SN Computer Science (2023) 4:665
https://doi.org/10.1007/s42979-023-02101-8

ORIGINAL RESEARCH

Dynamic Resource Management for Machine Learning Pipeline Workloads

Min-Chi Chiang[1] · Lu-Wen Zhang[1] · Yu-Min Chou[1] · Jerry Chou[1]

Received: 18 October 2021 / Accepted: 28 June 2023

Abstract

The recent success of deep learning applications is driven by the computing power of GPUs. However, as the workflow of deep learning becomes increasingly complicated and resource-intensive, how to manage the expensive GPU resources for Machine Learning (ML) workload becomes a critical problem. Existing resource managers mostly only focus on a single specific type of workload, like batch processing or web services, and lacks runtime optimization and application performance awareness. Therefore, we aim to develop a set of runtime dynamic management techniques (including auto-scaling, job preemption, workload-aware scheduling, and elastic GPU sharing) to handle a mixture of ML workloads consisting of modeling, training, and inference jobs. In our previous work, we have implemented these techniques as a set of extended operators on Kubernetes. In this paper, we further extend our approach by introducing a topology-aware scheduling algorithm based on the hypergraph partition problem to minimize the communication cost of distributed training for maximizing the system throughput and minimizing the job completion time. Our evaluations on AWS GPU clusters prove our approach can out-perform the native Kubernetes by 60% system throughput improvement, 70% training time reduction without causing any SLA violations on inference services. Compared to the start-of-the-art topology-aware scheduling algorithm, we shorten the average job completion time by 24–44%.

Keywords Deep learning · GPU resource management · Job scheduling · Performance optimization

Introduction

Deep Learning (DL) is popular in data-center as an important workload for artificial intelligence, because it powers a variety of applications, including image classification [1, 2], object detection [3, 4], language processing [5–9] to self-driving cars [10] and autonomous robotics [11]. However, deep learning is also known to be computing intensive. As reported in a recent survey [12], the amount of computations used in the largest AI training runs has been increasing exponentially with a 3.4-month doubling time, which is at a pace even faster than Moore's Law. The increasingly popular trend of AutoML techniques, such as automatic hyper-parameter tuning and network architecture search, further pushes the need for computing power as models must be repeatedly trained with different settings to refine DL models. Today's deep learning production systems are mostly built on shared multi-tenant GPU clusters, where abundant computing resources can be utilized and shared among users to enable large-scale model training and highly efficient model inference serving. Therefore, it has drawn

This article is part of the topical collection "Cloud Computing and Services Science" guest edited by Donald Ferguson, Markus Helfert and Claus Pahl.

Additional Information: This paper is an extended work of our previous paper published in the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021.We extend the work by introducing and solve the topologyaware scheduling problem of distributed training jobs. The algorithm design and analysis complement the implementation of a resource management platform introduced in our previous work. 12 of 28 (43%) pages are newly added in this extended journal version.

✉ Jerry Chou
jchou@lsalab.cs.nthu.edu.tw

Min-Chi Chiang
mcchiang@lsalab.cs.nthu.edu.tw

Lu-Wen Zhang
luwen@lsalab.cs.nthu.edu.tw

Yu-Min Chou
ymchou@lsalab.cs.nthu.edu.tw

[1] Computer Science Department, National Tsing Hua University, 101 Kung-Fu Road Sec. 2, Hsinchu 300, Taiwan

Published online: 30 August 2023

increasing attention from industry and research communities to improve the efficiency and performance of the expensive resources for DL workloads.

Resource management (such as resource allocation and job scheduling) is one of the main approaches for improving job performance, system throughput, and hardware utilization. But managing the resources of DL workload can be challenging, because DL production needs several processing stages, from data pre-processing to model training, validation, and finally, model deployment for serving inferences. The workflow is also known as the *ML pipeline*. For simplicity, we refer to the computation before model training as the *modeling jobs*, the computations for modeling and hyper-parameter tuning as the *training jobs*, and the computations for serving model inference requests as the *inference jobs*. The workload characteristics of each of these jobs can be widely different. For instance, modeling jobs can be non-GPU bound jobs, because some of the data preprocessing tasks involve interactive data analysis, and others may hardly be paralleled using GPU, such as feature extraction in advertisement [13] and data augmentation in computer vision [14]. Training jobs behave like traditional parallel batch processing with huge and stable usage patterns. However, they can suffer from significant communication overhead if the tasks of a job are placed across computing nodes. In contrast, inference jobs behave like web services with short and bursty GPU usage patterns. However, they have strict SLA requirements on the request response time. The mixture and diverse computing jobs in DL workload creates many difficulties as well as opportunities from the resource management perspective.

Existing cluster schedulers (e.g., Borg [15], YARN [16]) are designed for general-purpose workload. Their scheduling algorithms (e.g., DRF [17], TetriSched [18], corral [19], HTCondor [20]) are mostly designed to ensure resource fairness and utilization by allocating a fixed amount of resources for each job according to the resource requirements specified by the job owner upon job submission. As a result, the existing approach may lead to sub-optimal application performance and system throughput due to the following reasons.

1. Application-oblivious scheduling: Existing schedulers only concern about the amount of resources allocated to each job without being aware of the application performance and resource characteristics. For training jobs, the location of allocated GPUs can significantly affect training time due to communication overhead. For inference jobs, their performance is measured by SLA guarantee, which depends not only on the resource allocations but also on the time-varied service workload (i.e., inference requests from clients). For modeling jobs, they are consisted of non-GPU and interactive workloads, so their performance can be much less sensitive to GPU resources. Hence, without adequantly considering the performance impact from resource allocations can easily lead to under-utilization or under-provisioning problems.
2. Static resource management: Due to the workload diversity and time-varied resource demands of DL workload, dynamic resource management like job preemption, auto-scaling, is essential to guarantee application performance and resource utilization. However, existing cluster managers often require job owners to manually operate or re-submit their jobs to adjust the resources of jobs. The lack of resource management experience and information further discourages users from adjusting their job resources when necessary.
3. Coarse-grained GPU allocation: Due to the lack of multi-tasking management of GPU devices, the minimum granularity of GPU allocation today is a single GPU device. That means an application can have multiple GPUs, but each GPU can only be allocated to exactly one application. While DL jobs can be accelerated using GPUs, a single DL job may not always utilize the whole GPU card due to reasons like memory-bound training jobs with large batch sizes or large network models, non-GPU-bound modeling jobs with human interactions, and inference jobs with time-varied workloads. Hence, existing GPU clusters without supporting GPU sharing can result in low resource utilization and system throughput.
4. Homogeneous workload consideration: Not until recently, DL domain-specific management systems [21–23] have been proposed to tackle the issues above. However, all of them focus on DL training jobs only. With the emerging trend of MLOps and AutoML that requires to unify ML system development (Dev) and ML system operation (Ops) together, end-to-end ML pipelines with mixture workloads are likely to be run and managed in a shared resource pool provided by a single production system. Therefore, resource managers should be designed with the consideration of jobs with different performance metrics, workload characteristics, and execution priorities.

To address the aforementioned problems, we aim to design and implement a series of dynamic resource management solutions to optimize the performance and resource utilization for DL pipeline workloads. This paper is an extension of our published work [24] for summarizing our approach. Our approach first designs and implements ***DynamoML***, which is a resource management platform on Kubernetes. Then, we propose ***HYPREL***, which is a hypergraph topology-aware scheduling algorithm for parallel computing jobs. *DynamoML* provides the management techniques and capabilities that can effectively optimize the resource usage of ML workloads with basic or user-specified resource

management policies. However, unlike the performance of modeling and inference jobs is only affected by the amount of resources, the execution time of a distributed training job is highly dependent on its task locations. Therefore, *HYPREL* is designed to further optimize the performance of distributed training jobs for minimizing the communication overhead and maximizing the computing efficiency.

Our experiments running a mixture of ML pipeline workload on a 16-GPUs cluster show that *DynamoML* improves the native Kubernetes by increasing the system throughput by more than 60%, reducing the average training time by 70%, and eliminating all the SLA violations. Our in-depth analysis also shows that without proper coordination and collaboration between different management techniques to balance the resource between training jobs and inference jobs, several issues could occur, including SLA violation on inference service, wasted idle GPUs, and prolonged model training time. Our *HYPREL* scheduling algorithm is evaluated by the real workload trace collected from a parallel system, and the results show that *HYPREL* can achieve up to 47% performance improvement than the topology oblivious method. Even compared to the other state-of-the-art topology-aware algorithms [25, 26], we still observed 44–24% improvement in the job completion time(JCT).

The rest of the paper is structured as follows. "Problem Description and Objectives" discusses the DL workload characteristics and overall system architecture. "DynamoML: Resource Management Platform" describes the design and implementation of *DynamoML*. "HYPREL: Topology-aware Scheduling Algorithm" presents the *HYPREL* scheduling algorithm. "Evaluations of DynamoML and Evaluation of HYPREL" are the evaluations for *DynamoML* and *HYPREL*, respectively. Finally, the related work discussion is in "Related Work", and the paper is concluded in "Conclusions".

Problem Description and Objectives

ML Pipeline Workload

In this work, we consider a GPU cluster running a set of computing jobs produced from the ML pipeline workflow. In general, the jobs can be classified into three types: modeling, training, and inference. The workload characteristics of these three types of jobs are summarized in Table 1, and briefly discussed as follows.

