TapFinger: Task Placement and Fine-Grained Resource Allocation for Edge Machine Learning

Yihong Li*, Tianyu Zeng*, Xiaoxi Zhang*§, Jingpu Duan†, Chuan Wu‡
*School of Computer Science and Engineering, Sun Yat-sen University
†Department of Communications, Peng Cheng Laboratory
‡Department of Computer Science, The University of Hong Kong
Email:{liyh253, zengty}@mail2.sysu.edu.cn, zhangxx89@mail.sysu.edu.cn, duanjp@pcl.ac.cn, cwu@cs.hku.hk

***Abstract*—Machine learning (ML) tasks are one of the major workloads in today's edge computing networks. Existing edge-cloud schedulers allocate the requested amounts of resources to each task, falling short of best utilizing the limited edge resources flexibly for ML task performance optimization. This paper proposes *TapFinger*, a distributed scheduler that minimizes the total completion time of ML tasks in a multi-cluster edge network, through co-optimizing task placement and fine-grained multi-resource allocation. To learn the tasks' uncertain resource sensitivity and enable distributed online scheduling, we adopt multi-agent reinforcement learning (MARL), and propose several techniques to make it efficient for our ML-task resource allocation. First, *TapFinger* uses a heterogeneous graph attention network as the MARL backbone to abstract inter-related state features into more learnable environmental patterns. Second, the actor network is augmented through a tailored task selection phase, which decomposes the actions and encodes the optimization constraints. Third, to mitigate decision conflicts among agents, we novelly combine Bayes' theorem and masking schemes to facilitate our MARL model training. Extensive experiments using synthetic and test-bed ML task traces show that *TapFinger* can achieve up to 28.6% reduction in the average task completion time and improve resource efficiency as compared to state-of-the-art resource schedulers.**

I. Introduction

Edge computing is a distributed computing paradigm that extends cloud capabilities to the edge for better quality of service (QoS) and data privacy protection. Edge-based ML applications, ranging from traffic prediction to production workflow monitoring, commonly process online data streams generated on the edge [1]. Due to resource limitation of edge devices, these ML tasks have been deployed in edge clusters [1], [2], e.g., NVIDIA EGX [3] and Microsoft Azure Edge [4]. Managed by orchestration tools, they can be running on sufficient CPUs and GPUs, as well as customized software toolkits and network interface cards, e.g., for encrypted IoT sensor data [3]. However, the resources at the edge clusters are still limited. At the core of optimizing the QoS of edge-based ML applications is efficient resource utilization while learning the needed ML models timely.

This work was supported by NSFC grants 62102460 and 61902171, grants from Hong Kong RGC under the contracts HKU 17204619 and 17208920, and grant 202201011392 under the Guangzhou Science and Technology Plan Project.

§Xiaoxi Zhang is the corresponding author.

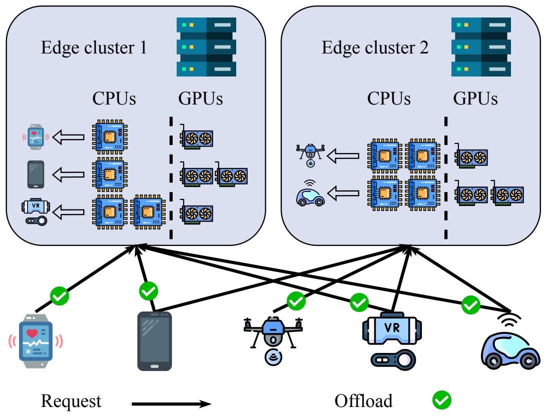


Fig. 1: Fine-grained allocation in a multi-cluster edge network.

With various models and datasets, ML training and inference tasks have uncertain and diverse performance [5]–[7], making it hard to achieve optimized resource efficiency. Practical schedulers such as YARN [8], Kubernetes [9], and KubeEdge [10] generally adopt pre-set rules for resource allocation. Those policies rely on accurate resource estimation of the tasks, while the resource demands of ML tasks are typically elastic and uncertain (e.g., the amount of time needed for model convergence), accommodating various performance-resource tradeoffs [11]. Learning-based cloud-edge schedulers have been proposed to address this uncertainty [11]–[15]. However, they cannot generalize to our scenario, where more complex decision dependencies need to be better encoded, e.g., for further addressing the high-dimensionality of problem inputs and decision variables. In fact, fine-grained resource allocation and strategic task placement are exceptionally important to maximize the aggregate performance of edge ML tasks. As illustrated in Fig. 1, due to the data intensity and low latency requirement, mobile edge devices need to offload their ML tasks to the "right" edge clusters for achieving the model convergence in the minimum time. Besides, different resources, e.g., CPU and GPU, can affect task performance, necessitating multi-resource allocation schemes. Considering the complexity of dynamic network connectivity, job interference [16], and multi-resource contention, we ask: how to design a scalable, fine-grained, and far-sighted resource scheduler customized for edge ML?

