs will be submitted by the user. Let us say the user submits 8 jobs. Initially, these jobs will all be placed on the V100 and time-shared. When jobs are time-shared and scheduler estimates that the job's memory requirements will fit on the older K80, the scheduler transparently migrates the job to K80 and profiles its performance there. Now, the scheduler can compare the job's performance on V100 and on K80, and decide how best to optimize performance for the user.

User	K80	V100	Speedup	Aggregate
Inter-user fair allocation				
A (VAE)	20	4	1.25	25
B (DCGAN)	20	4	5.0	40
C (ResNext)	20	4	6.25	45
Inter-user fair allocation with trading				
A (VAE)	40	0	1.25	40
B (DCGAN)	20	4	5.0	40
C (ResNext)	0	8	6.25	50

Table 3. Heterogeneous GPU allocations on a 12 V100, 60 K80 GPU cluster with inter-user fairness and trading. Aggregate performance shown in normalized K80 units.

For example, say the job is a VAE that gets 1.25x speedup on the V100 compared to the K80. In this case, since these jobs are being time-shared on the V100 currently, the scheduler transparently migrates the time-shared jobs to the K80 so that four jobs run fully on the V100 while four jobs run on the K80. In contrast, if the job is a DCGAN that gets a 5x speedup on the V100, *Gandiva$_{fair}$* will continue to time-share the eight jobs on the V100 so that these jobs proceed on average at 2.5x the rate. Thus, *Gandiva$_{fair}$ automatically chooses the GPU model that maximizes each job's performance*.

Once the jobs have been allocated their GPUs, *Gandiva$_{fair}$* supports GPU trading among users to further improve efficiency. To support automated GPU trading, *Gandiva$_{fair}$* utilizes a key observation from Section 2, viz., that DLT jobs have variable marginal utility from newer GPU generations.

Trading heterogeneous GPUs. One of the unique aspects of DLT jobs is the need to train jobs using a variety of hyper-parameters, in order to identify the best accuracy. Thus, the users submit what is called a multi-job [41], which is typically 10-100 copies of a DLT job, each with different hyper-parameters such as learning rate, weight decay, etc. Crucially, the job performance characteristics are identical across these large number of jobs from a given user. In this section, we show how trading can be used in such scenarios to increase application throughput of both users involved in a trade.

Consider a cluster with 60 K80s and 12 V100s, and three active users, A, B, and C, running hyper-parameter tuning over their VAE, ResNet and ResNext models, respectively. Assume that each of the users have submitted tens of jobs to the cluster as part of their hyper-parameter tuning experiments so that the cluster is fully utilized. If

the scheduler simply provided fair-share, each user will be allocated 20 K80s and 4 V100s. In terms of performance, user A will see a speedup of 25% when running in V100s, for an aggregate performance of 20 + 4*1.25 = 25 normalized K80 GPUs. Similarly, user B using ResNet will see a speedup of 5x from V100 for an aggregate performance of 20 + 4*5 = 40 normalized K80 GPUs. Finally, user C using ResNext will see a speedup of 6.25x from V100 for an aggregate performance of 20 + 4*6.25 = 45 normalized K80 GPUs. This is shown in Table 3.

Since users see varying marginal utility of using a V100 as compared to a K80, there is scope for improving overall efficiency using trading. Consider user A who benefits the least (125% faster) from a V100 compared to user C who benefits the most (625% faster). A V100 allocated to user C is 5 times more efficient (in terms of rate of job progress) than a V100 allocated to user A. Thus, it makes sense to trade K80s of user C to user A in exchange for V100s to maximize efficiency. *The key question is, at what price do we do this trade?*

Automatic trade pricing. One simple solution is to trade 1.25 K80s from user C for 1 V100 from user A. In this case, all the efficiency gains (trade surplus of 5x per V100) accrue to user C. However, this solution is susceptible to gaming by shrewd users. Consider user B who modifies his job to check for GPU architecture the job executes on and simply slows down its performance when running on K80 so that his model's overall speedup is 6.5X on V100 compared to K80. User B would then win the trade and recoup sufficient gains using the V100 to compensate for the slowdown in K80s. Thus, the trade price must be carefully chosen to avoid such gaming. Another solution is to trade 6.25 K80s from user C for 1 V100 from user A. In this case, the trade surplus goes entirely to user A and user C is not incentivized to trade.

$Gandiva_{fair}$ addresses this pricing dilemma by borrowing from the second-price auction mechanisms that have been extensively studied [8]. Specifically, $Gandiva_{fair}$ executes the trade at the second highest price, i.e., user C trades 5 K80s for each 1 V100 from user A, where the 5x is determined by second price (the speedup of user B; if there is no such user, we simply split the surplus equally). Second-price auction has nice properties such as incentive-compatibility, which implies that every user can achieve their best outcome by bidding their true preferences [32]. Under this scenario, artificially speeding up or slowing down their jobs will not help users. Further, both users of the trade benefit from some of the surplus efficiency gains.