- Modeling: We use modeling jobs to represent all the computing jobs before model training. In general, modeling jobs involve neural network model building and interactive data pre-processing, including data cleaning, labeling, validations, and feature extraction. In practice, users normally perform these tasks through web notebooks, such as Juypter Notebook. Since the computations in this pipeline stage commonly involve data processing and human interaction, the GPU usage is low, some execution delay can be tolerated, and the workload pattern can be bursty.
- Training: Model training can be extremely time-consuming. In order to reduce training time, distributed model training across multiple GPU-nodes has been supported by mainstream deep learning frameworks, such as Tensorflow, PyTorch, and Keras. Most of them adopt the BSP (Bulk Synchronous Parallel) computing method to implement the data parallel model training, where a training job is consisted of a set of worker processes, and all the workers must synchronously aggregate their gradients at the end of each training iteration to update the model

Table 1 Description of ML pipeline workloads and management solutions

(a) Worload characteristics

Job type	Type	Pattern	Usage	Urgency
Modeling (e.g., Notebook)	Interactive data analysis	Bursty	Low	Med
Training (e.g., TFJob)	BSP (Bulk Synchronous Parallel)	Persistent	High	Low
Inference (e.g., TFServer)	Web service	Periodic	Med	High

(b) Resource management problems and solutions

Job type	Problems	Solutions	Results
Modeling (e.g., Notebook)	1. Low utilization	Fractional & elastic GPU allocation	Increase utilization
Training (e.g., TFJob)	1. Resource monopoly 2. Communication 3. Synchronization	1. Gang & locality aware scheduling 2. Task preemption	1. Avoid idle resources 2. Reduce training time
Inference (e.g., TFServer)	1. SLA requirement 2. Elastic workload	Auto-scaling	Avoid SLA violation

weights. Since training time can be long, modern deep learning libraries also support checkpoint mechanisms to tolerate faults and to restart training with different resource configurations. Therefore, comparing to the other two types of jobs, training has the highest resource usage and the lowest urgency.

- Inference: To serve the model's inference requests from clients, an inference job often packages the model and deploys it as a web service (e.g., TFServer). Similar to web services, the workload of a web service can be time-varied according to the number of client requests, and the SLA requirement of a service can be guaranteed. Therefore, inference jobs should have the highest urgency with periodic workload patterns and medium GPU usage demand.

System Components

Our resource management platform, *DynamoML*, is implemented as an extended framework on Kubernetes, because Kubernetes has become the most popular resource orchestrator for hosting containerized computing workload. However, Kubernetes, like our cluster schedulers (e.g., Borg [15], YARN [16], HT-Condor [20]), lacks proper resource management for DL workloads. The implementation details are given in "DynamoML: Resource Management Platform", and the scheduling algorithm is discussed in "HYPREL: Topology-aware Scheduling Algorithm". The main components of our resource management platform include the following modularized operators:

- Shared GPU allocator: it enables fine-grained shared GPU allocation in Kubernetes. Hence, the GPU utilization can be increased by allowing multiple non-GPU bound modeling jobs to share a single GPU. In comparison, the native device plug-in framework of Kubernetes does not allow fractional allocation.
- Distributed training job scheduler: It is a runtime scheduler that addresses several resource allocation problems of distributed model training jobs. (1) It reduces the communication overhead of distributed training by packing the workers of a training job on a single node. (2) It avoids the idle resource problem of synchronous computations by proving gang scheduling, so that all the workers of a training job will be scheduled together as a group. (3) It uses checkpoint mechanism to force resource preemption on training jobs, so the resources will not be monopolized by the long running training jobs. In comparison, the native Kubernetes scheduler cannot achieve these goals, because it lacks the awareness of application performance.
- Inference service auto-scaling controller: It aims to dynamicly add or remove the server instances of a service job according to the response time of inference requests, so that an application-level SLA requirement can be guaranteed under workload variations. In comparison, the existing scaling mechanism in Kubernetes is based on resource usage, not application performance. Furthermore, when the system lacks resources for inference jobs, auto-scaling controller can ask job scheduler to release the resources of training jobs. Therefore, inference jobs always have the highest scheduling priority in our framework.

Design Requirements and Strengths

Besides the goal of resource management, we also have the following strengths from its design requirements.

- Transparency: All the resource control mechanisms are implemented as the extended components of Kubernetes using the technique like custom controllers, sidecar containers, and library hooks. Therefore, no code modification to the deep learning computing frameworks or user programs. Because of transparency, our system can work seamlessly with Kubernetes, and our resource management strategy can also be applied to any application with the same targeted workload characteristics.
- Modularization: Each resource management component provides a standalone management service, such as GPU sharing, scheduling, and auto-scaling. These services are triggered by their service-defined API and event. Hence, system administrators can independently deploy individual components according to their needs.
- Agility: To overcome the workload variation and diversity of ML pipeline jobs, we focus on runtime resource sharing and management. From the resource allocation aspect, our GPU sharing supports *elastic allocation*, which allows the actual resource usage to be bounded between a user-specified range specified by a pair of values (request, limit), so that the resource within the range can be elastically shared among users. From resource usage aspect, we support auto-scaling and preemption on training and inference jobs, so that resource demands can be adjusted and adapted to the runtime application behavior and performance. Finally, our HYPREL job scheduling algorithm can further improve the computing efficiency and resource utilization with the consideration of network topology and resource availability.

DynamoML: Resource Management Platform

This section presents the detailed design and implementation of *DynamoML*.

Shared GPU Allocator

GPU sharing is necessary to improve resource utilization, especially for modeling and inference jobs. Our GPU sharing solution consists of two parts. The first is to enable fractional GPU allocation in Kubernetes. The second is to ensure the GPU resource can be shared fairly among containers.

The device plug-in framework of Kubernetes treats GPU devices as a single non-divisible resource object, so fractional GPU allocation is not allowed. To overcome this limitation, our system first launches a set of pods to allocate the GPU resources from Kubernetes, and obtain GPU devices' UUID. Then, the same GPU can be attached to multiple containers installed with nvidia docker package by setting the GPU's UUID in the environment variable "NVIDIA_VISIBLE_DEVICES". To support fractional allocation, our shared GPU allocator will track the residual resource amount on each GPU, and ensure GPUs are not over-allocated.

After a GPU is attached and accessible by a container, we still have to ensure the actual resource usage does not exceed the allocation amount. As shown in Fig. 1, to throttle the GPU usage of a container, we insert an LD_PRELOAD hook library in the container to intercept its GPU API. The intercepted GPU APIs will be blocked until the hook library receives an execution token from the scheduler of our GPU allocator. A time-sharing scheduler is implemented to pass the token around containers according to their resource allocation demand. Therefore, the resource usage of a container cannot exceed its allocated demand. To maximize GPU utilization, we further support elastic allocation, which allows users to specify their minimum and maximum demand as (request, limit), so that the requested resource can be reserved and guaranteed, while the residual capacity can still be utilized by the container without exceeding its limit. More details of the design and implementation can be found in our previous work [27].

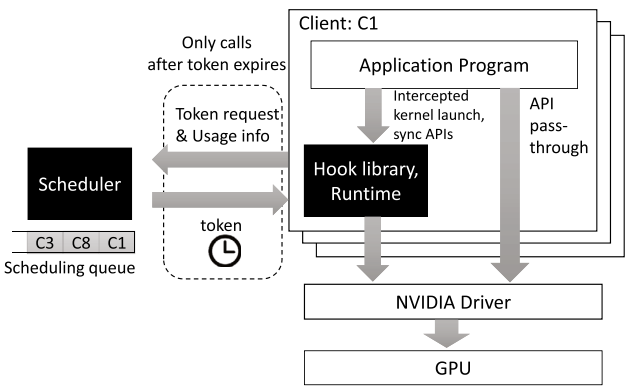


Fig. 1 GPU usage control framework

Training Job Scheduler

The native Kubernetes scheduler schedules the container (i.e., pod) of each worker task independently on random nodes with sufficient resources. However, distributed training jobs are communication-bound with task dependency among each other. Therefore, the naive FCFS random scheduling algorithm of the native Kubernetes scheduler can cause significant communication and synchronization overhead.

To address the above issues, we developed our own job-level scheduler on top of the task-level native scheduler to schedule and manage all the tasks of a training job as a unit. Through our scheduler, a job is only launched when the system has enough residual capacity to run all its tasks simultaneously. Thus, resources will not be occupied by tasks waiting for synchronization. To minimize the communication overhead of a job, our scheduler tends to pack all the tasks of a job on as fewer number of compute nodes as possible. The location of the task is controlled by specifying the "node selector" label in the pod spec of the workers, so the native scheduler can only create the pods on the designated nodes of our scheduler.

Finally, our scheduler monitors the system resource usage status to dynamically adjust the number of workers of training jobs. Additional workers are added to training jobs when the system loading is low, so that jobs can take advantage of the residual capacity to reduce their execution time. On the other hand, workers can also be taken from jobs when the system loading is high, so the resources can be reclaimed from the running jobs to launch the waiting jobs as soon as possible. More details of the design and implementation can be found in our previous work [28].

Inference Auto-scaling Controller

To dynamically scale up and down in response to a varied number of users' requests, we integrated Kubernetes Horizontal Pod Autoscaler (HPA). However, the naïve Kubernetes Horizontal Pod Autoscaler (HPA) is limited to scale pods according to the current state of CPU or memory consumption. The most direct way to scale pods should depend on the current number of users' requests, which requires us to integrate third-party solutions to fulfill this goal. We further integrate Istio, a popular tool to build service mesh in our implementation, to gather the requests' information within the cluster.

The term service mesh is used to describe the network of microservices that make up such applications and their interactions. Its requirements can include service discovery,

load balancing, metrics, and monitoring. Istio's solution creates a network of deployed services with traffic management, security, observability, and extensibility. It directly attaches a sidecar, a proxy to help pod exchange information, to every pod. Whenever there exists a need to pass data between pods, the sidecar intercepts all network communication. By intercepting the communications, Istio generates metrics for all service traffic in, out, and within an Istio service mesh. These metrics provide information on behaviors, such as the overall traffic volume, the error rates within the traffic, and the response times for requests. As a result, we use Istio as our primary tool to gather communications metrics.

As shown in Fig. 2, when we successfully fetch the metrics from Istio, we need to store the information for Kubernetes Custom Metrics API to query. Although we can forward data from Istio to Kubernetes Custom Metrics API directly, we could further store, query, and monitor the metrics if we store them in Prometheus. By employing Prometheus, a popular Time-Series Database, to periodically forward information to Prometheus, we gather the metrics, including the request number per second and response time. Then, since Kubernetes HPA is only able to fetch metrics from Kubernetes Custom Metrics API, we install the Prometheus Adapter and register it to Kubernetes Custom Metrics API. Finally, Kubernetes HPA is capable of scaling up and down pods according to the current number of users' requests.