To realize this, we propose a distributed scheduler to *jointly optimize task placement and fine-grained multi-resource allocation across edge clusters*, with a goal of minimizing total completion time of ML tasks. To achieve this goal, we face the following fundamental challenges.

Fine-grained resource allocation. Different from existing edge-cloud schedulers which allocate the requested resource amounts to each task [6], [12], fine-grained resource provisioning [7] according to demand and supply can achieve better resource efficiency and application QoS. The challenge is then to predict the performance of each concurrently running task and strategically choose the best resource amounts from a huge solution space for maximizing the aggregate task performance.

Uncertain impact of different resources. Most ML task schedulers allocate a single type of resource, e.g., GPU, in the cloud [7], [14], [17] or edge settings [2], failing to capture the effects of multiple types of resources on the task performance. A multi-resource allocation problem is typically NP-hard even with perfect knowledge of the problem input [18], which is more challenging when the task performance is unknown given the resources. Reinforcement learning (RL) can efficiently deal with uncertainty [19]–[21] but is still under-explored for online combinatorial problems with complex constraints.

Distributed scheduling across edge clusters. Practical edge-cloud clusters are usually managed by a central coordinator [8], [9]. However, centralized scheduling processes the global information over multiple geographical regions and often suffers from poor scalability. A decentralized approach is therefore preferable to reduce the decision space and enable better system reliability. Nonetheless, decentralized scheduling may lead to sub-optimal task performance if self-optimizing decisions in each edge cluster independently. Effective interactions among the distributed schedulers are crucial for optimizing global resource efficiency in the entire edge network.

To address these challenges, *TapFinger* adopts a multi-agent reinforcement learning (MARL) approach. Compared to rule-based distributed optimization, it can better generalize to unseen task characteristics and different edge networks. To achieve efficient MARL for our distributed resource scheduling, we make the following technical contributions:

First, we propose a state abstraction technique to enable efficient information interactions among distributed agents. Conventional MARL approaches can be sub-optimal due to the partial observability of each agent. We use a heterogeneous graph attention network (HAN) [22] to encode rich semantic information of different edge components and their dependencies. The local observations are then passed as messages among the agents and provide them with a global view. The raw states are then mapped into a compressed space and efficiently improve the learning ability of the agents.

Second, we propose several techniques to augment our MARL actor network and the training method. To decompose our actions on task placement and resource allocation, we design a task selection phase via a pointer network module [23], [24], inspired by natural language processing (NLP) techniques. We then construct a conflict resolution module based on Bayes' theorem to coordinate different agents' decisions. In addition, we use masking schemes to encode our constraints and filter out the gradient propagation that is irrelevant to the conflicts, which effectively speed up the MARL convergence.

Third, we conduct extensive experiments on synthetic and test-bed ML task traces. We observe a significant reduction in terms of the average task completion time, compared to representative scheduling algorithms. The experiments also show that our algorithm can capture the diverse resource sensitivities of different types of ML tasks and effectively improve resource efficiency. *TapFinger* also demonstrates its scalability as it maintains significant superiority with varying numbers of edge clusters, system span, and task arrival rates.

II. Related Work

Resource scheduling for DL workloads. Although general-purpose task scheduling algorithms such as Dominant Resource Fairness [25], Tetris [26], and their improved variants [27]–[29] have been extensively studied, the strategies tailored to machine learning (ML) workloads remain premature [30]. Peng et al. [5] design Optimus to schedule distributed DL training tasks by training speed predictions. To exploit the cyclic patterns in DL tasks, Xiao et al. [17] implement efficient time-slicing and profiling-driven introspection. Tiresias [6] combines least-attained service scheduling and multilevel feedback queues to design a preemptive scheduling algorithm. A few recent works achieve elastic GPU cluster scheduling for DL tasks using novel profiling methods [2], [31] or new performance metrics [7], [32]. In general, these are rule-based algorithms that rely on accurate estimations of task characteristics. This paper, instead, aims to enable a self-optimization framework without hand-crafting prediction models for capturing uncertain task performance.

Task scheduling with deep reinforcement learning. Deep Reinforcement learning (DRL) has demonstrated its effectiveness in online decision making including task scheduling and resource allocation [20], [21]. Several works [20], [33], [34] leverage the pointer network mechanism [23] for task placement. However, they have not considered ML workloads or fine-grained resource allocation. In [11], a DRL method that fuses rule-based policies is proposed for ML task placement. Peng et al. [13] propose a trace-driven task scheduler for DL clusters using offline supervised learning to boost the DRL training. Recent works have also extended DRL for resource allocation in edge clusters. For instance, Tuli et al. [12] decide the allocation of edge-cloud resources by regarding the distributed resources as a large number of independent hosts. Different tasks and hosts are divided among multiple agents for DRL training. These works adopt single-agent DRL and do not consider fine-grained resource allocation for ML tasks or jointly optimize task placement and resource allocation.