Vickrey auctions have some weaknesses as well. In particular, Vickrey auctions are not collusion proof. If all the bidders in an auction reveal their prices to each other, they can lower their valuations. In our setting, collusion is not a major concern as we are using the auction to mainly distribute the performance gains from the trade; each user in the trade is still guaranteed to get their fair share.

Table 3 shows the allocation at the end of four such trades. User A has 40 K80s for an aggregate performance of 40 (compared to 25 earlier) while user C has 8 V100s for an aggregate performance of 50 (compared to 45 earlier). Thus, both users achieve better aggregate performance from the trade and overall efficiency of the cluster is maximized while ensuring inter-user fairness.

As described in the next section, $Gandiva_{fair}$ uses profiling to continuously maintain job performance statistics such as mini-batch progress rate. For jobs that may change their performance behavior in the middle of training such as in [43], $Gandiva_{fair}$ can detect such a change and undo the trades if necessary. Finally, while we currently only consider trading for hyper-parameters jobs where the jobs of a given user are homogeneous, as part of future work, we are interested in exploring if trading can be made practical even for heterogeneous jobs.

3.5 $Gandiva_{fair}$

We now present the complete design of $Gandiva_{fair}$. The time quantum is set to one/two minutes to ensure that the overhead of GPU context switching is under 1%.

Algorithm 2: Allocating a job.

Data: Set of $users$, $gpus$, $servers$, and $jobs$.
Result: The job, it's $tickets$, and $allocation$.

1 **begin**
2 $user \leftarrow minBy(users, \lambda u : u.tickets)$
3 $job \leftarrow maxBy(jobs[user], \lambda j : j.priority)$
4 **if** $job.perf \neq \emptyset$ **then**
5 $gpu \leftarrow minBy(gpus, \lambda g : \frac{job.perf[g]}{user.tickets[g]})$
6 **else**
7 $gpu \leftarrow maxBy(gpus, \lambda g : g.rank)$
8 $tickets \leftarrow user.tickets[gpu] * job.size$
9 $k \leftarrow \lceil \frac{job.size}{numGPUsPerServer} \rceil$
10 $sortBy(servers[gpu], \lambda s : s.load)$
11 $allocation \leftarrow servers[gpu][1..k]$
12 **return** $job, allocation, tickets$

Allocation. As discussed in Section 3.2, the scheduler maintains updated values for the three key variables depicted in equations 1–3 to make its scheduling decisions. Algorithm 2 shows how a job is first allocated to a node. The scheduler maintains a job queue for *each user*. The scheduler first finds the user using the least resources so far based on $ticketsPerGPU$ (line 2) and picks one job from that user based on the job's priority/arrival time for

scheduling (high priority and jobs which were submitted earliest are preferred). If the job has been seen before (say, part of a multi-job) and $Gandiva_{fair}$ has the job's profile information available for different GPU models, then $Gandiva_{fair}$ picks the fastest GPU model (line 5, 6) as discussed in Section 3.4. Note that $job.perf[g]$ refers to the mini-batch duration of the job on GPU g. Otherwise, it simply picks the latest GPU model in line 8 (e.g., V100). Once the GPU model is chosen, the actual node to schedule the job is based on the nodes with the lowest $ticketLoadPerGPU$ (line 11, 12). The scheduler computes $jobTickets$, the number of tickets for the job (line 9) and then schedules the job on the chosen nodes. The split gang-aware stride scheduler uses the $jobTickets$ to then schedule the job at each time quantum and ensure its fair-share.

Profiling. $Gandiva_{fair}$ uses job profiling to determine the speed-up of each job on various GPU models in the system. Job profiling statistics are collected opportunistically as the jobs are scheduled and thus incur no additional overhead. For example, when a user submits their first job, it's scheduled on the fastest available GPU (e.g., V100). For every time quantum, the job is profiled to determine its average time taken per mini-batch and this data is collected by the scheduler. As the user submits more jobs, the jobs are scheduled on the V100s until the user exhausts their V100 allocation (based on their tickets). The next job the user submits is scheduled on the second fastest GPU (e.g., P100). The job is then profiled on the new GPU. If the user is performing hyper-parameter tuning, the jobs will be similar and, thus, the scheduler can determine the speedup of the job on a V100 over a P100. Thus, as jobs arrive and are scheduled on different GPU models, the profiled value gets updated to the average of most recent statistics to reflect the current speedups. By maintaining average of recent statistics, $Gandiva_{fair}$ can detect and adapt to jobs that may change their performance behavior in the middle of training such as in [43].

