GAI: A Centralized Tree-Based Scheduler for Machine Learning Workload in Large Shared Clusters

Ce Gao, Rui Ren(✉), and Hongming Cai

School of Software, Shanghai Jiao Tong University, Shanghai, China
{gaoce270863799,renrui,hmcai}@sjtu.edu.cn

Abstract. With widespread applications in image recognition, language translation, computer vision and other areas, deep learning (DL) have been proliferating over the past decade. Practitioners from different business groups in industries train DL models on a shared cloud computing infrastructure for these applications with different priorities. During the model training process, one of the key challenges is to minimize the life-cycle of high priority model training jobs. This paper analyzes the distributed training of machine learning (ML) models and identifies short board effect in the training process: GPU training requires higher network bandwidth compared to CPU training. The key insight motivates the design of GAI, a centralized scheduler for ML workload. It relies on two techniques: (1) tree-based structure. The structure stores the cluster information hierarchically to apply multi-layer scheduling. (2) well-extended priority algorithm. We consider priorities from multiple dimensions for model training jobs comprehensively to support resource degradation and preemption. The prototype of GAI is implemented on top of Kubernetes, Kubeflow, and TensorFlow. It is evaluated using a simulator and a real cloud-based cluster. Evaluations show 28% increase in scheduling throughput and 21% training convergence speedup on DL models.

Keywords: Resource management · Distributed machine learning Centralized scheduling · Resource utilization

1 Introduction

Over the past decade, we have witnessed the era of rapid advances in artificial intelligence, powered by the resurgence of ML, especially DL. DL has become a hot topic for both academia and industries like Alibaba, Facebook, and Google. These DL models exhibit a high degree of model complexity that raises new challenges and opportunities to cluster management.

ML frameworks like TensorFlow [1], MXNet [3], and Caffe [11] allow engineers to set up a one-off cluster to run distributed ML jobs with the support of parameter server architecture [17]. The architecture splits the job into two parts: parameter server and worker. A parameter server maintains a partition of the globally shared parameters. It collects the gradient and updates the parameters over training iterations. A worker server stores a portion of the training data locally to compute statistics such as gradients. The architecture has widely applied in DL model training.

Cluster management systems like Google Borg [2], Apache Mesos [10], Apache Yarn [23] and Daphne [25] now support multiple distributed computing systems which include TensorFlow and other ML frameworks in the same cluster. They greatly simplify the operation and maintenance work for the jobs submitted from different teams or users.

However, there is a problem in most cluster management systems which is limited rack-aware and priority support [26] that causes the difficulties to integrate real ML workload on the systems. None of the existing cluster management systems can efficiently handle ML workload in a large shared cluster. They are usually not able to offer the best hardware accelerators to the highest priority model training jobs. The main cause is lack of design and optimizations for ML workload from the scheduler side. Compared to traditional workloads, ML workload has some unique characteristics: First, distributed ML jobs are getting increasingly diverse both in terms of the size of input/output data and the scale of the models. Second, distributed ML jobs are usually network and computing intensive. Therefore hardware accelerators speed up the training progress significantly [16], and low latency network makes parameter updates efficiently. Last, the priority of ML jobs is more complex than traditional jobs. The distributed model training job usually contains a number of parameter servers and workers, and there are many dimensions, like distribution and the runtime of the job that will affect the priority.

To address these challenges, we propose Gatekeeper for AI (GAI), a centralized scheduler for ML workload on large shared clusters. Some contributions have been made in this paper:

– The system model for scheduling ML model training jobs on a given cluster is formalized in this paper. The formalization shows that the problem of scheduling ML jobs based on parameter server architecture is NP-complete.
– We present the ML workload characterization. Network and computing bottlenecks of ML jobs are verified experimentally. Different hardware devices (e.g. GPU, CPU) and communication modes (e.g. RPC, IPC) are used to train the model like Inception V3 [22], ResNet-50, ResNet-152 [9] and VGG-16 [21]. The experiments show clearly that when the ML training job uses CPUs, computing is the bottleneck; when using GPUs, the network is the bottleneck.
– Based on the key insight, GAI is presented to minimize the lifecycle of model training jobs and support priority for these jobs. GAI schedules distributed model training jobs based on parameter server architecture, on the data center. We offer best effort service and supports resource degradation and preemption due to two features: rack-aware tree scheduling; resource degradation and preemption.

– We implement the prototype of GAI on top of Kubernetes, Kubeflow, and TensorFlow. The evaluation shows that GAI improves the throughput by nearly 28% on a medium-sized cluster with the support of priority and achieves 21% training convergence speedup on DL models. We also demonstrate that the lifecycle of higher priority is shorter by average compared to those lower priority jobs. Then we can see that the overhead imported by GAI and light container-based virtualization is acceptable.

The rest of this paper is organized as follows: Sect. 2 describes the background, Sect. 3 motivates GAI with workload characterization, Sect. 4 presents the main methodologies adopted by this paper. We evaluate GAI in Sect. 5 and conclude this paper in Sect. 6.

2 Background

The design of GAI is related to distributed ML and cluster management systems. Therefore in this section, the parallel architectures of distributed ML jobs and cluster management systems are introduced as preliminaries. There is a discussion on the existing researches after the related work.

2.1 Parallel Architecture of Distributed ML

Distributed ML is an iterative-convergent program which is similar to single-process ML. Based on the property, the researchers proposed a parameter server framework for distributed ML [17]. Parameter server framework separates the system into parameter servers and workers. Parameter servers serve the globally shared parameters while workers maintain the training progress. The framework adopts either data parallelism or model parallelism [12].