GNN-based DRL methods for task scheduling. Beyond applying DRL approaches with standard formulation of the environmental states and actions, there is a trend of designing state abstraction to boost DRL training. Graph neural network (GNN) is a promising model that can encode the dependencies across state features. We have identified that two recent works are similar to ours methodology-wise. First, Zhao et al. [14] adopt GNN to encode cluster topology and server configurations, motivated by the interference of co-located DL tasks. They design an MARL algorithm for scheduling among GPU clusters. Each agent decides either to serve the task or to forward it to other agents. Second, Han et al. [15] propose a two-timescale scheduler for Kubernetes-oriented edge-cloud clusters. They introduce GNN-base DRL for request dispatch and MARL scheduling for service orchestration. However, the above works do not consider instant fine-grained resource allocation, hence being detrimental to resource efficiency.

III. Problem Description

In what follows, we describe our system infrastructure and formulate the task placement and resource allocation problem. We define $[X] \triangleq \{1, 2, \cdots, X\}$ in our description.

A. Edge Clusters

We consider that N geo-distributed edge clusters provide low-latency machine learning services to local edge devices. Today's edge clusters are equipped with sufficient computational ability to train and run deep neural network (NN) models [3], [4]. For compliance requirements and restrictive data policies, ML training and inference tasks may be required to run on certified edge clusters. We consider a total of R types of computation resources, e.g., CPU, GPU, and memory, offered by each edge cluster to perform ML tasks. The clusters are not necessarily homogeneous in real-world settings. Each resource $r \in [R]$ has a distinct smallest unit δ_r that can be allocated, and the resource capacity $C_{r,n}$ is equal to the total amount of δ_r available in cluster n. The tasks that come online may occupy resources for various task durations. Therefore, the maximum amount of each type-r resource at n that can be allocated at t, defined to be $B_{r,n,t}$, can be time-varying.

Edge computing networks are distributed in nature, and provides basic data exchange functions via low-cost network connections among edge clusters [35], unlike the stable network condition and sufficient bandwidth in cloud datacenters. It is then impracticable to conduct a large number of distributed parallel training and inference processes across multiple edge clusters [36]. Therefore, we consider in-cluster GPU-level parallel training and inference for better QoS of user applications. Further, to fully utilize the resources of the entire network, we allow edge devices to simultaneously send offloading requests to all accessible edge clusters, as in Fig. 1.

B. Task Arrivals, Scheduling Orders, and Placement

Since we aim to optimize the aggregate QoS over any exogenously determined workload, we consider a total of T timesteps, each of which is evenly partitioned based on the system requirements[1]. We consider a total of J ML tasks that arrive at arbitrary times in $t \in [T]$. For instance, we have image classification training tasks over ConvNets, speech recognition inference tasks using transformers, etc. In edge computing scenarios, tasks generated locally on resource-scarce edge devices need to be offloaded to an edge cluster. Due to network limitations, the edge devices may only connect to a subset of the edge clusters. We define $\mathcal{D}_j$ to be the set of edge clusters that task j can be offloaded to and $\mathcal{Q}_{n,t}$ to be the set of tasks in the queue of cluster n at t. We allow edge devices to send requests to all the edge clusters in $\mathcal{D}_j$ to avoid the long wait for being scheduled, but restrict that one task can only be finally offloaded to one edge cluster, as shown in Fig. 3.

The task requests continuously arrive from the edge devices into the queue of each connectable edge cluster. The scheduler of each cluster then independently decides which task $j \in \mathcal{Q}_{n,t}$ to serve when t starts. Since multiple clusters may choose the same task at the same time, a coordinator is needed to choose one of the clusters to serve the task. We define a binary variable $y_{j,n,t}$ to denote whether j is scheduled by cluster n in timestep t, and require $\sum_{j \in \mathcal{Q}_{n,t}} y_{j,n,t} \leq 1$, i.e., the task is only scheduled to at most one cluster at each timestep. Note that in modern container orchestration tools like Kubernetes, the scheduler first collects all feasible nodes in the cluster, then scores these candidates according to a series of factors, including hardware/software constraints, data locality requirements, etc., and finally binds the node with the highest score to the pod [9]. Multi-thread scheduling can schedule tasks concurrently but still needs a transaction commit process. In essence, this is a sequential operation for one scheduler. Therefore, it is consistent with our problem setting where each scheduler only schedules at most one task at any timestep instead of a batch of independent tasks.

C. Multi-Resource Allocation

Once selecting a task $j \in \mathcal{Q}_{n,t}$, the scheduler needs to decide its resource allocation. As shown in Fig. 2, we identify that the completion time of a representative ML task has various sensitivities under different combinations of CPU and GPU resources. For instance, the completion time first decreases with the number of CPU cores and then becomes rather flat. Improper resource allocation, e.g., 1 CPU and 8 GPUs, may lead to a rebound in the completion time. Motivated by our observations, we consider that the scheduler in each cluster decides the allocation of resources. We define an integer decision variable $x_{j,n}$ to represent the amount of type-r resource in cluster n allocated to j. Let $\hat{t}_j$, t^*_j, and ϕ_j denote the arrival time, start time of execution, and completion time of task j, respectively, where t^*_j is decided by our scheduling decisions and ϕ_j is affected by both t^*_j and our resource allocation. Since the tasks may arrive at different times and occupy the resources for various periods of time, we have a time-coupling capacity constraint $\sum_{j:t \in [t^*_j, \hat{t}_j + \phi_j)} x_{j,r,n} \leq C_{r,n}$ for each r, n, and t. In addition, we define $b_{j,r}$ as the minimum amount of type-r resource required by task j. Thus, each $x_{j,r,n}$ has to be at least $b_{j,r}$ if positive.