In prioritize the inference jobs and preempt the training jobs as needed, whenever it is necessary to increase the inference pods in response to flooded users' requests, our auto-scaling controller would check whether current residual resources are available for inference jobs to scale up. If not, we would sequentially evict the workers of training jobs (TFJobs), release sufficient GPU resources for inference pods. We leave at least one worker (chief worker) for every TFJobs in our implementation, so that TFJobs can proceed training process continuously without requiring checkpoint. Finally, the Kubernetes API scheduler would allocate GPU resources for the new inference pods, which makes platform reach inference jobs' Service Level Agreement (SLA).

In our implementation, we label the inference pods first, making Kubernetes Operator capable of identifying inference jobs. Then, by Kubernetes native event-driven mechanism, we registered OnAdd, OnUpdate events of inference pods. Whenever Kubernetes HPA decides to scale up inference pods, the training job scheduler would be notified by the Kubernetes control manager, check whether the current residual resource is available for the new pod, and decide whether to release the resources for inference jobs by scaling down training workloads. In addition, training job scheduler would periodically check the residual resources. After Kubernetes HPA scaled down inference pods and released the GPU resources, the training job scheduler would scale up training jobs once it detected residual resources.

HYPREL: Topology-Aware Scheduling Algorithm

Unlike inference jobs and modeling (i.e., notebook) instances, the execution time of distributed model training jobs can be significantly affected by the communication

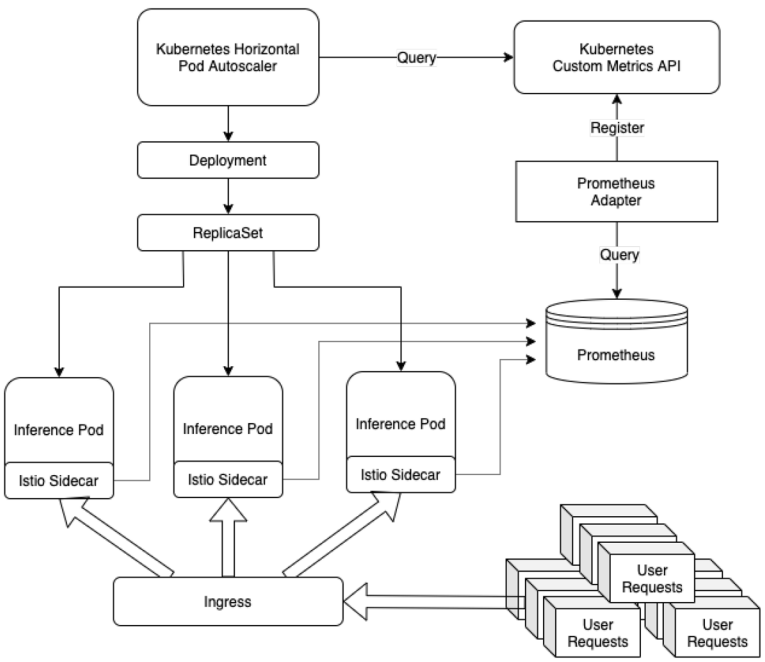


Fig. 2 System design diagram of the inference auto-scaling controller

overhead. Therefore, the scheduling and task placement decision of training tasks is critical to the training time and resource usage efficiency. The scheduling and scaling mechanisms of DynamoML can mitigate the problem with some dynamic resource management strategies driven by performance and resource utilization. However, the initial placement of the tasks is still important, because a bad initial task placement decision can lead to many disadvantages at runtime, such as sub-optimal job performance and higher task reconfiguration overhead. Therefore, we propose ***HYPREL***, a topology-aware scheduling algorithm for optimizing the performance of training jobs. *HYPREL* achieves better performance than other state-of-the-art topology-aware scheduling algorithms [25, 26] for two main reasons. First, it uses the hyper-graph data structure as an efficient and effective approach to address the task locality problem and minimize communication overhead. Second, it allows job re-ordering to maximize the benefits of task placement.

In the rest of the section, we first give an overview of our scheduling problem. Then, we introduce our task placement strategy based on k-way hypergraph partition algorithm and our job selection strategy for re-ordering the scheduling jobs. Finally, we detail and analyze the complexity of our algorithms.

Approach Overview

The scheduling problem we consider is to schedule a set of jobs waiting in the scheduling queue onto a set of computing devices (i.e., slots). The computing devices are connected by a network topology, and each job is consisted of a set of parallel tasks. The goal of the scheduling algorithm is to minimize the average job completion time. Noted, we are considering an online scheduling problem. Thus, the scheduler is triggered whenever a job arrives or departs from the system, and the scheduler does not aware of the future jobs that are not inserted in the waiting queue. To ensure our approach can be used in a practical system environment, we assume the scheduler has no prior knowledge of the job information, except the number of tasks requested by a job.

The scheduling problem is challenging, because it needs to address three problems at the same time. The first problem is the communication overhead. As known, the execution time of a parallel job can be affected by the task location because of the communication overhead. Therefore, the tasks of a job should be packed within a close distance on the network topology to minimize communication time. However, if we only allow a job to be scheduled on its closest locations, such as packing into a single node, we may encounter the resource fragmentation problem, where fragmented scattered across nodes cannot be fully utilized. Finally, the job scheduling order can affect the scheduling results as well. We consider the jobs in the waiting can be executed out of order for optimization, but starvation and long queuing delay problem should be avoided.

Our proposed topology-aware scheduling algorithm is called *HYPREL*. The overall design of the scheduling algorithm is shown in Fig. 3. The approach consists of two main steps: (1) job candidate set selection and (2) hypergraph min-cut partition. When the number of available slots is fewer than the total requested slots from the jobs in the queue, the job candidate set selection strategy is applied to select a subset of jobs from the queue and ensure they can be run on the available slots immediately. The hypergraph min-cut partition strategy is then applied to the selected job candidate set to decide the slot location for each task of the jobs.

To address the three aforementioned problems (naming communication, fragmentation, and starvation), we have the following four design principles in our approach. The details of our task placement and job selection algorithms are given in "Hypergraph Task Placement Strategy" and "Prioritized Job Selection Strategy", respectively.

1. We give higher scheduling priority to the job with a larger size (i.e., the number of requested slots). Larger jobs are more likely to suffer longer queuing delays and starvation, because their requested resources are more. Hence, higher scheduling priority gives them a better chance to be run without being blocked by a group of smaller jobs.
2. We formulate the task placement problem as a hypergraph partition problem, so the network topology information can be taken into account to minimize the communication cost of jobs.
3. To exploit the benefit of scheduling re-ordering, we generate not one candidate but a set of candidates from the job selection step, so we have the opportunity to choose

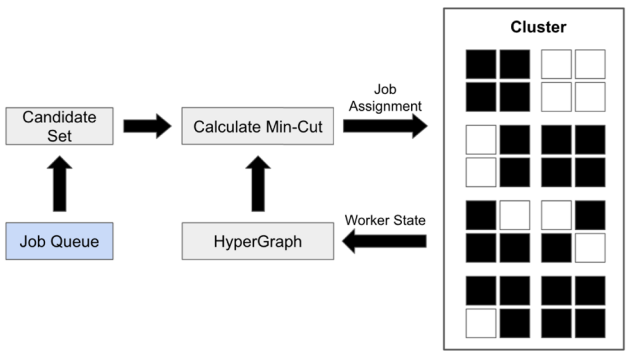


Fig. 3 Workflow of our proposed *HYPREL* scheduling algorithm. The network topology is characterized as a hypergraph. Our scheduling algorithm first selects a subset of jobs from the waiting queue without exceeding the residual capacity, and then applies a hypergraph min-cut partition algorithm to decide the task placement of these jobs to minimize the communication overhead

the job set with the lowest communication cost in the final scheduling decision.

4. To ensure resources are fully utilized, we aim to maximize the allocated slots of a candidate job set without exceeding the residual resource capacity.

Hypergraph Task Placement Strategy

This subsection explains how our task placement problem of a set of k jobs can be mapped onto a k-way hypergraph partition problem and proves why the result of the min-cut partition optimization problem can lead to better job locality and lower communication cost.

A hypergraph is a graph data structure that can have multiple nodes connected by an edge. Hence, different from the normal graph structure, an edge in hypergraph is represented as a set of nodes. This kind of modeling can simplify the complexity of the physical network topology and describe the physical network topology more concisely.

In our scheduling problem, our goal is to decide the computing slot (i.e., either a GPU device or a CPU core) for running a computing task. In addition, we assume the system is connected by a hierarchical network topology. Therefore, consider each of the computing slots as a leaf node in a hypergraph. The rest of the interconnect system components (including CPU sockets, computer nodes, network switches, and routers, etc) as the intermediate nodes, and the interconnect links (including network links, PCI bus, etc) as the edges in a hypergraph.

Figure 4 shows an illustration of hypergraph mapping for a cluster with 8 computing devices connected by a two-level tree network topology. As shown, the eight vertices

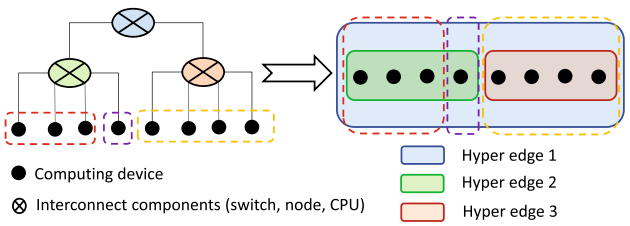


Fig. 4 Example of hypergraph partitioning result that maps to a scheduling decision on a network topology. The left subfigure is a three-level network topology connected with 8 computing devices. The right subfigure is a hypergraph, where each vertex represents a computing device, and each edge (i.e., colored block) represent an intermediate node in network topology. Giving a task placement of three jobs indicated by the dashed bounding box on the left side, its corresponding hypergraph partition result is shown on the right side. Each partition in the hypergraph represents the task placement of a job. A cutting edge between partitions denotes a parent node that is the common ancestor of the two jobs, but is not used by either of the jobs. The connectivity cardinality of the three edges are 3, 2, 1, respectively. Hence, if the weights of all edges are 1, the total edge cutting cost from the partition is $f_\lambda(\Pi) = 2 + 1 + 0 = 3$

in the hypergraph are the eight computing slots, while each block in the figure is a hypergraph edge representing a router in network topology. From the example, we can also observe that a hypergraph is capable of characterizing the hierarchical topology structure. The two inner blocks in green and red represent the two 2nd level routers, and the outer block in blue represents the 1st level router.