Figure 1(a) is the architecture of model parallelism. In the model parallel architecture, the model is partitioned and assigned to different workers. Each worker maintains a part of the ML model and is responsible for updating it. Model parallelism is usually used to train models that require more memory

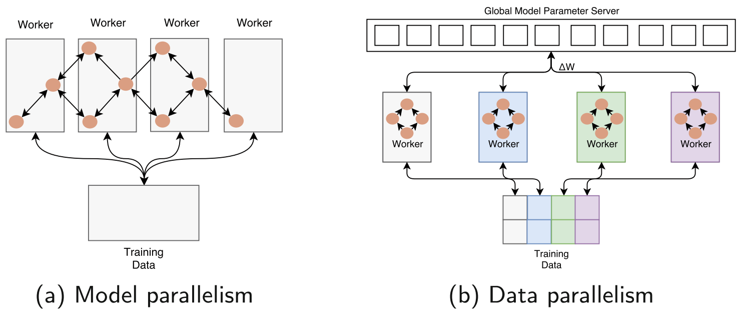


Fig. 1. Parallelism architecture

(e.g. image classification). Model parallel architecture introduces a certain amount of overhead, it relies on the good network connection.

Figure 1(b) is the data parallel architecture. Each worker in the architecture of data parallelism has a replica of the model and accepts a portion of training data. After one iteration, the workers push the gradients to the parameter servers and fetch parameters from the servers.

2.2 Cluster Scheduling System

Cluster scheduler plays an important role since hardware resources are allocated to specific jobs through the scheduler. Monolithic scheduler, such as Paragon [5], Quasar [6], Borg [24], Kubernetes [2] and Firmanent [7], uses a centralized single-process scheduler to schedule all kinds of jobs on the cluster. Monolithic scheduler is hard to expand with multiple workloads. Two-layer scheduler, such as Mesos [10] and Yarn [23], introduces application-specific scheduler into the monolithic architecture. The new layer guides the centralized scheduler to make suitable resource allocations for applications. To better concurrency, Shared-state scheduler, such as Omega [20], imports multiple schedulers based on the optimistic concurrency control strategy. It is assumed that the scheduling conflicts are rare, so shared-state scheduler performs high throughput. But there is a significant drop when the conflicts are frequent.

Distributed scheduler, such as Sparrow [19], is the architecture designed for batch jobs. In this architecture, there is no centralized scheduler to maintain the state of the cluster. The scheduler picks up some nodes and schedules the jobs in the subset of the cluster. Scheduling delay in distributed approach is relatively low but it is hard to support online business.

Hybrid scheduler, such as Hawk [4], Mercury [14], and Daphne [25], divides the jobs into long-running jobs and short jobs. It schedules long-running jobs using a centralized scheduler and assigns short jobs to a distributed scheduler. Hybrid scheduler adapts well for multiple workloads, but the complexity is high.

In the conclusion, now the existing researches on distributed ML mainly focus on the optimization from the ML framework side. It works well when the distributed training jobs are running on bare metal servers, while there is an increasing demand to run the workload in the cloud.

The existing cluster management systems usually treat batch jobs or long-running jobs as first-class objects. They are not designed for ML workload. Therefore, in this paper we analyze the workload and design GAI, to minimize the lifecycle of distributed ML jobs and import priority to model training.

3 Workload Characterization

In this section, we formalize the system model to introduce the problem that we hope to solve. Then short board effect of model training jobs on network and computing is presented, which shows the opportunities and challenges of scheduling ML jobs on clusters.

3.1 Problem Formalization

We consider that GAI schedules a set of jobs that contains a set of tasks on a homogeneous data center. To illustrate the process concretely, we take a model training job as an example. As shown in Fig. 2, a model training job has a number of parameter servers and workers. All the tasks (parameter servers and workers) of the job will be scheduled by the scheduler. And they will be placed on some servers. During the training progress, each worker communicates with all parameter servers via remote process call or inter process call in each iteration, and we call this the network cost. Workers execute real training logic using CPUs, GPUs or other hardware accelerators according to the model parameters from parameter servers, and this causes training cost.

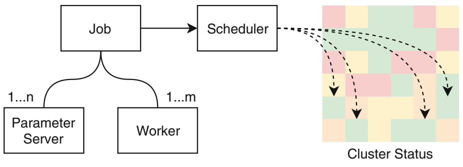


Fig. 2. Scheduling a model training job on a cluster

We assume that the resources in the data center are always strained, which is demonstrated in previous works [18]. Consider a set of model training jobs $J = \{j^1, j^2, \ldots, j^m\}$ running on a set of servers $S = \{s^1, s^2, \ldots, s^n\}$. We define a model training job j^i with j^i_{ps} parameter servers and j^i_{worker} workers, the time associated with the lifecycle of job T^i_j includes waiting time $T^i_{waiting}$, placement latency $T^i_{scheduling}$ and completion time $T^i_{completion}$. $T^i_{waiting}$ is spent when the job is queued to be scheduled. $T^i_{scheduling}$ is caused by the scheduler, which is dedicated to scheduling the job on the data center. $T^i_{completion}$ is the model training time and it can be defined by

$$T^i_{completion} = \sum_{z=1}^{j^i_{worker}} (C_{training}(w_z, s_z) + \sum_{k=1}^{j^i_{ps}} C_{network}(w_z, s_z, ps_k, s_k)) \cdot N \quad (1)$$

N is the number of iterations. s_z is the server that the worker w_z is performed. $C_{training}(w_z, s_z)$ is defined as the training cost of the worker w_z which is performed on server s_z in one iteration. It can be expressed as

$$C_{training}(w_z, s_z) = \begin{cases} C_{GPU}(w_z, s_z) & \text{If use GPU} \\ C_{CPU}(w_z, s_z) & \text{otherwise.} \end{cases} \quad (2)$$

We call C_{GPU} and C_{CPU} the cost using GPU and CPU. If the server has idle GPUs and the scheduler assign the GPU to the worker, the cost is the running time of training on the GPU. While C_{CPU} is the running time on CPU.