[1]We assume this time-slotted fashion for ease of implementation. In fact, our scheduler can be adjusted to execute task scheduling at non-uniform timesteps, e.g., upon the arrival of each task.

Here we assume that our resource allocation cannot be modified once decided. Although preemptive schedulers may achieve better resource utilization and reduce completion time [6], [7], they are not widely implemented in real-world ML clusters [37]. One of the main reasons is that pausing and resuming ML tasks using the checkpointing mechanism frequently incurs overhead, increases system complexity, and introduces potential failures, especially for training tasks [38]. Distributed DL training frameworks like PyTorch and Horovod provide elastic training functions that allow users to scale the number of workers during the training process. However, that might need dynamically adjusted hyper-parameters e.g., learning rate and batch sizes [32], which requires extra coding from task owners and still incurs unpredictable system overhead. These uncertainties prevent reproducing the training convergence results and contradict the original intention of using edge computing to guarantee the QoS of ML tasks.

We now formalize our joint task placement and multi-resource allocation problem. The goal is to minimize the total completion time of all tasks that arrive in $[T]$, while satisfying the resource capacity constraints and the tasks' resource requirements. In practice, a training task is completed when a certain accuracy or convergence is met. The completion time of an inference task is also hard to predict due to network instability and fluctuations in available resources [39]. Therefore, the completion time of each task ($\phi_j(\cdot)$) is unknown a priori and affected by our task placement $\mathbf{y}$ and resource allocation $\mathbf{x}$ decisions for all the tasks that co-exist with j. Finally, we formulate the optimization problem as follows.

$$\underset{\mathbf{x},\mathbf{y}}{\text{Minimize}} \quad \sum_{j=0}^{J-1} \phi_j(\mathbf{x},\mathbf{y}) \tag{1}$$

$$\text{S.t.} \sum_{j\in\mathcal{Q}_{n,t}} y_{j,n,t} \le 1, \quad \forall n,t \tag{2}$$

$$\sum_{n\in\mathcal{D}_j}\sum_{t\in[T]} y_{j,n,t} \le 1, \quad \forall j \tag{3}$$

$$y_{j,n,t} \in \{0,1\}, \quad \forall j,n,t \tag{4}$$

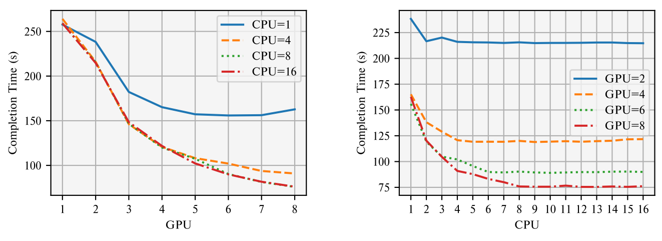
$$\sum_{j:t\in[t_j^*,\hat{t}_j+\phi_j)} x_{j,r,n} \le C_{r,n}, \quad \forall r,n,t \tag{5}$$

$$t_j^* = \max_{n,t}\left(t\cdot y_{j,n,t}\right), \quad \forall j \tag{6}$$

$$x_{j,r,n} \le \max_t\left(y_{j,n,t}C_{r,n}\right), \quad \forall j,r,n \tag{7}$$

$$x_{j,r,n} \in \{0\}\cup\{\mathbb{Z}\cap[b_{j,r},C_{r,n}]\}, \forall j,r,n \tag{8}$$

Here, since the task scheduling decision $y_{j,n,t}$ is binary (constraint (4)), (2) ensures that at most one task can be added from the queue onto each cluster n in each timestep t. For each task j, $y_{j,n,t}$ is only positive ($=1$) at the timestep when j is scheduled by n and remains zero in other timesteps, indicating that *task j will be processed by cluster n until it completes* (inequality (3)). Our definition of $y_{j,n,t}$ requires that our placement decision for task j cannot be modified and yields a simpler way to model the resource allocation problem, although other modeling choices can also work. Further, (5) requires that each type of resource occupied by all the alive tasks $\{j\,|\,t\in[t_j^*,\hat{t}_j+\phi_j)\}$ in any time t cannot exceed the corresponding capacity. Equation (6) defines the start time that j is scheduled, which is the only time that $y_{j,n,t}$ is positive for task j. Finally, (7) and (8) define the time-varying feasible region of $\mathbf{x}$ in each time t.



(a) Varying with GPU. (b) Varying with CPU cores.

Fig. 2: Testbed results on the completion time of training a language modeling task using a transformer network [40] by varying the combination of allocated resources.