The k-way hypergraph partition problem has been well-studied. It is NP-hard problem [29], but exists linear time heuristic algorithms [30, 31]. The problem is formally defined as follows. Given a hypergraph with a set of vertices V and edges E, the k-way hypergraph partitioning problem definition is to find a way to divide the vertex set, so that the divided k blocks satisfy:

$$\begin{aligned} &\Pi = \{V_1, \ldots, V_k\} \\ &\text{such that } \bigcup_{i=1}^{k} V_i = V \\ &V_i \neq \emptyset, 1 \leq i \leq k \\ &V_i \cap V_j = \emptyset, i \neq j \end{aligned} \tag{1}$$

The min-cut algorithm for k-way hypergraph partitioning problem aims to find the partition solution with the minimum edge cutting cost. The edge cutting cost can be formally defined as follows. For each edge e, the connectivity set of the edge e can be defined as

$$\lambda(e) := \{V_i | V_i \cap e \neq \emptyset\} \tag{2}$$

The cardinality of each connectivity set can be denoted as $\lambda(e) := |\Lambda(e)|$, which also means the number of blocks connected to the edge e. If the edge e satisfies $\lambda(e) > 1$, then this edge is called a cut edge. Otherwise, if $\lambda(e) = 1$, it is called an internal edge. Let E' be the set of all cut edges from a partition solution, the cost function of the min-cut algorithm is defined as

$$f_\lambda(\Pi) := \sum_{e \in E'} (\lambda(e) - 1)\omega(e), \tag{3}$$

where $\omega(e)$ is the weight of the edge e.

Take the partition in Fig. 4 as an example. If we let all the edge weights equal to 1, the total cutting cost is 3, because the cutting costs of the three edges in blue, green and red are 2, 1 and 0, respectively.

To map our task placement problem to a k-way hypergraph partitioning problem, we let k be the number of jobs and add the constraint for ensuring the size (i.e., number of vertices) of a block V_i must be the same as the number of tasks of a job i. The edge weight is assigned to be the reciprocal of the farthest distance between all slots under its corresponding node in network topology, so that the nodes closer to the root have lower edge weight. Table 2

Table 2 Description and mapping of mathematical symbols between the hypergraph partitioning and job scheduling problems

Hypergraph		Scheduling Problem	
Term	Definition	Term	Definition
v_i	v_i is a vertex	t_i	t_i is a task
e_j	e_j is an edge that contains a set of elements	r_j	r_j is an interconnect component
V_k	A partition that contains a set of elements	j_k	j_k is a job
$v_i \in e_j$	Vertex v_i is included in edge e_j	$t_i \in r_j$	Task r_i contains task t_i when t_i is UNDER node r_j in the network topology
$v_i \in V_k$	A vertex v_i is included in partition V_k	$t_i \in j_k$	Task t_i belongs to job j_k

summarizes the notations and their corresponding mapping relationship between the hypergraph partitioning problem and our scheduling problem.

Finally, we provide Lemma 1 and Theorem 1 to prove the min-cut hypergraph partition algorithm can also minimize the communication cost of our task placement solution.

Lemma 1 *An edge in hypergraph must be cut if and only if there exist tasks from different jobs are under the node corresponding to the edge in the scheduling problem.*

Proof Based on our problem mapping, a node has tasks from different jobs, $t_i \in r_j$ and $t_{i'} \in r_{j'}$. It implies the edge corresponding to the node in the hypergraph must also contain vertices from different partitions, $v_i \in e_j$ for $v_i \in V_k$, and $v_{i'} \in e_{j'}$ for $v_{i'} \in V_{k'}$, $k \neq k'$. By definition, an edge in hypergraph must be cut if it contains the vertices from two different partitions. Therefore, the edge corresponding to a node with tasks from different jobs must be cut.

Similarly, if an edge is cut, it must contain the vertices from two different partitions. Hence, the node corresponding to the edge must also contain the tasks from different jobs. □

Theorem 1 *A min-cut partition solution in hypergraph tends to improve the job locality and thus reduce the execution time of jobs.*

Proof In a min-cut partition problem, we know a partitioning algorithm must aim to reduce the number of cutting edges and avoid the edges with larger weight to be cut. We prove that these two optimization objectives can both lead to better job locality and thus short job execution time.

To reduce the number of cutting edges in a hypergraph, we must reduce the number of nodes shared among jobs, according to Lemma 1. The number of shared nodes can only be reduced if we can limit the number of nodes that contain the tasks of a job. As known, only when the jobs are packed in the minimum number of nodes in the closest distance, the number of nodes of a job can be minimized. Therefore, a min-cut partition algorithm for hypergraph will try to pack the tasks of a job in the minimum distance which also leads to better job locality and execution time.

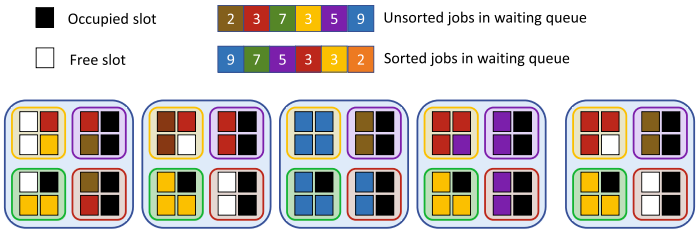
If we want to avoid the edges with larger weights to be cut in a hypergraph, we must ensure these edges only contain the vertices from a single partition. Therefore, a min-cut algorithm should prefer to pack the edge with larger weights with the vertices from the same partition with higher priority. Based on our edge weight assignment rule, an edge with higher weight also implies a node with a smaller distance among its computing slots. Therefore, a min-cut algorithm will pack the computing slots with smaller distances by a single job with higher priority, which also leads to better job locality and execution time. □

Prioritized Job Selection Strategy

This subsection details how we select the job set from the waiting queue for the task placement algorithm. The only constraint from job selection is the total requested slots from the selected jobs cannot exceed the current available slot count. The traditional FIFO job selection strategy will sort the jobs by their arrival time and pick the first k jobs without exceeding the available slot count.

The FIFO strategy is simple and avoids the starvation problem, but it misses the opportunity to select a better job set for the task placement algorithm to achieve lower communication costs and higher resource utilization.

Let us consider the scheduling example shown in Fig. 5. Under FIFO scheduling order, we can only run the 3 of the 4 jobs in the waiting queue, and three slots cannot be utilized by the other jobs in the waiting queue. Furthermore, the job ordering decision can also affect the task placement decision and communication. For instance, the commonly-used packing strategy which aims to pack jobs in a single node may get a higher communication cost if the job with only 2 tasks is put on the node with 4 slots first. The job selection strategy can also have a great impact on our hypergraph partition algorithm. As shown in the example, each of the three different job selection decisions for *HYPREL* results in different utilization and communication. Therefore, it is



Method	FIFO & random	FIFO & packing	HYPREL 1st candidate	HYPREL 2nd candidate	HYPREL 3rd candidate
Scheduled jobs	{2,3,3}	{2,3,3}	{9,2}	{5,3,3}	{3,3,2}
Comm. Cost (edge cost=1)	5	3	1	2	2
Utilization	8/11	8/11	11/11	8/11	8/11

Fig. 5 Comparison of the scheduling results from different task placement and job selection algorithms. The number of the jobs indicates their requested number of slots (or tasks). The communication cost is computed by Eq. 3 under the setting of $\omega(e) = 1, \forall e$. As observed, the job order can affect the task placement decision and communication cost. For instance, FIFO with packing strategy can lead to a sub-optimal solution when it is unaware of the other waiting jobs and schedules the jobs with 2 tasks to the node with 4 available slots. *HYPREL* evaluates all three candidates and picks the first one which has the highest utilization and lowest communication cost

important to exploit job selection decisions for optimizing the overall system performance.

However, there can be a total of $O(2^n)$ job selection options from n jobs. To exploit out-of-order job selection with lower computing complexity, we propose an effective heuristic algorithm with the following steps.

Step1: The jobs in waiting queue are sorted by their job size (i.e., number of requested slots) in descending order. This allows larger jobs to be selected first, so that can avoid starvation or long delay problems.

Step2: We scan through the sorted job list to generate the candidate job set by selecting the jobs can fit into the residual resource capacity with the first-fit policy. This step stops when no more jobs can fit into the residual capacity, so the resource utilization can be maximized.

Step3: We try to explore more job selection options by repeating Step2 to generate n job selection candidates, where n is the number of jobs in waiting queue. For the i^{th} candidate job set generation, we add a job filtering condition to remove the top $(i-1)^{th}$ largest jobs from waiting queue. Our intention is to ensure that the largest job first will not always dominate the selected job set. We are also capable to generate candidate sets that consists of smaller jobs.

Step4: To ensure the resource utilization is maximized, we introduce an utilization threshold δ to filter the candidates from Step4 whose utilization is less than δ. If none of the candidates can satisfy this threshold, we keep the candidate with the highest utilization.

Step5: Finally, we apply the hypergraph partition algorithm introduced in Hypergraph Task Placement Strategy to evaluate the communication cost from each of the job set selections from the candidates and choose the job set that results in the lowest communication cost.

Algorithm and Complexity Analysis

According to the description of our approach mentioned above, we summarize the pseudo code of our complete algorithm in 1–3. Algorithm 1 is the overall scheduling algorithm of *HYPREL*, which call `hyperAlloc()` and `findCandidateSet()` to identify the best job selection and task placement result. The `hyperAlloc()` function can be any existing hypergraph partition library implementation or algorithm, such as KaHyPar [30]. Algorithm 2 is `findCandidateSet()`, which is the implementation of our job selection strategy described above. Finally, Algorithm 3 is the search function for selecting the jobs from the waiting queue described by Step2. Finally, we provide the overall complexity of our algorithm is $O(n^2)$ in Theorem 2.