Migration. Job migration options are evaluated every time quantum. In a homogeneous cluster, migration is used only for load balancing. Jobs from the highest loaded server are migrated to the least loaded server if the difference in their $ticketLoadPerGPU$ is above a threshold. In a heterogeneous cluster, migration is used for improving job performance and efficiency through trading. The scheduler looks at the average progress rate and $tokensPerGpu$ of each user's jobs on each GPU model and calculates effective performance (mini-batch rate divided by $tokensPerGpu$). Based on the profiled information, if the user will benefit from having their jobs migrated from one GPU model to another and if the resulting improvement is greater than a threshold, then the scheduler adds this user to the candidate migration list. Among all such (user, source GPU model, destination GPU model) tuples, the scheduler picks the one that will gain the most and migrates one job to the node(s) with the lowest $ticketLoadPerGPU$ of the destination GPU model.

Trading. GPU trading options are also evaluated every time quantum. For each (fast GPU model, slow GPU model) tuple (e.g. V100, K80), the scheduler finds a "seller" (the user with the fastest speedup on fast GPU relative to slow GPU) and a "buyer" (the user with the least speedup). The trade price is obtained by using the second highest bid, i.e., if the user with second fastest speedup has speedup r, then r slow GPUs are traded for one fast GPU between the buyer and seller, and then jobs for the seller and buyer are migrated to the traded GPUs. Each trade results in increased efficiency corresponding to the difference between the speedups of the buyer and the seller while the efficiency gains are distributed to both the buyer and seller based on the second price (Section 3.4). Further, during both job arrivals and departures, a trading check is performed to see if some of the trades must be undone (e.g., a new user has arrived and the traded GPUs are needed for fair allocation).

Scalability. The auctioning run time is $O(numUsers)$ and the allocation run time is $O(numUsers + numJobs)$. Thus, scaling with the number of users/jobs is not computationally challenging in $Gandiva_{fair}$.

4 Implementation

$Gandiva_{fair}$ uses Kubernetes [10] as a cluster manager with a custom scheduler that allocates jobs to nodes. Jobs are submitted as Docker [31] containers. We have implemented the scheduler in around 4k lines of Scala [7] code, using the Akka Actors library [1] for concurrency and gPRC [5] for performing remote procedure calls. There are four main modules: *manager*, *scheduler*, *executor*, and *client*.

Manager exposes a REST API and a gRPC endpoint for the clients to connect to the scheduler. Scheduler makes decisions like placement, migration, ticket allocation, management of bonus tokens, trading, *etc.*. There is one global executor for performing gang scheduling of multi-server jobs and one local executor for each server in the cluster and together they are responsible for running the jobs on servers in proportion to the tickets allocated by the scheduler. Finally, the client, which runs inside the container alongside the job, also exposes a gRPC endpoint, and is responsible for receiving commands from the executor to perform operations like suspend/resume, checkpoint/migrate, report job metadata, and report the status of the running jobs.

A key mechanism utilized by *Gandiva*$_{fair}$ is the ability to migrate jobs between nodes. In order to migrate jobs, we need to be able to checkpoint jobs on-demand and then resume these jobs on a different node. Some DLT jobs are written with checkpoint capability so that they can resume from a previous checkpoint if it exists. A simple search of GitHub repositories at the time of this writing showed that only about 20% of PyTorch jobs had such checkpoint functionality implemented. Moreover, even among those DLT jobs that use checkpoint, they typically only checkpoint every epoch. An epoch can last several hours or more. While such checkpoints are useful to guard against occasional server failures, *Gandiva*$_{fair}$ requires much more fine-grained checkpointing for fairness and efficiency. Thus, we implement an automatic, on-demand checkpoint mechanism in *Gandiva*$_{fair}$.

To support job migration, we have modified PyTorch and Tensorflow frameworks. Our implementation can handle unmodified user code, and requires only surgical changes to both the frameworks. Although generic process migration tools such as CRIU [3] exist, they cannot handle processes with GPU state. In our implementation, we fork a proxy process from the main process. We intercept all CUDA calls made by the process, and direct it via our proxy. This way the main process' address space remains CPU only, and can be easily checkpointed via CRIU. The proxy process is responsible for 1) translating all CUDA handles such as stream, context, etc. 2) keeping a log of all state changing CUDA calls, so that they can be replayed upon a restore, and 3) memory management of GPU memory. The memory manager `mmaps` the virtual address space to the physical GPU address space in a consistent manner across migration, so that pointers to GPU memory remain completely transparent for the parent process. Upon checkpoint, the proxy's memory manager copies the GPU state to the parent process' CPU memory and dies. The parent process can then be simply CRIU'ed. Upon restore the proxy process replays the log of state changing CUDA calls and copies the GPU memory back. All communication between proxy and the parent process is handled via shared memory with negligible overhead. The proxy implementation remains unchanged between PyTorch and Tensorflow, and requires only minimal modifications to the actual frameworks.