Our formulation (1)–(8) mathematically models the dependencies between the decisions of task placement and resource allocation across co-existing tasks. It provides insights to our algorithm design, e.g., its impact factors that need to be observed as environmental states and the feasible region of our actions. However, this formulation captures a centralized optimization and needs to be factorized for distributed scheduling based on MARL, which we will elaborate in Section IV.

IV. Algorithm

We next walk through the designs of our proposed algorithm that jointly optimizes the task placement and resource allocation for ML tasks across multiple edge clusters.

A. Algorithm Overview

Driven by the complexity of co-optimizing the decision variables in our offline optimization (1)–(8), we propose to enable a distributed optimization where $\mathbf{x}$ and $\mathbf{y}$ can be first independently and preliminarily decided by each cluster, and then coordinated by a central coordinator residing in one of the edge clusters. In addition, the unknown completion time $\phi_j(\mathbf{x},\mathbf{y})$ can be optimized through trials and errors if we can learn the statistical patterns of tasks' resource sensitivities and their dependencies. These design intuitions fit into the basic idea of MARL. Our goal is then to push the limit of MARL towards solving dependent decisions under time-varying constraints, which drives the design of *TapFinger*. Fig. 3 shows our system components, illustrating the task placement and multi-resource allocation based on the actor-critic architecture [19] across at least two edge clusters. We introduce several techniques to the MARL algorithm, e.g., state representation through GNN, action decomposition, and loss function design based on masking schemes. We brief the concepts of state, action, and reward in this section. The technical specifics of our design are elaborated in Sections IV-B and IV-C.

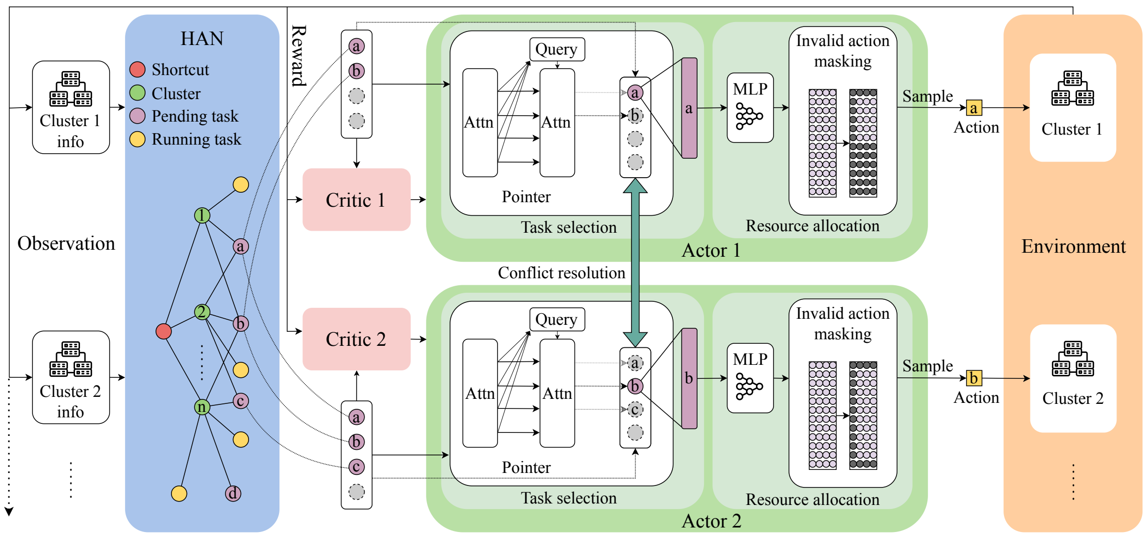


Fig. 3: The design components of *TapFinger*.

State space. Intuitively, the dynamics of task arrivals, connectivity between devices and clusters, and resource sensitivity of each task will all affect our objective function and constraints in (5)–(8). The challenge is that it is inefficient either to stack all these factors across the entire edge network into a global state or to split the state space into sub-spaces for each individual agent. We adopt a heterogeneous graph attention (HAN) network [22] to encode our features and their interrelation semantics. The output of HAN serves as the abstracted states for each agent, which compress the raw states and deliver learnable global environmental patterns. This design will be detailed in Section IV-B.

Action space. Even using MARL, the action space of each agent will be huge if directly concatenating our decisions $\mathbf{x}_n$ and $\mathbf{y}_n$ as a vector with a dimension of $(|Q_{n,t}| \times \prod_{r=1}^{R} C_{r,n})$. To reduce this space, we design an action decomposition technique that combines the pointer network and decision conflict resolution. Each agent n takes the HAN embedding of the pending tasks corresponding to cluster n as the input of its actor network, and outputs both the task selection and resource allocation actions. For task selection, we use transformer layers to further encode the HAN embedding and introduce a pointer mechanism [23] to decode the task selection actions. Then a conflict resolution module takes control to resolve task selection conflicts. Finally, the HAN embedding of the selected task is input into our invalid action masking module which encodes the constraints and outputs the final resource allocation. Technical details are shown in Section IV-C.