$C_{network}(w_z, s_z, ps_k, s_k)$ in Eq. 1 denotes the network cost for worker w_z on server s_z and parameter server ps_k on server s_k in one iteration. And it is defined by

$$C_{network}(w_z, s_z, ps_k, s_k) = \begin{cases} C_{IPC} & s_z = s_k \\ C_{RPC} & s_z \neq s_k \end{cases} \tag{3}$$

C_{IPC} is the cost using inter-process call (IPC). When the parameter server and the worker are performed on the same server, the method of communication between them is inter-process call. While when the parameter server in the server s_z while the worker in the server s_k, remote process call (RPC) is used to communicate. The cost is defined by C_{RPC}.

The scheduling algorithm for ML workload seeks mappings from tasks of the jobs to the servers with idle resources. The goal of the algorithm is to minimize $\sum_{i=1}^{m} T_j^i$, which has been demonstrated to be NP-complete [13].

3.2 Short Board Effect

As described in Sect. 3.1, computing and network communication are the major cost of a model training job, thus we present a study of the short board effect of data parallel ML training jobs on computing and network. The study demonstrates two main points through well-designed experiments:

- Short board effect is significant at cluster scale.
- ML jobs with GPUs suffer from low throughput network, while jobs using CPUs does not require high bandwidth connection.

Our hardware and software environment for the experiments are shown in the Tables 1 and 2. In 10GB Ethernet networks, we use CPUs and GPUs to train different models (Inception V3 [22], ResNet-50, ResNet-152 [9] and VGG-16 [21]). In the first experiment, we use 1 CPU, 1 GPU, 2 GPUs to train the ML models respectively. As shown in Fig. 3(a), the experiment of GPU based ML jobs yields speedups of 20 times than CPU based jobs. ML jobs using 2 GPUs are 90%–95% faster than the jobs using 1 GPU. And data parallel ML jobs using a mix of GPUs and CPUs does not fully exploit the GPU's performance because of short board effect.

Figure 3(b) shows the result of the training speed of 32 batch-size Inception V3 model in different distributed architectures and different hardware resources.

Table 1. Hardware configurations

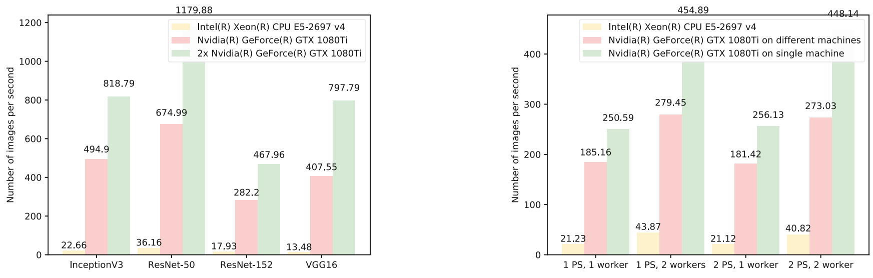
Hardware	Configuration
CPU	Intel Xeon CPU E5-2697 v4 @ 2.30 GHz
GPU	Nvidia GeForce GTX 1080Ti
Network card	Intel Corporation 82599 10 Gigabit
Switch	H3C S5820V2-52QF

Table 2. Software configurations

Software	Configuration
OS	CentOS Linux release 7.3.1611
ML framework	TensorFlow 1.4

To demonstrate the universality, We conduct in four architectures: (i) 1 parameter server, 1 worker, (i) 1 parameter server, 2 workers, (iii) 2 parameter servers, 1 worker, (iiii) 2 parameter servers, 2 workers. The result shows that the bottleneck of CPU based training jobs is computing, and the network does not affect the scalability. We place parameter servers and workers in different machines and get 35%–64% speed degradation compared with placing all parameter servers and workers in one machine.

Therefore we summarize the key insights: Network is not always the bottleneck for distributed ML jobs. It affects the training speed of GPU based ML jobs but CPU based jobs do not require high network throughput.



(a) Local training (b) Distributed training

Fig. 3. Training speed using different configurations

4 GAI: A Scheduler for ML Workload

In the previous section, we show the short board effect of model training jobs. In this section, based on the effect, we propose a tree-based scheduling model, then present a resource preemption and degradation algorithm for better utilization.

Based on the observations and the characteristics of ML workload in the previous section, we propose the goals of GAI:

- Minimize the lifecycle of model training jobs.
- Guarantee the priority of ML jobs. High priority jobs are allowed to preempt hardware accelerator resources to accelerate the training progress.

Figure 4 presents the overview of GAI. The input is a series of distributed model training jobs, and the output is the mappings from the tasks (parameter servers and workers) of the jobs to the servers. GAI relies on two main techniques:

Rack-Aware Tree Scheduling: GAI uses a centralized rack-aware tree scheduling method and maintains a resource tree in memory to place all tasks of the ML training jobs in one machine or in the machines belong to the same rack as far as possible.