Our overhead for suspend-resume is similar to [41], i.e., about 100-250ms depending on the size of the model. However, compared to [41], we optimize the migration performance overhead by implementing a three-phase context switch called the suspend-preload-resume. When the framework is notified to suspend, it completes it within about 100ms by copying the minimal data in the GPU (proxy process) at the end of a mini-batch of training to the CPU memory (parent process), thus, allowing the scheduler to run another job on the GPU. If the job needs to be migrated across servers, then the scheduler performs a CRIU checkpoint on the job container and restores it on the target server. The framework then waits for a preload notification. When it receives the preload, it sets up the state on the new GPU(s) by replaying the log of all stateful operations but does not resume. Thus, preload hides the 5s latency for initialization of the GPU context. Finally, when the framework is notified to resume, it copies the data back to GPU memory, which takes about 100ms, and quickly resumes the GPU computation.

Thus, migration mostly occurs in the background while other jobs utilize the GPU.

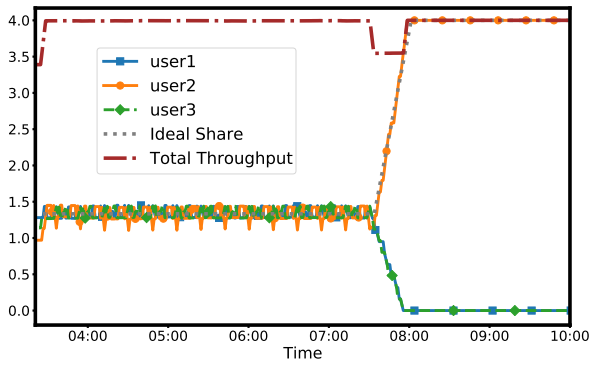

Figure 4. User Total GPU Throughput on 4 V100s.

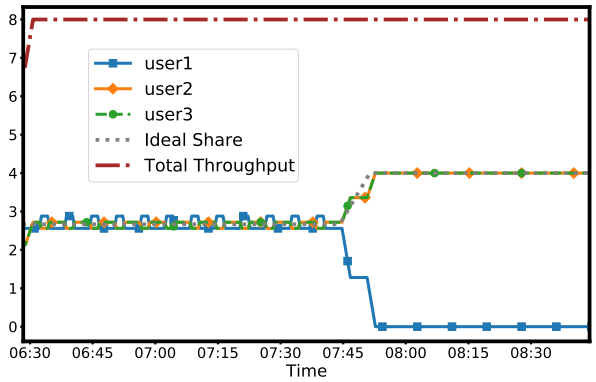

Figure 5. User Total GPU Throughput on 8 P100s.

5 Evaluation

We present our evaluation of *Gandiva*$_{fair}$. In aggregate, our cluster has 50 servers with 200 GPUs in total, comprising 24 V100, 48 P100 and 128 K80 GPUs. Each server has four GPUs of a single model, i.e., either K80, P100 or V100. The servers are 12-core Intel Xeon E5-2690@2.60GHz with either 224GB RAM (K80s) or 448GB RAM (P100, V100) and with 40Gbps ethernet (no RDMA), running Ubuntu 16.04 (Linux version 4.15).

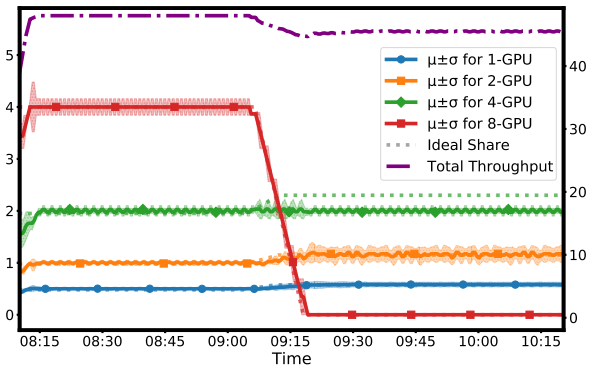

Figure 6. User Total GPU Throughput on 48 P100s (70 users/4 types).

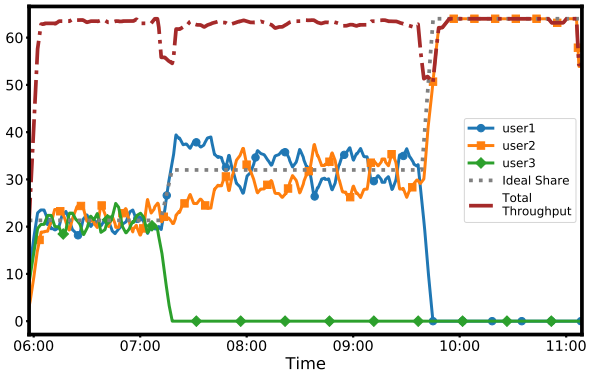

Figure 7. User Total GPU Throughput on 64 K80s (3 users/groups).