Reward. Note that the total completion time equals the sum of the total number of alive tasks over all timesteps, i.e., $\sum_j \phi_j(\mathbf{x}, \mathbf{y}) = \sum_t^T R_t(\mathbf{x}, \mathbf{y})$. $R_t(\mathbf{x}, \mathbf{y})$ is the total number of all tasks in the system in t including the ones that are waiting or running at any edge cluster. We define $r_{n,t} = -R_{n,t}$ as the reward for n in timestep t. Similar to R_t, $R_{n,t}$ is the number of tasks associated with the edge cluster n in timestep t.

B. State Representation via GNN Design

Reviewing the structure of our optimization problem, a good scheduler should account for the workload on each cluster and their currently allowed allocation choices, i.e., the minimum required and maximum available resource amounts based on (5)–(8). It should also adapt with learned *resource sensitivity* of different tasks which determine $\phi_j(\mathbf{x}, \mathbf{y})$. A key insight is that the running tasks can continuously provide resource sensitivity information, and the waiting tasks indicate their increased completion times and future contention. Therefore, each edge cluster needs to constantly monitor resource utilization, running tasks, waiting tasks, and newly submitted tasks. To speed up the action searching, we define a set of pending tasks $\mathcal{P}_{n,t}$ with a fixed size as the input of our task selection phase. Each scheduler n appends up to $|\mathcal{P}_{n,t}|$ tasks that have the earliest arrival times from the queue (i.e., $\in \mathcal{Q}_{n,t}$) to its pending set. First, we define the features of raw states for the agent n, namely $s_{n,t} = \{\mathcal{O}_{n,t}, \mathcal{P}_{n,t}, e_{n,t}, q_{n,t}\}$, which serves as the input of our HAN and is formalized as follows.

- **Running** task feature set $\mathcal{O}_{n,t}$ consists of entities $o_{j,n,t} \in \mathbb{R}^{R+J+1}$, each of which is the concatenation of the resource allocation, task type and elapsed time for a running task in n and t. These running tasks not only provide real-time feedback on resource sensitivity but also indicate the future available resources that they will release.
- **Pending** task feature set $\mathcal{P}_{n,t}$ contains entities $p_{j,n,t} \in \mathbb{R}^{R+J}$, each of which encodes the concatenation of the minimum resource requirement and the type of task j. We pad the pending set with dummy entities p_0, differentiated from the real entities by an extra binary digit for indication.
- **Resource** feature vector $e_{n,t} \in \mathbb{Z}^R$ represents the available resources of cluster n in timestep t.
- **Queuing** task feature $q_{n,t}$ is a scalar of the remaining number of tasks in the queue excluding those in the pending set. It reflects the current system workload and thus has a

strong correlation with the objective value.

In our MARL framework, if each agent n can only observe the inner-cluster features $\{\mathcal{O}_{n,t}, \mathcal{P}_{n,t}, e_{n,t}, q_{n,t}\}$, they will be more prone to being trapped in their local optima. A global representation of features in all clusters is thus needed. However, we cannot simply stack $s_{n,t}$ over all n into a global state matrix and feed it to each agent's model. The main drawbacks are: 1) the graph structure of the states will be lost; 2) the state space is not compact due to duplicate information such as the tasks shared in multiple $\mathcal{P}_{n,t}$ and $\mathcal{Q}_{n,t}$. Our solution instead embeds the entire graph into a neural network and enables iterative state interaction across clusters, with inter-cluster information reinforced over time based on importance.

Since HAN is designed to embed heterogeneous nodes, link relations, and their semantics [22], it fits well with our state representations. As shown in Fig. 3, we design our HAN to be a graph $\mathcal{G}_t(\mathcal{V}_t, \mathcal{E}_t)$, where the node set $\mathcal{V}_t$ consists of $\mathcal{O}_t$, $\mathcal{P}_t$, cluster nodes $\mathcal{N}$, and a shortcut node. We abuse the notation of the task and cluster indices to denote the corresponding nodes as well. The edges in $\mathcal{E}_t$ are defined as follows. A running task node $j \in \mathcal{O}_t$ connects with a cluster node $n \in \mathcal{N}$ only if task j is running on cluster n. Analogously, a pending task node $j \in \mathcal{P}_t$ connects with the cluster node n when task j is in the pending set, i.e., $j \in \mathcal{P}_{n,t}$. Each j can connect to multiple cluster nodes since they can be in the queues of multiple n, and we construct each $\mathcal{P}_{n,t}$ by dequeuing tasks in $\mathcal{Q}_{n,t}$ until $\mathcal{P}_{n,t}$ is full or $\mathcal{Q}_{n,t}$ is empty. The information propagation of our HAN follows the update steps in [22] by passing the features as messages from the neighbors to each node $u \in \mathcal{V}_t$ and aggregating them with the features of u using a two-level attention network in a configurable number of interactions. To speed up the message propagation between nodes that are far from each other, we add a shortcut node that connects all the cluster nodes to better adapt to a large number of agents. The propagation model of our HAN is formalized below.