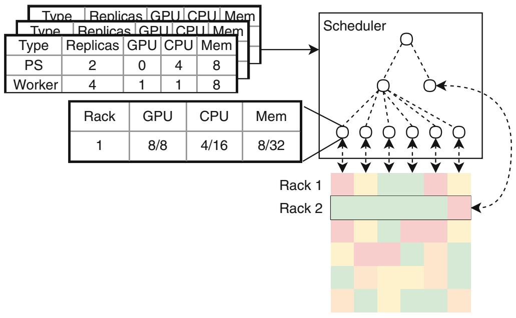


Fig. 4. Overview of GAI

Resource Degradation and Preemption: We present a resource degradation and preemption algorithm for data parallel ML jobs. There are different ML jobs in different priorities similar to traditional workloads. We use a vector to represent the priority and support degrading low priority jobs to release the hardware accelerators for high priority jobs.

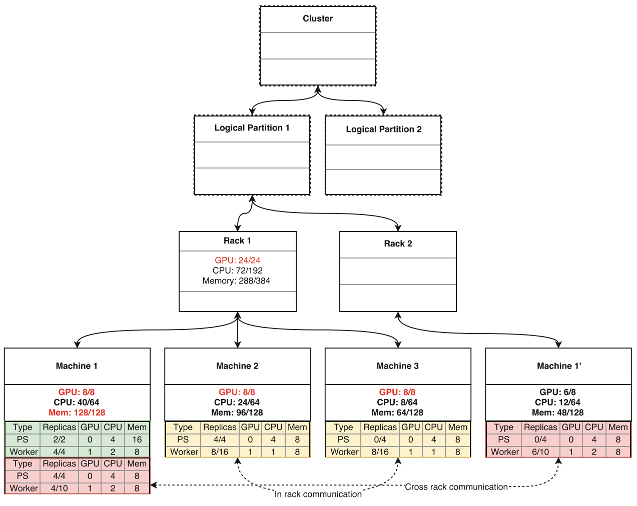
4.1 Rack-Aware Tree Scheduling

The previous section shows that GPU based training jobs are network-sensitive applications. Thus, GAI presents rack-aware tree-based scheduling and maintains two different scheduling paths. GAI chooses different paths according to the status of the cluster to keep high utilization.

In a commercial data center, the servers on the same rack share the same Ethernet switch, thus the servers are communicated with each other through a high bandwidth, low latency network. The feature is indifferent to network insensitive applications, such as web services, while it has a significant impact on the distributed training jobs which introduce heavy communication traffic between parameter servers and workers. GAI keeps a multi-level tree structure to organize all the servers in the cluster according to the network conditions between the servers.

ML training jobs usually require multiple hardware accelerators to accelerate the training. Thus, we gather the servers in the same rack into a small cluster, and the resources in the cluster can run at least one distributed ML job at the same time. Figure 5 represents the architecture of GAI scheduler. The resources in the cluster are abstracted into resource tree, where the leaf nodes in the tree represent servers and the second-level nodes represent the racks. The parent nodes in the tree collect and gather the runtime information (e.g. CPU, GPU and memory usage) of all its child nodes.

To preserve the extensibility, GAI supports logical partition in the resource tree. In some application scenarios, there are some servers without GPUs. These servers can be added to the same logical node to indicate that we can not schedule distributed training jobs to the servers. Most extensibility requirements can be supported indirectly through logical nodes.



Type	Replicas	GPU	CPU	Mem
PS	2/2	0	4	16
Worker	4/4	1	2	8

Type	Replicas	GPU	CPU	Mem
PS	4/4	0	4	8
Worker	4/10	1	2	8

Type	Replicas	GPU	CPU	Mem
PS	4/4	0	4	8
Worker	8/16	1	1	8

Type	Replicas	GPU	CPU	Mem
PS	0/4	0	4	8
Worker	8/16	1	1	8

Type	Replicas	GPU	CPU	Mem
PS	0/4	0	4	8
Worker	6/10	1	2	8

Fig. 5. Resource Tree in GAI

GAI performs multiple validations during scheduling on the resource tree to determine the sub-optimal placement:

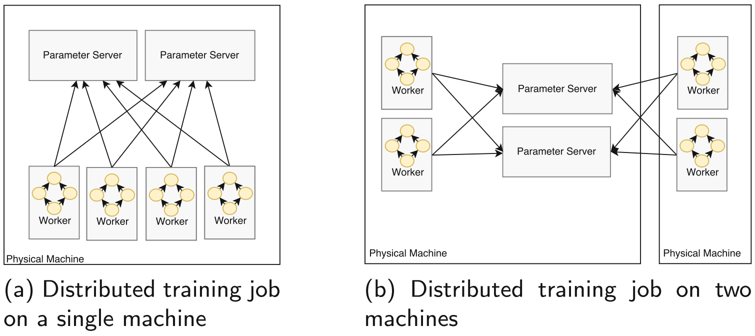
– First, GAI checks if the machines in the rack satisfy the resource requirements of the ML jobs. It is executed in rack-level nodes to determine if the sum of the free resources of all the machines on the rack can run the new job.
– After the first step, GAI validates the resource slots of each server to avoid resource stranded problem.

The placement algorithm is executed twice. The first pass is to schedule GPU resources. When there is no idle GPU, we run the resource degradation and preemption algorithm based on priority described in Sect. 4.2. If the cluster still does not have GPUs for the job, the requirement is relaxed and the algorithm is run for CPU again.