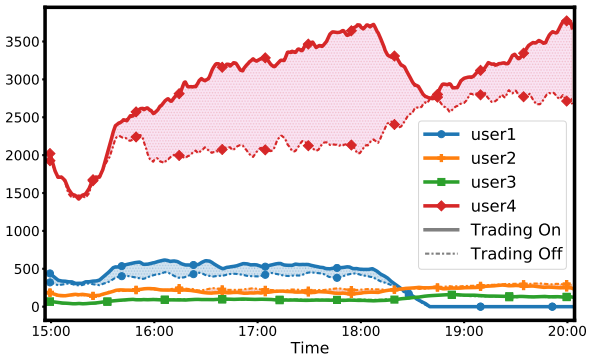

Figure 8. Mini-batch Rate with and without Trading.

Time	Bought	Sold	Rate
15:09	V100/user1	K80/user4	1.71
15:11	V100/user1	K80/user3	1.94
15:29	P100/user2	K80/user4	3.42
15:31	P100/user2	K80/user4	3.43
18:06	P100/user1	K80/user3	-1.49
18:06	V100/user1	K80/user2	-1.93
18:06	V100/user1	K80/user3	-1.94
18:07	V100/user2	K80/user4	1.50
18:23	V100/user3	P100/user2	1.00
18:25	V100/user3	P100/user2	1.00

Figure 9. Sample of trades executed by the scheduler.

User data is stored in Azure blobs and mounted in the container using blobfuse [2]. Additionally, we use Docker version 18.06.2-CE, NVIDIA Driver version 390.116, and CUDA version 9.1.

We evaluate *Gandiva*$_{fair}$ using a variety of deep learning training jobs written in PyTorch 0.4, TensorFlow 1.13, and Horovod 0.18.1. For jobs that use more than one GPU, we use data parallelism with synchronous updates as this is common practice. For large jobs, we use Horovod [36] that performs efficient all-reduce directly between the GPUs. The batch size and hyper-parameters used are model defaults.

In the following sections, we first present results to provide a detailed view of the various components of the *Gandiva*$_{fair}$ system. We then present cluster-wide evaluation results with all components of the system.

5.1 Single-server Gang-aware Stride scheduling

In this experiment, we show that our gang-aware stride scheduling is able to guarantee fair share to users with small jobs. We use a single server with 4 V100 GPUs and three users, each with 100 tickets. User1 submits four 1-GPU jobs running VAE [26] on MNIST [28] dataset, User2 submits four 2-GPU jobs running DCGAN [35] on CIFAR10 [27] dataset, and User3 submits four 4-GPU jobs running ResNet-50 [20] on CIFAR10 dataset. From Figure 4, we see that the gang-aware stride scheduler gives each user their ideal fair share of 1.33 GPUs on the 4 GPU server. After about four hours, jobs of User1 and User3 complete and we see that User2 starts to get the full throughput of the 4 GPU server.

5.2 Multi-server Split Stride scheduling

In this experiment, we show that our split stride scheduler is fair to both small jobs and large jobs that span multiple servers. We use 2 servers each with 4 P100 GPUs and three users, each with 100 tickets. User1 submits one 8-GPU job running ResNet-50 [20], User2 submits two 2-GPU jobs running ResNet50 [20], and User3 submits four 1-GPU jobs running VAE [26]. From Figure 5, we see that the split stride scheduler gives each user their ideal fair share of 2.66 GPUs on the 8 GPU cluster. After about one and a half hours, the job of User1 completes and jobs of User2 and User3 now get 4 GPUs each.

5.3 Homogeneous cluster

We now evaluate *Gandiva*$_{fair}$ on a large homogeneous GPU cluster with 48 P100s to illustrate load balancing working along with the Split stride scheduler. To design a realistic multi-user workload, we use the distribution of job sizes from Microsoft's Philly cluster job trace [24] and sample 70 jobs from this distribution randomly. Each user submits one job of the corresponding size.