$$\begin{aligned}
G_t^{\text{shortcut},(0)} &= \mathbf{0} \\
G_t^{N,(0)} &= \cup_{n=1}^{N}\{(e_{n,t}, q_{n,t})\} \\
G_t^{\mathcal{P},(0)} &= \cup_{n=1}^{N}\mathcal{P}_{n,t} \\
G_t^{\mathcal{O},(0)} &= \cup_{n=1}^{N}\mathcal{O}_{n,t}
\end{aligned} \tag{9}$$

We concatenate $e_{n,t}$ and $q_{n,t}$ for all the clusters as the input features of the cluster nodes, and assign $p_{j,t}$ and $o_{j,t}$ to every pending and running tasks node respectively. We denote the initial global state input as $G_t^{(0)} = \{G_t^{\text{shortcut},(0)}, G_t^{N,(0)}, G_t^{\mathcal{P},(0)}, G_t^{\mathcal{O},(0)}\}$, as in (9). The node embedding is propagated in each layer l, i.e., $G_t^{(l)} = g(G_t^{(l-1)})$, where $g(\cdot)$ represents the two-level attention network aggregating the features of each node with its neighbors. After L layers of graph message passing, we get the final graph embedding $G_t^{(L-1)}$. We then map $G_t^{\mathcal{P},(L-1)}$ to the corresponding agents as the input of their actor networks.

C. Action Decomposition and Constraint Encoding

Now we formalize our design of the NN architecture and functions for our actor network. The basic idea is to decompose the task placement and resource allocation decisions since naively concatenating them together yields a much larger action space. Instead, we leverage their dependencies, i.e., we only need to allocate resources to the selected task (in (7)). The challenges are using NN modules to encode such dependencies and enable them to learn our optimization constraints. We finally propose our training methods that can boost the agents to avoid conflicts and the decisions that violate constraints.

Pointer module. We design a pointer network inspired by the architecture of sequence-to-sequence NNs for natural language processing. The key insight is that we both need to solve problems in that the vocabulary of the output sequence will change with the length of the input sequence. Combinatorial problems like the traveling salesman problem usually have this trait [23], [24]. In our task selection phase, although we can fix the length of the input sequence with dummy entities, we cannot fix the vocabulary length since the size of the pending set changes with time, and scheduling a dummy entity is illegal. So we implement a pointer network for decoding task selection actions by a functionally simplified attention layer. We formulate our pointer module for agent n as follows.

$$\begin{aligned}
\bar{h}_{n,t} &= \frac{1}{|\mathcal{P}_{n,t}|}\sum_{j:p_{j,n,t}\in\mathcal{P}_{n,t}}^{|\mathcal{P}_{n,t}|} h_{j,n,t} \\
\hat{u}_{j,n,t} &= \begin{cases} -\infty, & \forall j : p_{j,n,t} = p_0 \\ v^T \tanh(W_1 h_{j,n,t} + W_2 \bar{h}_{n,t}), & \text{otherwise.} \end{cases} \\
\hat{z}_{j,n,t} &= \text{softmax}(\hat{u}_{j,n,t})
\end{aligned} \tag{10}$$

Here, $h_{j,n,t}$ is the output of task j of the transformer encoder. W_1, W_2, and v^T are model parameters. We design attention scores of the model, $\hat{z}_{j,n,t}$, to represent the probability of scheduling a pending task i, shown as the output of the second *Attn* in Fig. 3. We then sample the initial task selection action for the scheduler from the probability distribution constructed by $\hat{z}_{j,n,t}$, i.e., $z_{j,n,t} \sim \{\hat{z}_{j,n,t}\}_{j\in\mathcal{P}_{n,t}}$. E.g., the task labeled by a is sampled by agent 1 in Fig. 3. We mask $\hat{u}_{j,n,t}$ of the dummy entities as $-\infty$ so that the corresponding $\hat{z}_{j,n,t}$ will be 0 and thus the dummy entities will never be sampled. Since $z_{j,n,t}$ is the task selection action chosen by the agent n before being coordinated with other agents, we have $z_{j,n,t} \geq y_{j,n,t}, \forall j, n, t$ where $y_{j,n,t}$ is defined as our final task placement decisions.