GAI implements a short scheduling path when the utilization of the cluster is low. Most of the servers has sufficient resources, then the default scheduling algorithm in GAI takes relatively long time to schedule a training job, so GAI imports randomized method to speed up the scheduling process. GAI randomly selects some secondary nodes and decides which rack to assign the new ML jobs based on the resource usage. GAI uses a random approach as Sparrow [19] does in a centralized manner. The randomized scheduling method reduces the size of potential candidate set and the scheduling delay as well when the cluster is idle.

4.2 Resource Degradation and Preemption

In a commercial cluster, ML jobs have different types and belong to different business groups, thus have different priorities. We design GAI's priority strategy based on priority vector. It is used to perform resource degradation or preemption. We summarize some factors that affect the priority of ML jobs:



(a) Distributed training job on a single machine

(b) Distributed training job on two machines

Fig. 6. Distribution of ML jobs

The Distribution of ML Jobs. The distribution of ML jobs on the cluster is very complicated. For example, we create a ML job with two parameter servers and four workers. In the best case, all replicas are scheduled to one server which has free resources, as shown in Fig. 6(a), to avoid the short board effect.

Figure 6(b) shows a worse case: two workers are placed on another server, thus the communication between the parameter servers and these two workers is not as good as the other two workers. Such job is in relatively low priority since the training speed is lower than the situation in Fig. 6(a).

Therefore, we design a priority algorithm based on the distribution of ML jobs. The algorithm can be expressed as Algorithm 1. If all parameter servers and workers are placed in the same machine, the job is in highest priority on this dimension. We prefer to preempt or degrade the jobs whose internal communication is cross rack or cross server.

ML Job Runtime. The runtime of ML jobs affects the priority, since the cost of restarting or interrupting an ML job that has been running for a relatively long time.

To determine the distribution of the duration of ML jobs, we analyze the data trace of ML workload in Facebook [8], extract the description and summarize the workload characteristics as described in Table 3.

The training jobs of neural network models such as CNN and RNN are the longest-running jobs and take approximately tens of hours, while GBDT and SVM jobs take less time. We use a power-law-like heavy-tailed distribution to sample the duration of the jobs. In the long-tailed distribution, the vast majority of jobs are completed in a short time. Therefore, we use the logarithm to calculate the priority and ensure that the priority of the job is distributed within a reasonable range. And we truncate the priority if we encounter the situation that it exceeds the threshold (The highest priority for a single dimension is set to 5).

Algorithm 1. Priority algorithm for distribution dimension

```
$Priority \leftarrow 0$
$MaxPriority \leftarrow 0$
**for** $ps \in PSes$ **do**
    **for** $worker \in workers$ **do**
        $MaxPriority \leftarrow MaxPriority + HighPriority$
    **end for**
    **if** $isInOneMachine(ps, worker)$ **then**
        $Priority \leftarrow Priority + HighPriority$
    **else if** $isInOneRack(ps, worker)$ **then**
        $Priority \leftarrow Priority + MediumPriority$
    **else**
        $Priority \leftarrow Priority + LowPriority$
    **end if**
**end for**
**return** $Priority/MaxPriority$
```

Table 3. Characteristic of ML workload in Facebook

Model	Resource	Frequency	Duration	Inference relative capacity
SVM	CPU	Every hours	Few seconds	10x
GBDT	CPU	Daily	Few hours	1x
CNN	GPU	Weekly	Many hours	10x
RNN	GPU	Weekly	Many hours	1x

The Type of ML Jobs. We define the type according to multiple dimensions as described in Table 4.

The online model training jobs have the highest priority, so we set the priority of these jobs to 5. And there are two types of research model training jobs: normal training jobs and hyperparameter tuning jobs. Hyperparameter tuning jobs consume more resources and usually are not urgent jobs. The priority of this type is set to a lower value.

Table 4. Priority for types of model training jobs

Type	Training	Hyperparameter training
Production	5	N/A
Research	3	1

The Type of Dominant Resource. In general, hardware accelerators are more expensive, then the utilization of this kind of resource is more critical than other hardware resources. The primary goal of this dimension is to increase the utilization of hardware accelerator resources. It is not an ideal solution to degrade the jobs whose replicas are all running on CPUs since CPU is not the first-class resource for ML jobs. We calculate the priority based on the numbers of GPUs that used by workers. Equation 4 shows the calculation. In this equation, n represents the number of GPUs that workers of the job are using. Sigmoid function is used to determine the upper and lower bounds of the convergence of the function.

$$Priority(n) = \frac{5}{1 + e^{-n}} \tag{4}$$

Number of Preemptions. Starvation occurs when a higher priority job dominate a resource and a lower priority job is blocked from gaining access to the resource. As a result, the lower priority job cannot make progress. To avoid the problem, GAI adds a bias. GAI offers the jobs that have been preempted one or more times the highest priority in this dimension. And the jobs without any degradation are in the lowest priority.

We aggregate the priorities of different dimensions into a priority vector. GAI refers to the predicate-priority model in Kubernetes.

– Firstly, the predicate process is performed. In this process, we find out all the jobs that can be preempted according to the hardware resource requirement of the new job. GAI supports single-job preemption in this process because of the complexity.
– Secondly, GAI determines if the dimension of job type in the priority vector is strictly greater than the preempted job, to ensure that ML jobs in the production environment are not preempted by the ML jobs for research.
– Finally, GAI removes the jobs in the candidate set that are in higher priority than the newly submitted jobs. The distribution of the jobs is a property at runtime, thus GAI sets the dimension of the new job to the highest score by default. While the job gains the lowest score in the dimension of ML job runtime. GAI performs weight-based calculations on the four dimensions of priority, as shown in Eq. 5.

$$Priority_{total} = \frac{\sum_{i=1}^{4} W_i \cdot Priority_i}{5\sum_{i=1}^{4} W_i} \tag{5}$$

5 Evaluation

In this section, we compare GAI with default scheduler in Kubernetes. Evaluations show that GAI improves the scheduling throughput and speeds up the training of ML jobs.