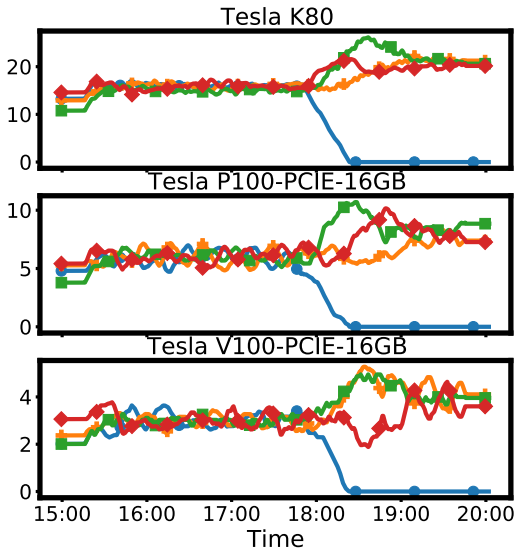

Figure 10. Total GPU Throughput on each GPU Model with Trading Disabled. Legend same as Figure 8.

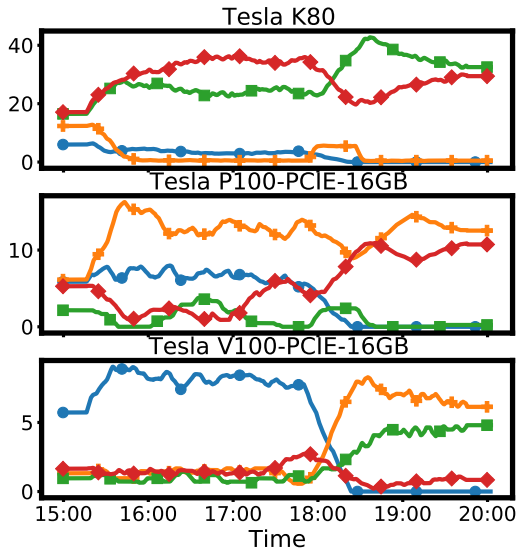

Figure 11. Total GPU Throughput on each GPU Model with Trading Enabled. Legend same as Figure 8.

In this experiment, we have 70 users in total, divided into four classes: 62 users with one 1-GPU job running VAE or SuperResolution [37] on BSDB300 [30] dataset, 3 users with one 2-GPU job running ResNet50, 3 users with one 4-GPU job running ResNet50, and 2 users with one large (8-GPU) job running ResNet50. Each user has tickets equal to the size of the job. Job arrival is random and around midway during the experiment, all the 8-GPU jobs complete. Figure 6 shows the average total GPU minutes for the four classes of users as well as their ideal fair share throughput (dotted-line) based on their tickets. One can see from the figure that each class of user gets throughput close to their fair-share and after the large jobs depart, the remaining users get their fair-share of the full cluster. The total throughput obtained by our scheduler, the scale for which is on the right, remains close to 48, with only momentary dips during job departures. However, migration policy activates, re-balancing the *ticketAdjustedLoad* and bringing the aggregate throughput (therefore efficiency) and fair-share back on track.

In the second experiment, we have three users but each with a large number of jobs. This configuration can represent a large-scale hyper-parameter tuning experiment since hyper-parameter exploration benefits from increased parallelism [29].

Alternatively, this can also represent the workload of three product groups, each with multiple users working on one particular model. In this experiment, each user/group has 100 tickets each. User1 submits 128 1-GPU jobs running SuperResolution while User2 submits 128 4-GPU jobs running ResNet50 and User3 submits 128 2-GPU jobs running DCGAN. From Figure 7, we see the three users/groups get equal share of the cluster-wide GPU minutes (about 21.3). After about 1 hour, jobs of User3 start finishing. At that point, we see that the cluster is now equally shared between User1 and User2 with each user/group getting on average 32 GPUs in aggregate. Finally, after 4 hours, the jobs of User1 completes and User3 has full access to the entire cluster throughput. As before, the aggregate throughput obtained by the scheduler throughout the duration of experiment remains close to 64. Thus, these experiments show that *Gandiva*$_{fair}$ provides fair-share and efficiency across a mix of large and small jobs as well as a mix of small and large number of users.

5.4 Heterogeneous Cluster: No Trading

We now evaluate *Gandiva*$_{fair}$'s performance on a heterogeneous 100-GPU cluster with 64 K80s, 24 P100s and 12 V100s. In order to highlight the dynamics of various components of *Gandiva*$_{fair}$, we have four users/groups using the cluster with large number of jobs submitted per user/group. Each user/group again has 100 tickets, and each user/group submits 150 jobs of various GPU sizes. In this experiment, User1 submits 150 1-GPU jobs running ResNet50, User2 submits 150 2-GPU jobs running ResNet50, User3 submits 150 4-GPU jobs running ResNet50, all on CIFAR10, and User4 submits 150 1-GPU jobs running SuperResolution on BSDS300 dataset.

Figure 8 shows the total job progress rate of each user using the total mini-batch rate of all jobs of each user and shows the comparison of the progress rate with trading.