Algorithm 1 `HYPREL`(*Jobs*, *Slots*)

Require: *Jobs*: the set of jobs in waiting queue
Require: *Slots*: the available slots in network topology
1: min_cost = ∞
2: *CandidateSet* = `findCandidateSet`(*Jobs*, *Slots.size*)
3: **for** each *JobSet* ∈ *CandidateSet* **do**
4: < *placement*, *cost* >= `hyperAlloc`(*JobSet*, *Slots*)
5: **if** *cost* < *min_cost* **then**
6: *best_placement* = *placement*
7: *min_cost* = *cost*
8: **end if**
9: **end for**
10: return *best_placement*

Algorithm 2 findCandidateSet(*Jobs*, *NumSlot*)

Require: *Jobs*: the set of jobs in waiting queue
Require: *NumSlot*: the number of available slots
1: $CandidateSet = \emptyset$
2: $maxAllocSize = 0$; $maxAllocIdx = null$
3: sort the jobs in *JobPool* in the descending order of job size
4: **for** $i = 1$ to $|Jobs|$ **do**
5: $Candidate[i] = \emptyset$
6: $allocSize = 0$; $JobPool = Jobs.remove\_head(i-1)$
7: **while** $j =$ findNextJob$(JobPool, NumSlot) \neq null$ and $numSlot > 0$ **do**
8: $Candidate[i].add(j)$
9: $allocSize+ = j.size$; $numSlot- = j.size$
10: **end while**
11: **if** $Candidate[i].size/NumSlot > \delta$ **then**
12: $CandidateSet.add(Candidate[i])$
13: $NumSlot- = allocSize$
14: **end if**
15: **if** $allocSize > maxAllocSize$ **then**
16: $maxAlloc = allocSize$; $maxAllocIdx = i$
17: **end if**
18: **end for**
19: **if** $CandidateSet == \emptyset$ **then**
20: $CandidateSet.add(Candidate[maxAllocIdx])$
21: **end if**
22: return *CandidateSet*

Algorithm 3 findNextJob(*obPool*, *NumSlot*)

Require: *JobPool*: the set of jobs in waiting queue
Require: *NumSlot*: the number of available slots
1: **while** $j = JobPool.dequene() \neq null$ **do**
2: **if** $j.size \leq NumSlot$ **then**
3: return j
4: **end if**
5: **end while**
6: return null

Theorem 2 *The overall time complexity of HYPREL* (*i.e.*, *Algorithm* 1) *is approximated to* $O(n^2)$, *where n is the number of jobs in waiting queue.*

Proof The algorithm is consisted of two parts: job selection and task placement. According to Algorithm 2, the job selection complexity is $O(n^2)$, because the complexity of generating a candidate job set is $O(n)$, and we need to generate n candidates. On the other hand, the complexity of task placement is depending on the hypergraph partition algorithm adapted by *HYPREL*. Since we have to apply the partition algorithm on each of the job set candidates. The total complexity of task placement is $O(n \times C)$, where C is the complexity of the hypergraph partition algorithm adapted by *HYPREL*. Since several linear heuristic partition algorithms exist based on well-known greedy optimization methods, such Louvan [30] and Fiduccia-Mattheyses (FM) [31], we can approximate our complexity to $O(n^2)$. □

Evaluations of DynamoML

Experiment Setup

We evaluate our implementation by conducting the experiments on AWS cloud platform using a Kubernetes cluster consisting of 2 nodes (p3.16xlarge instance type). Each node is equipped with a 64-cores CPU (Intel Xeon E5–2686 v4), 488GB of RAM, and 8 Nvidia Tesla V100 GPUs with 128GB of device memory. To conduct a comprehensive evaluation of our implementation, we design workloads that include the computing jobs for modeling, training, and inference. The training and inference jobs are based on the popular DL framework—Tensorflow. The modeling jobs are based on the Jupyter Notebook, which is a primary tool for developers to build and test their models.

For the training jobs, we employ two common-seen image classification model training tasks: The three-layer CNN model training task for the Mnist data set (i.e., the task is referred as Mnist in the rest of the paper), and the ResNet-50 model training task of the ImageNet data set (i.e., the task is referred as ResNet-50 in the rest of the paper). Every TFJob would consist of one Parameter Server (PS) and several workers, where workers can be added or removed in response to the state of the residual resource in our implementation. Each worker requests one GPU, and a TFJob runs for fixed training iterations. As shown in Fig. 6, both models can have higher training throughput and shorter training times when using more GPUs. The speedup of ResNet-50 is close to linear. The speedup of Mnist does not increase much with more than 3 GPUs, because its model is too small to have enough computations for parallel processing. In the experiments, we set the maximum number of workers to 4 for both models.

For the inference jobs, we use a Resnet-50 model as our inference service for all requests. Inference job runs on TF-Serving applications, which computes the forward propagation upon each arrival client request. Hence, its GPU usage

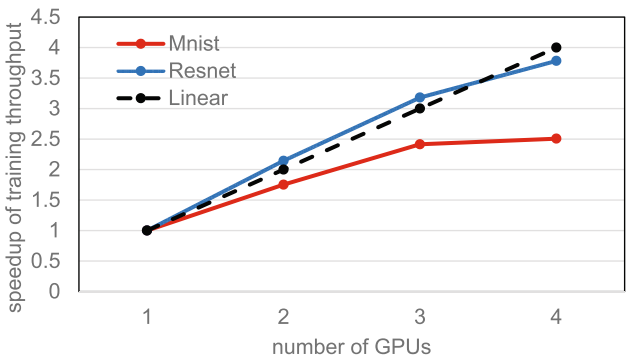


Fig. 6 Speedup of model training throughput of using multi-GPUs

is also approximately proportional to the number of client requests. In our experiment, One TF-Serving inference pod is configured to consume only one GPU. As a result, the number of inference pods would dynamically increase or decrease in response to the number of active clients. For the purpose of evaluating the response time of inference jobs under different workloads, a web client would keep sending requests with a varied inter-arrival time.

Finally, for the modeling jobs, we sporadically issue model evaluation requests to the Jupyter Notebook instances, and the average GPU usage of a modeling job never exceeds 25%.

To evaluate the benefits of our resource management techniques in a runtime system. We construct a testing work, as shown in Fig. 7. It contains a total of 4 modeling (notebook) jobs, 1 inference job, and 16 training jobs. The inference jobs and modeling jobs are persistently running in the system, but their user workload changes over time. As mentioned above, the modeling has a sporadic random generated workload with less than 25% GPU usage. The workload of the inference job controlled by adjusting the number of concurrent active web clients in each time interval (per minute) which can be seen from the bars at each minute in Fig. 7. Finally, the 16 training jobs are submitted to the system every 10 min in 4 groups with 4 jobs per group. The first group arrives at 0^{th} minute with 4 Mnist training jobs. The second group arrives at 10^{th} minute with 4 Resnet-50 training jobs. The last group arrives at 20^{th} minute with 2 training jobs for each of the two models.

The goal of our evaluation is to compare the system and application performance running the above test work under different resource management configuration settings. The names and resource management techniques of each setting are summarized in Table 3. The initial number of workers for a training job is 2, and the initial number of server instance for a inference job is 1. Both training and inference jobs can be scaled upto to 12 instances (training workers or inference servers) when auto-scaling techniques are applied (i.e., the total number of GPUs in our testbed is 16.). By default, all types of jobs request one GPU per container instances (i.e., pods). Only when GPU sharing technique is applied, a modeling job can allocate 0.25 GPU.

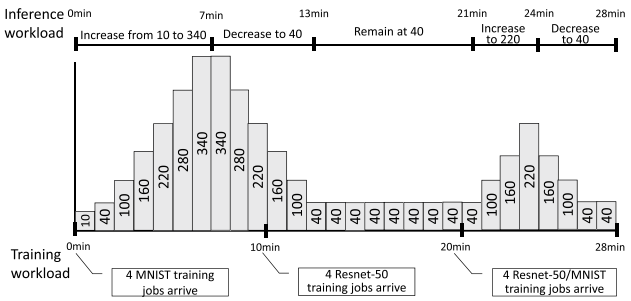


Fig. 7 Testing workload for system evaluations. The workload of interface jobs are time-varied by adjusting the number of active users. The workload of training jobs are submitted in 4 groups and 4 jobs per group at time 0, 10, 20. The modeling jobs are consisted of 4 notebook instances persistently running throughout the experiments

Table 3 Compared system configurations

	Training Scheduling	Inference Auto-scaling	GPU Sharing
DynamoML	V	V	V
K8S+Sharing			V
K8S+Scaling		V	
Native K8S			

The setting with a resource management technique is marked by "V"

System Performance Comparison

Figure 8 plots the GPU resource allocation results over each time interval under different system settings. Due to space limits, we show the results of DynamoML and K8S+Scaling to illustrate the key benefit of our approach. As mentioned, DynamoML supports all three proposed techniques: inference scaling, training scheduling, and GPU sharing. K8S+Scaling only supports inference scaling. K8S+Scaling also represents the common use case when people are only able to use the native Kubernetes installation with the HPA auto-scaling package to run ML workloads.

According to the workload variation of inference job, the allocation result can be discussed in the following three time frames from the DynamoML timeline.

0min~7min: The inference workload keeps increasing during this time frame. Our auto-scaling operator detected the increased request response time, and started to launch more inference servers. Hence, the number of GPUs allocated to inference increases from 1 to 12. Noted, the inference job can occupy as many as 12 GPUs, because the training jobs are preempted and forced to release their GPUs for inference jobs. Therefore, some the training job only allocated 1 GPU at times. On the other hand, when there is residual capacity or un-used GPUs, they can also be dynamically allocated to the training jobs for reducing training time. Therefore, the GPU allocation during this time frame is almost always fully utilized.