5.1 Methodology

Implementation. We implement the prototype of GAI as a stand-alone scheduler for Kubernetes 1.8.5. GAI can work together with the default scheduler with the help of Kubernetes by design. In that case, GAI schedules ML jobs while

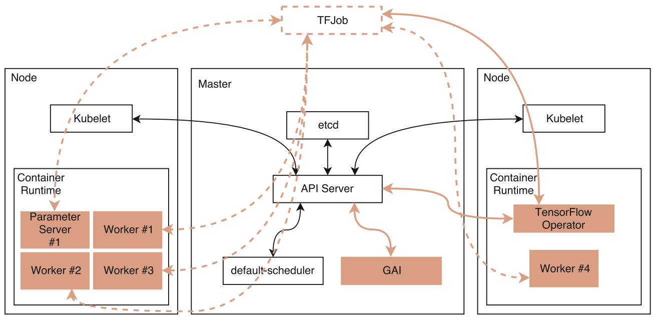


Fig. 7. Architecture of GAI

default scheduler deals with other jobs. We choose TensorFlow 1.4.0 as the framework for running ML jobs. Kubeflow 0.1 is applied to combine TensorFlow and Kubernetes.

Figure 7 shows the architecture of GAI. We build GAI on top of Kubernetes instead of revising the original code of Kubernetes. In the prototype, we register a custom resource definition $TFJob$ for distributed TensorFlow model training jobs in the cluster and run an operator to manage the lifecycle of TensorFlow training jobs on Kubernetes. TensorFlow operator from Kubeflow creates informers for $TFJob$, which is TensorFlow custom resource, pod and service which are Kubernetes internal resources. It watches the shared state of the cluster through Kubernetes API server and makes changes the attempting to move the current state towards the desired state. GAI is placed in the master node and it is responsible for scheduling TensorFlow jobs.

Workload. There is no public trace now for ML workload, hence we construct the workload trace mainly based on the description of the internal ML workload in Facebook [8]. In the real cluster, there are some jobs for research purpose which duration and number of tasks per job are shorter than the jobs for production purpose. Thus we also create a trace of ML jobs submitted by researchers. We use a power-law distribution similar to the production environment to generate the trace.

Simulator. We implement a simulator to simulate how GAI behaves in a large shared cluster. Different hardware leads to different training speeds, thus we assume that training jobs using GPUs are 20 times faster than jobs using CPUs according to historical records. The simulator reads trace data as input, run the real scheduling algorithm and assign jobs to virtual nodes. Scheduling and communication delays are set to random numbers which change within a relatively small range. We run the simulator on one 8-core Intel(R) Core(TM) i7-6700 CPU bare metal server. It can simulate the scheduling process in the cluster with 20000 virtual servers.

Real Cloud-Based Cluster. We establish a real cluster based on the cloud. The cluster has 5 8-core CPU servers with hyper-threading enabled and 5 8-core servers with 1 GPU ($10 \times 8 \times 2 = 160$ virtual CPU cores and 5 GPUs in total). Most of the experiments are run based on the simulator approach while we use the real cloud-based cluster to get the real load information of GAI.

5.2 Scheduler Throughput

We run Kubernetes default scheduler as the baseline implementation and GAI, to demonstrate the performance. We submit the workload described above and run the experiments in the cluster with 200, 500, 1000, 5000, and 10000 nodes iteratively. Virtual servers with different hardware configurations are created. 50% servers in the cluster have 1 GPU and 30% servers have 2 GPUs, while the other 20% only have CPUs. To avoid the potential problem that the cluster is full of use, the duration per job is set to 5 s. We disable the preemption and degradation functionality, because the feature is not expected in the benchmark. The corresponding logic about preemption in Kubernetes is also skipped.

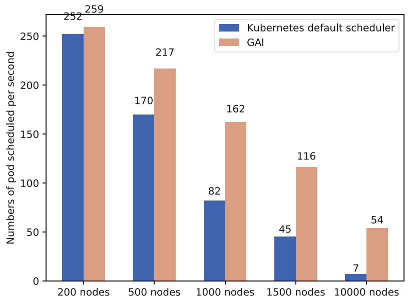


Fig. 8. Scheduler throughput

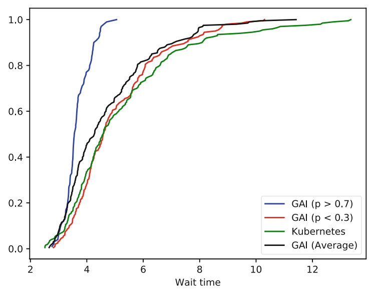
We implement the benchmark based on the scheduler performance test in Kubernetes and run it for 5 min. Then we calculate the average throughput for the scheduler. As shown in Fig. 8, the throughput of GAI is 27.6% higher than the baseline implementation at medium scale (500 servers), and behaves better at large scale.

GAI maintains a tree-based architecture. When the requests are sent from the control panel, GAI queues the requests in different queues for nodes. Thus the tree-based architecture has good scalability. Kubernetes native scheduler uses a single queue to manage all the resource requests, and it needs to run the predicate and priority processes for each server. The design allows Kubernetes to schedule the traditional workloads well but it also imports some overhead when the cluster size is growing.