Figure 10 shows three plots, each corresponding to one of the three GPU models. As we can see from the figure,

the jobs of the four users start by occupying an equal fair-share of aggregate throughput on each GPU model, *i.e.*, 3 V100s, 6 P100s and 16 K80s. After 4 hours, the jobs of User1 finish. At this time, we can see that the inter-user fair share readjusts and the remaining three users now get an equal share of about 4 V100s, 8 P100s and 21.3 K80s. Thus, $Gandiva_{fair}$ is able to provide inter-user fair-share in a heterogeneous setting.

5.5 Heterogeneous Cluster: Automated trading

On a similar heterogeneous 100-GPU cluster with 64 K80s, 24 P100s and 12 V100s and the same workload, we evaluate the efficacy of automated trading. As is evident from Figure 11, users start out by receiving roughly their fair share of GPU throughput. However, through the course of the experiment, $Gandiva_{fair}$ performs a multitude of GPU trades between the various users. This can be seen by comparison with Figure 10 as well as by the variation in the GPU throughput each user receives on each GPU model. Note that the scales of the y-axis of Figure 11 is larger than that of Figure 10. We can see the effect of trades where the user share diverges from the fair-share in terms of absolute throughput. However, the net job performance improves or remains the same, thus ensuring that the trades performed do not result in unfairness. Table 9 shows a subset of the trades performed during the experiment. A negative exchange rate implies that the trade was being undone because of changing system dynamics (job arrival/departure).

Figure 8 shows that the mini-batch progress rate of users with and without trade. Specifically, User1 and User4 gain about 30% higher efficiency due to automated trading. Also, observe how around the four hour mark, the progress rate with trading dips and then increases. This is because one of the users has departed and some trades are undone (resulting in a dip in efficiency back to pre-trade levels) and then new trades take place (resulting in increase in efficiency). This shows that trading is robust to user/job arrivals and departures.

6 Related Work

Big data cluster schedulers. There has been a lot of work on cluster schedulers in the context of scheduling big data jobs [16, 18, 22, 23, 44]. These jobs are modeled as data flow graphs and tasks from the graph are scheduled dynamically on a cluster, respecting the dependency relationships between tasks.

Fairness for big data jobs have also been explored extensively [4, 14, 15, 17, 21, 34]. By default, fairness in big data schedulers takes only CPU memory into account [4]. Dominant resource fairness [14] generalizes the notion of max-min fairness to multiple resources, specifically CPU and memory. Faircloud [34] considers the problem of sharing network while Tetris [17] schedules based on multiple resource requirements such as CPU, memory, disk and network.

Compared to big data jobs, deep learning jobs have very different characteristics. GPU resource is typically the primary bottleneck resource for DLT jobs. Further, unlike big data jobs, DLT jobs are unique in that they execute a repetitive set of processing tasks on the GPU [41]. $Gandiva_{fair}$ leverages the uniqueness of DLT jobs for providing inter-user fairness and transparent heterogeneity support.

DLT job schedulers. Early schedulers for DLT jobs borrowed directly from the big data schedulers [9, 25]. These schedulers treat DLT jobs as generic big data jobs that simply required an extra resource, the GPU. In these systems, GPUs are assigned exclusively for the lifetime of a DLT job. Thus, large job queueing times of hundreds of minutes [19] are common.

Optimus [33] is one of the first cluster schedulers that is customized for DLT jobs. Optimus builds a performance model for each DLT job on the fly and then schedules jobs to reduce overall job completion time. However, a key assumption in Optimus is that remaining time for a job is predictable. Tiresias [19] also optimizes job completion time for DLT jobs but without making any assumptions. Gandiva [41] introduces mechanisms like time-sharing, migration and profiling, made efficient by customizing to DLT jobs, to improve efficiency and reduce job queueing time. However, none of these schedulers address fairness issues; in fact, these schedulers don't even have a notion of users, and only operate at the granularity of DLT jobs. $Gandiva_{fair}$ also leverages the uniqueness of DLT jobs and is the first DLT job scheduler that supports inter-user fairness and GPU heterogeneity.

7 Conclusion

$Gandiva_{fair}$ is a cluster scheduler designed to meet three key requirements of deep learning training. First, it isolates one users' jobs from another, ensuring that each user gets their fair-share of cluster-wide GPU time. Second, it ensures that all users' jobs make progress. It achieves these through a mix of load balanced allocation at the cluster-level coupled with a gang-aware fair scheduler at the server level. Third, $Gandiva_{fair}$ manages GPU heterogeneity transparently to users. It achieves this through a combination of profiling of jobs across GPU models and a novel automated trading scheme that maximizes cluster efficiency while maintaining fairness.

Acknowledgments

We thank our shepherd Jacob Gorm Hansen and the anonymous reviewers for their valuable comments and suggestions.

References

[1] Akka Actors. https://akka.io.