8min~20min: As the inference workload decreases, we can observe DynamoML quickly allocate the available GPUs free from the inference job to the training jobs. As a result, a training job can use up to 4 GPUs at a time, and the training time is greatly reduced. Because the training job finishes too early, only 4GPUs needs to be used for handling the inference requests between time 18min~20min which can lead

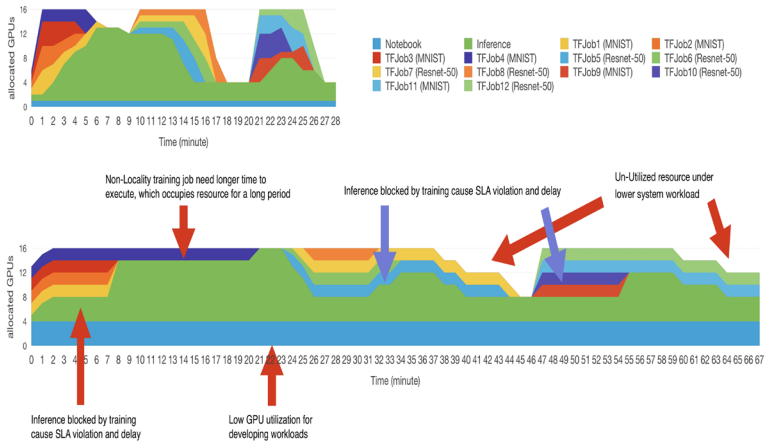


Fig. 8 Overall resource allocation for DynamoML (top) and K8S+Scaling (bottom)

to an additional benefit of energy and cost saving for system administrators or service providers.

21min~28min: The last group of the training job arrives at 20min, and the inference workload also starts to increase at 21min. So similar to the first time frame, both training jobs and inference jobs can get more resources at runtime, but the inference jobs have higher priority than the training jobs. In addition, because of the lower inference workload in this time frame comparing to the first time frame, training jobs received more GPUs and complete all the training jobs before 27min.

In comparison, we can observe several problems from the K8S+Scaling setting. (1) It fixes the allocation of training job to 2 GPUs, and each modeling job (Notebook instance) occupies 1 GPU. Hence, even though the inference job can be scaled to obtain more GPUs, it can only use the residual capacity from training and modeling jobs. Therefore, between 0min to 7min, the inference job only receives 4 GPUs, while it would receive 11 GPUs by DynamoML. (2) The Kubernetes scheduler did not pack the workers of a training on the same node which results in much longer training time. In particularly, the communication overhead has a greater impact on the small size models, like MNIST, because their communication time often takes higher ratio of the overall execution time. Therefore, compared to the results of DynamoML, TFJob4, TFJob7 and TFJob11 all took much longer time to finish under the K8S+Scaling setting. (3) Because the resource of training jobs is fixed, they can take advantage of the residual capacity in the system when system workload is light. For instance, there are GPUs available between 37min to 46min, and 59min to 67min, but they cannot be allocated to training jobs and cause unwanted resource waste.

In sum, we can observe the important of dynamic resource management when running complex and diverse workloads on a shared resource pool. With our techniques, the overall workload execution time is significantly reduced from 67mins to 27mins, an improvement of almost 60%, and there are still rooms for us to free-up some idle resources for energy or cost saving. More importantly, DynamoML can improve resource utilization and training performance while guaranteeing the SLA requirement of inference jobs. In the next two subsections, we further analyze the performance of training jobs and the SLA violations of inference jobs to analyze the reasons of our improvement.

Training Time Analysis

This subsection analyzes the impact of our resource management techniques on the training jobs from running test workloads shown in "Experiment Setup". Because a training job has lower execution priority and can be queued in the submission queue when not enough resources are available, the total execution time of training can be divided into two parts: the training time and the waiting time. The training time is the actual running time for training, and the waiting time is the time of a job waiting in the scheduling queue. Therefore, we compare the improvements of these two time measurements, and the total execution time in Fig. 9. Interestingly, we found that K8S+Scaling produces the worst results across all

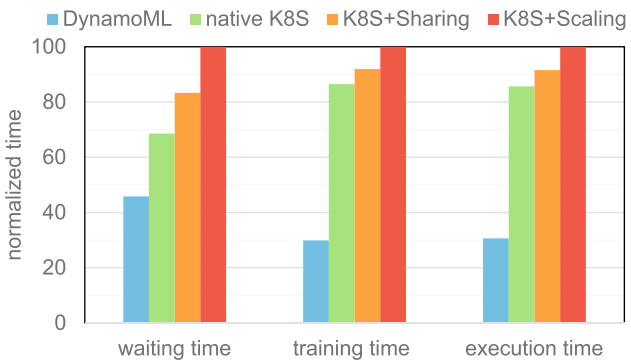


Fig. 9 Training time comparison

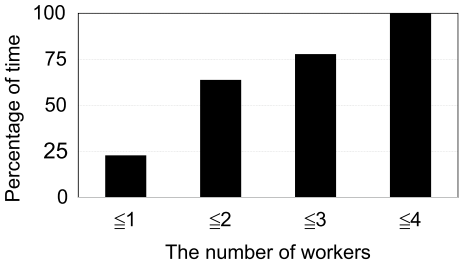


Fig. 10 Distribution of the number of worker per TFJob under DynamoML

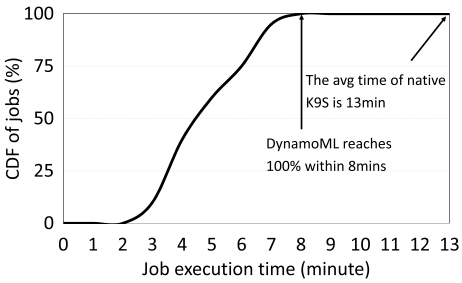


Fig. 11 Distribution of the training job execution time under DynamoML

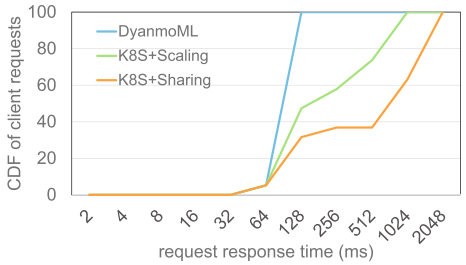


Fig. 12 Comparison of the distribution of the response time from inference jobs

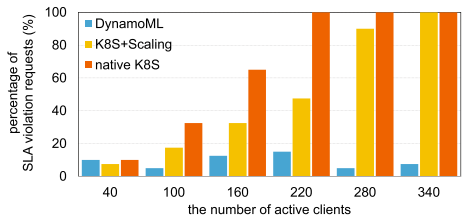


Fig. 13 SLA violation of inference jobs under different number of active users

the measurements, even worse than the native K8S. This is because K8S+Scaling only optimizes for the inference jobs not for the training jobs. Hence, to satisfy the SLA requirements, it allow inference jobs to allocate more resources, and sacrifices training jobs. As a result, training jobs have a higher probability to be blocked or running with few GPUs. K8S+Sharing is better than native Kubernetes, because it reduces the amount of GPUs used by the modeling jobs, so training jobs can gain better performance. However, the improvement of GPU sharing for training job is limited, because the communication overhead is the main performance bottleneck for training jobs as we saw from the cases like TFjob4 in Fig. 8. In comparison, DynamoML combines all our techniques to significantly reduce the waiting time by 55%, the training time by 70%, and the total execution time by 70%. Finally, to prove our training job scheduler did dynamically add or remove workers to training jobs according to the system loading, Fig. 10 shows the time distribution of a training job with a given number of workers when using DynamoML. As seen, in average, about 22% of the job execution time uses only 1 GPU, 63% of the time uses 2 or less GPUs. Only the reminding 37% of time using 3 or 4 GPUs. However, according to the execution time distribution shown in Fig. 11, DynamoML still significantly reduces the overall training time. With DynamoML, more than 60% of the training jobs finish within 5mins, and the longest execution time is 8mins. In contrast, the average execution time for native K8S is 13mins.

Inference Performance Analysis

Finally, we analyze the SLA violation of inference job from the running test workload shown in "Experiment Setup". Figure 12 shows the overall client response time distribution of the TF-serving inference job under different system settings. K8S+Sharing has the worst results, where some of the requests have response time over 1024ms. This is because it does not supports auto-scaling on the inference service. DynamoML performs the best with no requests with a response time over 128 ms, because it can preempt training jobs when necessary. One the other hand, although K8S+Scaling also supports auto-scaling on inference inference, but it cannot preempt training jobs. As a result, the amount of resources for inference jobs can be bounded by the training jobs. Therefore, the response time of K8S+Scaling is mostly between 128ms~512ms. Figure 13 further breaks down the SLA violation probability under different inference workloads which is controlled by the number of active clients. As expected, the violation probability increases under highly workload. Only DynamoML can be resilient to the workload, because its ability to obtain enough resources to satisfy the SLA requirements.

Evaluation of HYPREL

This section evaluates our *HYPREL* topology-aware scheduling algorithm. We developed an event-based simulator to evaluate *HYPREL*, because it is easier for us to implement all the compared scheduling algorithms and to have more consistent job execution time under different task placement and workload intensity. Comparing to other state-of-the-art topology-aware schedulers [25, 26], our assessment revealed the following highlights:

- *HYPREL* is superior to other solutions in terms of job completion time (JCT).
- *HYPREL* can make a trade-off between cluster resource utilization and job placement sensitivity.
- *HYPREL* consistently outperforms other topology-aware scheduling algorithms under a wide range of resource contentions.

Experimental Setup

The network topology considered in our evaluation is a typical hierarchical tree structure shown in Fig. 14. The 32-GPUs cluster is consisted of 2 nodes, where each node has 4 CPUs and each CPU has 4 sockets to control 4 GPUs.

The evaluation is based on the trace of the online arrival parallel computing job collected from the Ohio Supercomputing Center (OSC) [32]. The trace contains the submission time and the number of workers (i.e., tasks) of a set of submitted jobs. The maximum number of workers per job in the trace is less than 32. Therefore, we treat each worker as a GPU request for a job.

A total of 1540 jobs from the trace were used for the experiment. Each job is considered as distributed training job for a ResNet-50 or Mnist model in our simulation. Given a job with n workers, there could be $n!$ placement decisions which will be too time-consuming to collect their real execution time. Therefore, we relay on a parallel job performance model proposed in [33] to derive the job execution time in our simulation.

In the performance model, the total running time is divided into two parts: computing time and communication time. The computing time of a job is considered as constant. The communication time is modeled by a cost function $\tau(CC_{\text{job}})$, where CC_{job} is the average pairwise L1 distance (Manhattan distance) across all the communicating nodes of a job under a placement decision.