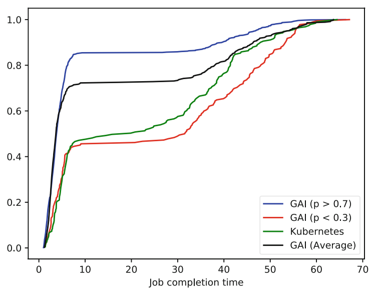
5.3 Job Waiting Time

Job waiting time is the time from the job is queued to be scheduled to the job is actually scheduled by the scheduler. In this section, we submit jobs with different priorities. We use the workload above while the training is not actually executed. In order to control the duration of the ML jobs precisely, the jobs only create the parameter servers and workers but it does not train the models. We set the active duration for jobs and kill the training jobs when it is time.

We group the jobs whose priority is larger than 0.7 as high-priority jobs. And jobs whose priority is lower than 0.3 are grouped as low-priority jobs. Because the priority is dynamic, we count all jobs which have come to the threshold at least once valid. We also run the workload in Kubernetes for comparison.



(a) Job waiting time



(b) Job completion time

Fig. 9. End-to-end latency

We run the experiment in the real cloud-based cluster. Figure 9(a) shows the result of the experiment. Jobs whose priority are greater than 0.7 achieves lower waiting time because of the support of resource degradation and preemption. And we can see that the waiting time of the jobs scheduled by Kubernetes default scheduler is slightly longer than GAI. The distribution of Kubernetes is a long-tailed distribution since the jobs are overstocked.

5.4 Job Completion Time

Job completion time is the training time spending on the model. We design an experiment using the real cluster to demonstrate that the high priority jobs are more likely to use GPUs to train. We run simple MNIST model training jobs using CNN and collect the completion time of jobs. The training job usually takes 30–50 s when training based on CPUs and takes nearly 2–4 s on GPUs.

We run this experiment in the real cluster and limit the number of iterations to make the model training process predictable. Figure 9(b) demonstrates that jobs whose priority are greater than 0.7 usually have GPUs to run, therefore the completion time is shorter and 80% jobs finish their training in 8.7 s. Low priority jobs spend more time to do the same mode training task. 80% of low priority jobs finish in 48 s. As shown in Fig. 5, the average training time of GAI is 14.6 s which achieves 21% speedup compared to Kubernetes default scheduler.

Table 5. Training convergence speedup in top k% ML jobs

Top k%	Speedup	Convergence (s)
25	26.1%	2.03
50	22.7%	2.56
100	21.4%	14.59

5.5 Comparison with Native Distributed TensorFlow

GAI runs the ML workload on container-based platforms, and it takes some overheads for the training. We evaluate the convergence speed of GAI on MNIST [15] training job, and GAI with native distributed TensorFlow when training a DNN for the MNIST dataset. We run the DNN with 1 parameter server, 5 workers and 2 parameter servers, 4 workers.

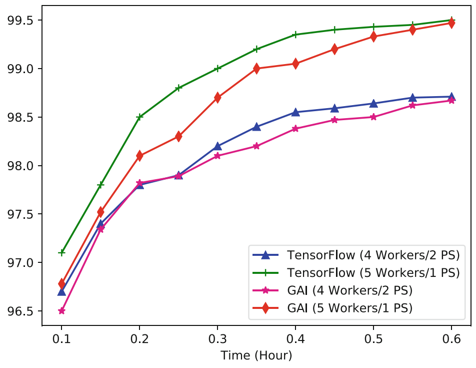


Fig. 10. Model convergence (MNIST)

Figure 10 shows the result. The convergence speed of the jobs scheduled by GAI and run on Kubernetes does not have significant differences compared to native distributed TensorFlow. The jobs running with 5 workers and 1 parameter server converge slower than native distributed TensorFlow since the job is network and computing intensive and the virtualization of containerization (e.g. Docker) uses cgroup and apparmor for isolation and security. These features import overhead for computing.

5.6 Discussion

In this section, we reconsider the design decisions of GAI and discuss the limitations.

We provide more insights on the performance and effect of GAI. GAI's high throughput capability benefits from the tree-based architecture. Experiments demonstrate that GAI provides best effort service for jobs. GAI improves the throughput by nearly 28% on a medium-sized cluster, and achieves 21% training convergence speedup on DL models compared to Kubernetes default scheduler. We also compare container-based solution with native distributed TensorFlow to illustrate the overhead imported by the prototype of GAI is low. Table 6 shows an overview of a selection of orchestration frameworks, their architecture, and features.

Table 6. Comparison with existing scheduler frameworks for ML workload

Scheduler	Architecture	Resource granularity	Rack awareness	Priority preemption	Priority degradation	Gang scheduling
GAI	Centralized	CPU, GPU, Memory	Yes	Yes	Yes	Yes
Kubernetes	Centralized	CPU, GPU, Memory	Partial	Partial	No	No
Mesos	Centralized (Two Layer)	CPU, GPU, Memory	Yes	No	No	Yes
Yarn	Centralized (Two Layer)	CPU, GPU, Memory	Yes	No	No	No
Sparrow	Distributed	CPU, Memory	No	No	No	No
Hawk	Hybrid	CPU, Memory	No	No	No	No

GAI relies on many parameters and thresholds in the scheduling process. Currently, we assign the values to these parameters and thresholds manually, and statically. We can use some ML algorithms about hyperparameter tuning, to choose the optimal values for these parameters. This requires a reasonable model and data set. In addition, GAI should support dynamic parameter adjustment. Under different loads, the weight of each dimension of the priority vector should be adjusted.