[2] Blobfuse. https://docs.microsoft.com/bs-latn-ba/azure/storage/blobs/storage-how-to-mount-container-linux.

[3] Checkpoint/Restore in User Space. https://criu.org/Main_Page.

[4] Fair scheduler in hadoop. https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[5] gRPC, A high-performance, open-source universal RPC framework. https://grpc.io.

[6] Lstm training on wikitext-2 dataset. https://github.com/pytorch/examples/tree/master/word_language_model.

[7] The Scala Programming Language. https://www.scala-lang.org.

[8] AUSUBEL, L. M., MILGROM, P., ET AL. The lovely but lonely vickrey auction. *Combinatorial auctions 17* (2006), 22–26.

[9] BOAG, S., DUBE, P., HERTA, B., HUMMER, W., ISHAKIAN, V., JAYARAM, K., KALANTAR, M., MUTHUSAMY, V., NAGPURKAR, P., AND ROSENBERG, F. Scalable Multi-Framework Multi-Tenant Lifecycle Management of Deep Learning Training Jobs. In *Workshop on ML Systems, NIPS* (2017).

[10] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes. *ACM Queue 14* (2016), 70–93.

[11] CHANDRA, A., AND SHENOY, P. Hierarchical scheduling for symmetric multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 19*, 3 (2008), 418–431.

[12] CHO, K., VAN MERRIËNBOER, B., BAHDANAU, D., AND BENGIO, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259* (2014).

[13] FEITELSON, D. G., AND WEIL, A. M. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing* (1998), IEEE, pp. 542–546.

[14] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi* (2011), vol. 11, pp. 24–24.

[15] GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 365–378.

[16] GOG, I., SCHWARZKOPF, M., GLEAVE, A., WATSON, R. N. M., AND HAND, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 99–115.

[17] GRANDL, R., ANANTHANARAYANAN, G., KANDULA, S., RAO, S., AND AKELLA, A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 455–466.

[18] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 81–97.

[19] GU, J., CHOWDHURY, M., SHIN, K. G., ZHU, Y., JEON, M., QIAN, J., LIU, H., AND GUO, C. 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.

[20] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[21] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R. H., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI* (2011), vol. 11, pp. 22–22.

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72.

[23] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 261–276.

[24] JEON, M., VENKATARAMAN, S., PHANISHAYEE, A., QIAN, J., XIAO, W., AND YANG, F. 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.

[25] JEON, M., VENKATARAMAN, S., QIAN, J., PHANISHAYEE, A., XIAO, W., AND YANG, F. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *MSR-TR-2018-13* (2018).

[26] KINGMA, D. P., AND WELLING, M. Stochastic gradient vb and the variational auto-encoder. In *Second International Conference on Learning Representations, ICLR* (2014).

[27] KRIZHEVSKY, A., NAIR, V., AND HINTON, G. Cifar-10 (canadian institute for advanced research).

[28] LECUN, Y., AND CORTES, C. MNIST handwritten digit database.

[29] LI, L., JAMIESON, K., ROSTAMIZADEH, A., GONINA, E., HARDT, M., RECHT, B., AND TALWALKAR, A. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934* (2018).

[30] MARTIN, D., FOWLKES, C., TAL, D., AND MALIK, J. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision* (July 2001), vol. 2, pp. 416–423.

[31] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[32] NISAN, N., ROUGHGARDEN, T., TARDOS, E., AND VAZIRANI, V. V. *Algorithmic game theory*. Cambridge university press, 2007.

[33] PENG, Y., BAO, Y., CHEN, Y., WU, C., AND GUO, C. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems* (2018), ACM.

[34] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. Faircloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 187–198.

[35] RADFORD, A., METZ, L., AND CHINTALA, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[36] SERGEEV, A., AND DEL BALSO, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799* (2018).

[37] SHI, W., CABALLERO, J., HUSZÁR, F., TOTZ, J., AITKEN, A. P., BISHOP, R., RUECKERT, D., AND WANG, Z. Real-time

single image and video super-resolution using an efficient subpixel convolutional neural network. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 1874–1883.

[38] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (2013), ACM, p. 5.

[39] WALDSPURGER, C. A. Lottery and stride scheduling: Flexible proportional-share resource management, MIT.

[40] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation* (1994), pp. 1–es.

[41] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., ET AL. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 595–610.

[42] XIE, S., GIRSHICK, R., DOLLÁR, P., TU, Z., AND HE, K. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on* (2017), IEEE, pp. 5987–5995.

[43] YOU, Y., LI, J., REDDI, S., HSEU, J., KUMAR, S., BHOJANAPALLI, S., SONG, X., DEMMEL, J., KEUTZER, K., AND HSIEH, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations* (2020).

[44] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Eighth Symposium on Operating System Design and Implementation* (December 2008), USENIX.