The cost function $\tau(CC_{\text{job}})$ is derived from a set of real experiment random samples. Each sample is the relationship between the communication time (τ) and average hop distance (CC_{job}) of a job running on a randomly selected placement decision. We apply linear regression to fit the experimental data to get the cost function of $\tau(CC_{\text{job}})$.

The responsibility of a scheduler is to periodically schedule the jobs waiting in the scheduling queue onto available GPU slots. As in practice, we assume the scheduler has no prior information about the job execution time or the future arrival jobs. Only the number of workers required by an arrival job is known by the scheduler. We compare HYPREL with the following representative job scheduling strategies for deep learning workloads:

Random: This is a scheduling algorithm without the awareness of network topology and task communication. Every time a job arrives. When a job arrives or departs, the scheduler simply selects the next job from the waiting queue and randomly assigns its tasks to available GPU slots. Its performance is considered as the baseline comparison result in our evaluation.

HiveD [25]: This is one of the recently proposed topology-aware scheduling algorithm for scheduling deep learning jobs. It is designed based on the well-known buddy memory allocation, which is an efficient heuristic allocation of best-fit allocation for minimizing resource fragmentation. However, memory allocation is a one-dimensional space allocation problem. Hence, HiveD annotates the resource (GPU slots) with network topology information to ensure the allocated resource slots of a job can be within close proximity. However, it does not consider the job ordering problem, so it could be applied to other job scheduling algorithm as well. In our evaluation, we assume HiveD is used with FIFO scheduling algorithm.

Topology-aware [26]: This is another recently proposed topology-aware scheduling algorithm for scheduling deep learning jobs. It formulates the task placement problem of a parallel job as a recursive graph partition problem. Whenever the tasks of job are scheduled across a network link, it is considered as a partition of the job. The main objective is to minimize the partition cost of a job while mapping its communication graph to the network topology. The cost of

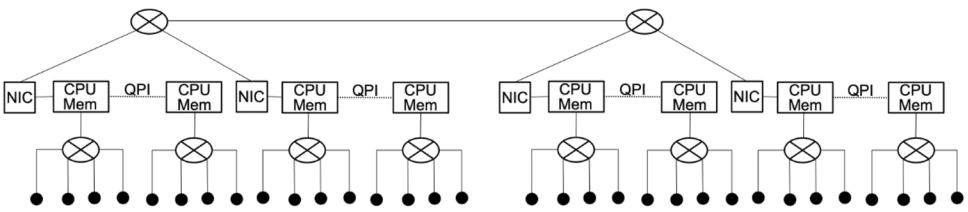


Fig. 14 Network used by the simulation

the partition is modelled by a cost function consisted of the cost of resource fragmentation, network communication and performance interference. It also delays jobs in the waiting queue when its partition cost is too high. The main drawback of the approach is that its result could be highly dependent on the setting of the cost function parameters and threshold of job delay. In our evaluation, we could only roughly tune the parameters according to our testing workloads.

Job Completion Time Comparison

Figure 15 shows the average job completion time (JCT) of the four compared scheduling algorithms. It is clear that HYPREL achieves the best result with 47%, 44% and 24% improvement on average JCT over Random, HiveD, and Topology-aware, respectively. As expected, Random has the worst result and the highest average JCT, because it has no awareness of the network topology and communication overhead. HYPREL is able to outperform the other two algorithms for the following reasons. For HiveD, it is aware of the network topology structure, but not consider the communication traffic of tasks or the scheduling order of jobs. As a result, it can greedily reduce communication overhead, but may not obtain better optimization result comparing to Topology-aware and HYPREL. For Topology-aware, it uses a sophisticated cost function to model communication cost, but the parameters in the cost function is hard to be tuned. In addition, Topology-aware does not reschedule job execution according to the job characteristics and only postpones jobs according to a delay threshold. In contrast, HYPREL uses hypergraph as an effective heuristic solution to capture network topology characteristics and schedules jobs with higher communication loading first. As a result, HYPREL is able to achieve lower communication overhead and lower JCT.

Figure 16 further plots our improvement on average JCT under varied workload contention. We increase the level of workload contention by increasing the job arrival rate of our workload trace. The y-axis value in Fig. 16 is the average JCT of a scheduling algorithm normalized to the result of our approach. Hence, a higher value implies greater performance improvement. As observed, HYPREL achieves the lowest average JCT across all the workload contention levels. The improvement over Random grows greater as the workload contention increases, because communication overhead and resource usage efficiency becomes more critical under higher workload contentions. On the other hand, the improvement of HYPREL over HiveD and Topology-aware roughly remains consistent across all contention levels. But noted, when the workload contention is too high, the JCT may be dominated by the queuing wait time. Hence, we can also observe that the differences between three topology-aware scheduling algorithms may become less under higher contention level.

Finally, we plot the accumulative distribution of individual job JCT in Fig. 17. The result shows that HYPREL not only reduces the average JCT of all the jobs, but also ensures shorter JCT for majority of the jobs. In addition, the longest JCT of the jobs is also significantly reduced. Therefore, HYPREL improves job execution performance from both overall system and individual jobs perspectives.

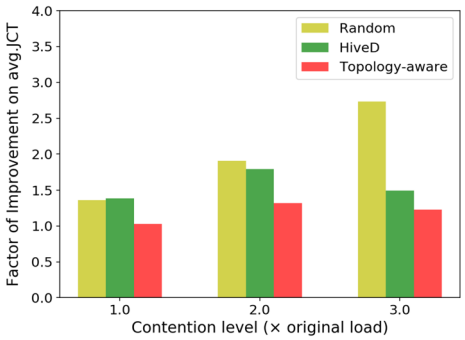


Fig. 16 HYPREL's improvement on the average JCT over the baselines under varying contention levels

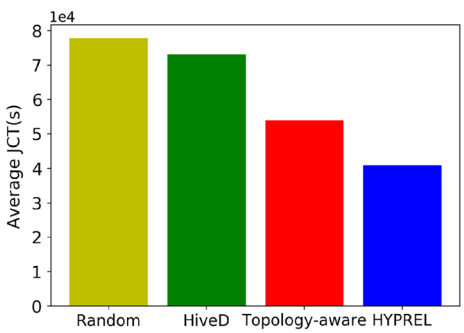


Fig. 15 Average Job Completion Time (JCT) comparison among job scheduling algorithms

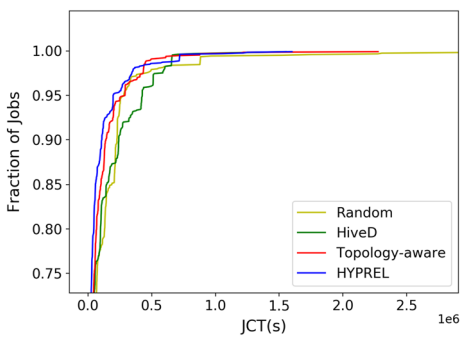


Fig. 17 Cumulative distribution comparison of individual job completion time among job scheduling algorithms

Related Work

In recent years, both research and industry have made great efforts to improve the performance of deep learning jobs in a GPU cluster by utilizing domain-specific knowledge. However, all these works target distributed training jobs alone. While our work addresses the ML pipeline workload, including modeling and inference jobs as well. Also, most of the proposed techniques require modifications to the deep learning frameworks, while our work can be transparent and general to DL applications.

Gandiva [22] is a scheduling framework developed by Microsoft. It supports many management techniques together to maximize system throughput for training jobs. It provides GPU sharing among jobs in both temporal and spatial domains. In the temporal domain, jobs run on GPU in an interleaved manner through the suspend-resume mechanism. In the spatial domain, jobs simply run simultaneously on a GPU at the same time, but jobs can be migrated to another GPU if performance degradation is detected. Gandiva also applies a scaling mechanism to jobs that self-declare to have good scalability. To minimize the overhead of their managing overhead like migration, suspend-resume, and scaling, Gandiva has to modify the deep learning frameworks, like Tensorflow and Pytorch.

Some of the work focuses on the scaling policy of distributed training jobs. For instance, Optimus [21] proposes a GPU resource scheduler to decide the proper resource amount and resource allocation according to a performance model of distributed training jobs. Similar to Optimus, DL2 [34] is a DL driven scheduler that aims to decide proper resource allocation to a distributed training job. However, the performance model proposed by DL2 is based on deep reinforcement learning. Tiresias [35] addresses the unpredictable job execution problem of model training to efficiently schedule and place DL jobs in GPU cluster for minimizing the job completion times (JCTs).

Other works focus on job placement problems to minimize the communication time of distributed training jobs. Most of these jobs meet the communication requirements of the job by imposing locality restrictions on the allocated resources. Still, too strict locality restrictions will prolong the waiting time of the job and cause fragmentation problem. For instance, [26] proposed a topology aware scheduling to decide the mapping between worker tasks and GPU slots based on the Hierarchical Static Mapping Dual Recursive Bi-partitioning algorithm. Tetris [36] is a multi-resource scheduler that uses packed tasks to avoid fragmentation. Feitelson [37] proposed a peer-based algorithm to reduce the fragmentation of gang scheduling jobs in supercomputers. The issue of topology-aware scheduling is not new, and it has been explored in diverse computing environments, where task dependency arises from data access or transfer between tasks. Just to name a few, for instance, serverless and streamming computing needs to address the dataflow problem among computing tasks [38–40], parallel computing needs to consider the communication patterns on interconnect networks [41–43]. Similar to our work, most of these approaches formulated their problems as graph partitioning and graph mapping problems. We chose the hypergraph algorithm in our approach, because the network topology considered in our problem has a tree structure. However, different from the previous hypergraph algorithms, we aim to minimize the resource fragmentation in our partition algorithm as well.

Finally, there is growing interest to explore GPU sharing technique for DL jobs. Spatial GPU sharing can suffer from unpredictable performance interference and resource contention. Therefore, temporal GPU sharing is more commonly adapted in practice. However, temporal sharing can be limited by the GPU memory size, and context switch overhead. Salus [44] takes advantage of the highly predictable and largely temporal usage memory pattern to provide a fine-grained sharing mechanism by switching jobs at the lowest memory usage point. Antman [23] further modifies the execution and scheduling engine of deep learn frameworks to support switching at the unit of operators (GPU kernels).