The prototype is a scheduler plugin in Kubernetes, and it can work with Kubernetes default scheduler to schedule multiple workloads via different schedulers. The feature is implemented from Kubernetes side, while Kubernetes has no mechanism to handle scheduling conflicts between different schedulers. Therefore we do not evaluate it. It should be supported after Kubernetes has a good support for the feature. Moreover, GAI currently uses container-based virtualization to isolate resources, which is the default option in Kubernetes. We are investigating using hypervisor-based containers for better isolation during model training.

6 Conclusion

The work presented in this paper consists in a centralized scheduler for ML workload named GAI to effectively share a single cluster among different DL applications. To this aim, we propose tree-based scheduling to establish the hierarchical structure of the cluster and the multi-dimensional priority algorithm which considers different aspects of model training jobs to degrade or preempt the resource for higher priority jobs. By hiding the short board effect, we have demonstrated the capability of our approach to support large shared clusters containing hundreds of thousands of servers. We implement the prototype of GAI on top of Kubeflow, Kubernetes, and TensorFlow. Moreover, we create a trace based on the real ML workload in Facebook and evaluate GAI using the trace. The result shows that the throughput of GAI is 27.6% higher than default scheduler in Kubernetes at medium scale when scheduling ML jobs and it achieves 21% training convergence speedup on DL models. Then there is an experiment to demonstrate that GAI imports fairly low overhead to improve isolation compared to native distributed TensorFlow.

The main directions for future work are twofold. The first one we are currently investigating is fine-grain control of hardware accelerator management. GAI currently requires the exclusive use of GPUs. As future work, GAI should import fine-grained scheduling and affinity control to make the most advantage of GPUs.

As long-term future work, We are investigating approaches and methods of improving scheduling and isolation of distributed model training jobs to make GAI production ready. We hope that GAI will inspire more ideas on scheduling for ML workload and ship off practical implementation.

Acknowledgements. This research is supported by the National Natural Science Foundation of China under Grant No. 61373030.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, vol. 16, pp. 265–283 (2016)
2. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016)
3. Chen, T., et al.: MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
4. Delgado, P., Dinu, F., Kermarrec, A.M., Zwaenepoel, W.: Hawk: hybrid datacenter scheduling. In: Proceedings of the 2015 USENIX Annual Technical Conference, No. EPFL-CONF-208856, pp. 499–510. USENIX Association (2015)
5. Delimitrou, C., Kozyrakis, C.: Paragon: QoS-aware scheduling for heterogeneous datacenters. ACM SIGPLAN Not. **48**, 77–88 (2013)
6. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and QoS-aware cluster management. ACM SIGPLAN Not. **49**, 127–144 (2014)
7. Gog, I., Schwarzkopf, M., Gleave, A., Watson, R.N., Hand, S.: Firmament: fast, centralized cluster scheduling at scale. In: 12th USENIX Symposium on Operating Systems Design and Implementation. USENIX (2016)
8. Hazelwood, K., et al.: Applied machine learning at Facebook: a datacenter infrastructure perspective. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 620–629. IEEE (2018)
9. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

10. Hindman, B., et al.: Mesos: a platform for fine-grained resource sharing in the data center. In: 8th USENIX Symposium on Networked Systems Design and Implementation, vol. 11, p. 22 (2011)
11. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
12. Jiang, J., Yu, L., Jiang, J., Liu, Y., Cui, B.: Angel: a new large-scale machine learning system. Natl. Sci. Rev. **5**, 216–236 (2017). https://doi.org/10.1093/nsr/nwx018
13. Jin, J., Luo, J., Song, A., Dong, F., Xiong, R.: Bar: an efficient data locality driven task scheduling algorithm for cloud computing. In: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 295–304. IEEE Computer Society (2011)
14. Karanasos, K., et al.: Mercury: hybrid centralized and distributed scheduling in large shared clusters. In: USENIX Annual Technical Conference, pp. 485–497 (2015)
15. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
16. Lee, D., Mehta, N., Shearer, A., Kastner, R.: A hardware accelerated system for high throughput cellular image analysis. J. Parallel Distrib. Comput. **113**, 167–178 (2018)
17. Li, M., et al.: Scaling distributed machine learning with the parameter server. In: 11th USENIX Symposium on Operating Systems Design and Implementation, vol. 1, p. 3 (2014)
18. Lu, C., Ye, K., Xu, G., Xu, C.Z., Bai, T.: Imbalance in the cloud: an analysis on Alibaba cluster trace. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 2884–2892. IEEE (2017)
19. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 69–84. ACM (2013)
20. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Proceedings of the 8th ACM European Conference on Computer Systems, pp. 351–364. ACM (2013)
21. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
22. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)
23. Vavilapalli, V.K., et al.: Apache Hadoop YARN: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, p. 5. ACM (2013)
24. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with borg. In: Proceedings of the Tenth European Conference on Computer Systems, p. 18. ACM (2015)
25. Xia, Y., Ren, R., Cai, H., Vasilakos, A.V., Lv, Z.: Daphne: a flexible and hybrid scheduling framework in multi-tenant clusters. IEEE Trans. Netw. Serv. Manag. **15**, 330–343 (2017)
26. Zhang, Q., Zhani, M.F., Boutaba, R., Hellerstein, J.L.: Dynamic heterogeneity-aware resource provisioning in the cloud. IEEE Trans. Cloud Comput. **2**(1), 14–28 (2014)