Conclusions

Deep learning workflow has become one of the primary workloads in data centers and GPU clusters. In this paper, we aim to optimize the application performance and system utilization through a set of runtime dynamic resource management techniques. GPU sharing increases the resource utilization of non-GPU bounded modeling jobs. Performance-driven auto-scaling guarantees the SLA requirement of inference jobs. Workload-aware scheduling and preemption utilizes the idle GPUs for reducing the execution time of training jobs. Finally, topology-aware scheduling minimize the communication overhead and maximize computing efficiency of distributed training jobs. While all these techniques have been discussed and used for different kinds of computing workload, we are one of the few work that really integrate and apply them together specifically for the ML pipeline workflow. Our system are built as extended operators on Kubernetes, and transparent to applications. Hence, our solution can be easily applied to general GPU clusters and DL workload. In the future, we plan to evaluate our system with more complex and real ML pipeline workload and implement the HYPREL scheduling algorithm in DynamoML for real testbed evaluation.

Funding This research is partially supported by the "Elastic Distributed Deep Learning Training Implementations and Optimizations Project" of National Tsing Hua University (NTHU), sponsored by the Ministry of Science and Technology, Taiwan, R.O.C. under Grant no. 108–2221-E-007-036.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Ethical Approval This article does not contain any studies with human participants performed by any of the authors.

References

1. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016;770–778. https://doi.org/10.1109/CVPR.2016.90
2. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Commun ACM. 2017;60(6):84.
3. Redmon J, Divvala SK, Girshick RB, Farhadi A. You only look once: Unified, real-time object detection. CoRR arXiv:abs/1506.02640 2015.
4. Xu D, Anguelov D, Jain A. Pointfusion: Deep sensor fusion for 3d bounding box estimation. CoRR 2017 arXiv:1711.10871.
5. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I. Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (editors) Advances in Neural Information Processing Systems, 2017;30, 5998–6008. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. Accessed 4 Dec 2017.
6. Devlin J, Chang M, Lee K, Toutanova K. BERT: pre-training of deep bidirectional transformers for language understanding. CoRR arXiv:abs/1810.04805 2018.
7. Yang Z, Dai Z, Yang Y, Carbonell J, Salakhutdinov RR, Le QV. Xlnet: Generalized autoregressive pretraining for language understanding. In: Wallach H, Larochelle H, Beygelzimer A, d' Alché-Buc F, Fox E, Garnett R. (eds.) Advances in Neural Information Processing Systems, 2019;32, 5753–5763. https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf
8. Liu Y, Ott M, Goyal N, Du J, Joshi M, Chen D, Levy O, Lewis M, Zettlemoyer L, Stoyanov V. Roberta: A robustly optimized BERT pretraining approach. CoRR 2019 arXiv:1907.11692.
9. Lan Z, Chen M, Goodman S, Gimpel K, Sharma P, Soricut R. ALBERT: A lite BERT for self-supervised learning of language representations. CoRR 2019 arXiv:1909.11942.
10. Tian Y, Pei K, Jana S, Ray B. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. CoRR 2017 arXiv:1708.08559.
11. Levine S, Pastor P, Krizhevsky A, Quillen D. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. CoRR 2016 arXiv:1603.02199.
12. Amodei D, Hernandez D. AI and Compute. https://openai.com/blog/ai-and-compute/ 2018.
13. He X, Pan J, Jin O, Xu T, Liu B, Xu T, Shi Y, Atallah A, Herbrich R, Bowers S, Candela JQn. Practical lessons from predicting clicks on ads at facebook. In: Proceedings of the Eighth International Workshop on Data Mining for Online Advertising, 2014:1.
14. Zoph B, Cubuk ED, Ghiasi G, Lin T, Shlens J, Le QV. Learning data augmentation strategies for object detection. CoRR arXiv:abs/1906.11172 2019.
15. Verma A, Pedrosa L, Korupolu M, Oppenheimer D, Tune E, Wilkes J. Large-scale cluster management at google with borg. In: EuroSys, 2015;1–17.
16. Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E. Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of Symposium on Cloud Computing 2013.
17. Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I. Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI, 2011:323–36.
18. Tumanov A, Zhu T, Park JW, Kozuch MA, Harchol-Balter M, Ganger GR. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: EuroSys, 2016:1–16.
19. Jalaparti V, Bodik P, Menache I, Rao S, Makarychev K, Caesar M. Network-aware scheduling for data-parallel jobs: Plan when you can. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015;407–20.
20. Tannenbaum T, Wright D, Miller K, Livny M. Condor - a distributed job scheduler. MIT Press; 2001.
21. Peng Y, Bao Y, Chen Y, Wu C, Guo C. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In: EuroSys, 2018;1–14.
22. Xiao W, Bhardwaj R, Ramjee R, Sivathanu M, Kwatra N, Han Z, Patel P, Peng X, Zhao H, Zhang Q, Yang F, Zhou L. Gandiva: Introspective cluster scheduling for deep learning. In: OSDI, 2018;595–610.
23. Xiao W, Ren S, Li Y, Zhang Y, Hou P, Li Z, Feng Y, Lin W, Jia Y. Antman: Dynamic scaling on GPU clusters for deep learning. In: OSDI, 2020;533–548.
24. Chiang M, Chou J. DynamoML: Dynamic Resource Management Operators for Machine Learning Workloads. In: Proceedings of the 11th International Conference on Cloud Computing and Services Science - CLOSER,, 2021;122–132.
25. Zhao H, Han Z, Yang Z, Zhang Q, Yang F, Zhou L, Yang M, Lau FCM, Wang Y, Xiong Y, Wang B. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020;515–532. https://www.usenix.org/conference/osdi20/presentation/zhao-hanyu. Accessed 4 Nov 2020.
26. Amaral M, Polo J, Carrera D, Seelam S, Steinder M. Topology-aware gpu scheduling for learning workloads in cloud environments. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017;1–12.
27. Yeh T-A, Chen H-H, Chou J. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, 2020;173–84.
28. Lin C-Y, Yeh T-A, Chou J. DRAGON: A dynamic scheduling and scaling controller for managing distributed deep learning jobs in kubernetes cluster. In: International Conference on Cloud Computing and Services Science (CLOSER), 2019;569–577.
29. Shmoys D, Hall L. Approximation schemes for constrained scheduling problems. In: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, pp. 134–139. IEEE Computer Society, Los Alamitos, CA, USA (1989). https://doi.org/10.1109/SFCS.1989.63468. https://doi.ieeecomputersociety.org/10.1109/SFCS.1989.63468.

30. Schlag S. High-quality hypergraph partitioning. PhD thesis, Karlsruher Institut für Technologie (KIT). https://doi.org/10.5445/IR/1000105953. 46.12.02; LK 01 2020.
31. Akhremtsev Y, Heuer T, Sanders P, Schlag S. Engineering a direct k-way hypergraph partitioning algorithm. In: ALENEX 2017.
32. Center OS. Ohio Supercomputer Center 1987. http://osc.edu/ark:/19495/f5s1ph73.
33. Meng J, McCauley S, Kaplan F, Leung VJ, Coskun AK. Simulation and optimization of hpc job allocation for jointly reducing communication and cooling costs. Sustain Comput Inf Syst. 2015;6:48–57.
34. Peng Y, Bao Y, Chen Y, Wu C, Meng C, Lin W. DL2: A deep learning-driven scheduler for deep learning clusters. IEEE Transact Parall Distribut Syst. 2021;32(08):1947–60.
35. Gu J, Chowdhury M, Shin KG, Zhu Y, Jeon M, Qian J, Liu H, Guo C. Tiresias: A GPU cluster manager for distributed deep learning. In: NSDI, 2019;485–500.
36. Grandl R, Ananthanarayanan G, Kandula S, Rao S, Akella A. Multi-resource packing for cluster schedulers. SIGCOMM Comput Commun Rev. 2014;44(4):455–66. https://doi.org/10.1145/2740070.2626334.
37. Feitelson DG. Packing schemes for gang scheduling. In: Feitelson DG, Rudolph L, editors. Job Scheduling Strategies for Parallel Processing. Berlin, Heidelberg: Springer; 1996. p. 89–110.
38. Palma GD, Giallorenzo S, Mauro J, Trentin M, Zavattaro G. Topology-aware Serverless Function-Execution Scheduling 2022
39. Zheng S, Liu B, Lin W, Ye X, Li K. A package-aware scheduling strategy for edge serverless functions based on multi-stage optimization. Fut Generat Comput Syst. 2023;144:105–16. https://doi.org/10.1016/j.future.2023.02.013.
40. Li B, Sun D, Chau VL, Buyya R. A topology-aware scheduling strategy for distributed stream computing system. In: Xiang W, Han F, Phan TK, editors. Broadband communications, networks, and systems. Cham: Springer; 2022. p. 132–47.
41. Wang Y-C, Chou J, Chung I-H. A deep reinforcement learning method for solving task mapping problems with dynamic traffic on parallel systems. In: The International Conference on High Performance Computing in Asia-Pacific Region. HPC Asia 2021, pp. 1–0. Association for Computing Machinery, New York, NY, USA 2021. https://doi.org/10.1145/3432261.3432262.
42. Bhatele A, Jain N, Isaacs KE, Buch R, Gamblin T, Langer SH, Kale LV. Optimizing the performance of parallel applications on a 5d torus via task mapping. In: 2014 21st International Conference on High Performance Computing (HiPC), 2014;1–10. https://doi.org/10.1109/HiPC.2014.7116706.
43. Deveci M, Rajamanickam S, Leung VJ, Pedretti K, Olivier SL, Bunde DP, Catalyurek UV, Devine K. Exploiting geometric partitioning in task mapping for parallel computers. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014;27–36. https://doi.org/10.1109/IPDPS.2014.15.
44. Yu P, Chowdhury M. Salus: Fine-grained GPU sharing primitives for deep learning applications. CoRR arXiv:abs/1902.04610 2019.