Task selection conflicts. Edge devices can submit offloading requests to multiple edge clusters and this is where conflicts may arise. We do not consider the scenario where each task is served by multiple edge clusters simultaneously, for the unstable network condition between the edge clusters. To resolve task selection conflicts, we transform the attention scores in the output of the task selection phase, into task-conditioned probabilities using Bayes' theorem. Our attention scores $\hat{z}_{j,n,t}$ can be interpreted as the probability of selecting task j in the task selection phase by the agent n, which means task j was selected with a $\Pr(j|n)$ chance. For every agent that chooses task j, we consider the maximum number of task j that can be held in the cluster with the minimum required resource satisfied, as the marginal probabilities of choosing agents. The intuition is that considering load-balancing, we hope that the clusters with more available resources can more possibly get the conflicted task. Finally, the agent with the largest task-conditioned probability can schedule the task.

$$\begin{aligned}&\Pr(j|n) = z_{j,n,t}, \quad \Pr(n|j) = \frac{\Pr(j|n)\Pr(n)}{\sum_{n=1}^{N}\Pr(j|n)\Pr(n)} \\ &y_{j,n,t} = \begin{cases}1, & n = \operatorname{argmax}_{n'} \Pr(n'|j) \\ 0, & \text{otherwise.}\end{cases}\end{aligned} \tag{11}$$

Invalid action masking. We do not use the reward penalty to constrain the agents not to take invalid actions, because we observe an unstable training curve by giving a large negative reward to the agents predicting invalid actions, which is consistent with the findings of [41]. Besides, adding penalty terms into rewards needs careful tuning of the associated parameters. Intuitively, the penalty should be large enough since invalid actions violate our constraints and thus need to be prohibited. But the value functions of different actions are approximated through an NN. Large penalties may also decrease the value estimates of other actions, especially those that are close to the invalid actions but turn out good or even optimal resource allocations. To overcome this, we implement a different approach, which is to use an invalid action masking module to identify and mask all the invalid actions in the combinatorial action space to prevent our actors from predicting invalid resource allocation. We assign 0 probability to all the actions that either exceed the resource capacity or are less than the minimum required amount of resources.

Loss function and training method. In our algorithm, the granularity of the scheduling intervals is relatively small compared to the long-running ML tasks. As a result, in some timesteps, the edge clusters may either be idle or lack resources to schedule. We skip all those timesteps for training since they contain little information for the agents to learn and bring noise and instability to the training. We use the multi-agent proximal policy optimization (MAPPO) [42] to train our MARL model. The main reason why we use PPO is that we prefer a stable on-policy learning algorithm that can easily handle the large combinatorial action space. To design our training loss function, we adopt the clipped surrogate loss function of PPO as the base and customize it for our optimization problem. First, we mask all the invalid resource allocations. Then the scheduling policy can be derived from the chain rule of conditional probabilities. We further mask the final resource allocation predictions $\Pr^*(x_{j,n}|j,n)$ of the conflict agents that fail to schedule the task, while keeping the task selection predictions $\Pr(j|n)$ in the loss function.

$$\Pr^*(x_{j,n}|j,n) = \text{MASK}(\Pr(x_{j,n}|j,n), b_{j,r}, C_{r,n}) \tag{12}$$

$$\pi_{\theta_n}(\boldsymbol{a}_n|\boldsymbol{s}_n) = \Pr(j|n)\Pr^*(x_{j,n}|j,n)^m \tag{13}$$

$$m = \begin{cases}1, & y_{j,n,t} = 1, \\ 0, & \text{otherwise.}\end{cases} \tag{14}$$

We train our MARL model offline because online training is time-consuming and may easily lead to poor performance if not well tuned [13]. For every policy update step, we collect a batch of trajectories $(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r})$ from the environment, each representing the concatenation of the states, the actions and the rewards for all the agents. Every agent learns a policy $\pi_{\theta_n}(\boldsymbol{a}_n|\boldsymbol{s}_n)$, which is a joint distribution of the combinatorial actions giving the states.

V. Evaluation

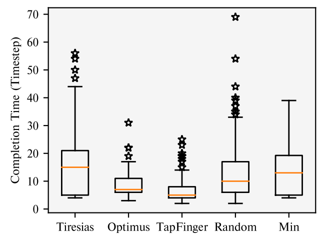
We evaluate *TapFinger* using both synthetic and test-bed ML tasks. *TapFinger* achieves considerable average completion time reduction in both environments. Experiments also demonstrate the scalability of *TapFinger*, as it outperforms baselines with increasing network scale and system workload.

A. Experiment Settings

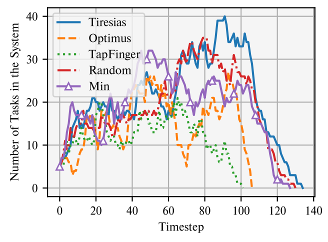
Model configurations. We implement *TapFinger* with PyTorch and use Tianshou [43], an RL library based on PyTorch, to manage the model training process. Our simulation environment implements the Gym [44] standard interface to communicate with MARL agents. For the HAN implementation, we use PyTorch Geometric library [45] to accelerate the data loading, training, and inference for HAN. We use 6 HAN layers with 4 heads for multi-head attention to generate the HAN embedding with a hidden size of 256. By default, we set the size of the pending set as 10. The pointer network module has a 2-layer transformer encoder with 4 heads for multi-head attention, and a functionally simplified attention layer to predict task selection actions. We use a 2-layer perceptron for resource allocation actions. As for the critic, we use a 2-layer perceptron that accepts the flattened HAN embedding of all the pending tasks in the system as the input. A server with $1\times$ Intel i9-12900K CPU and $1\times$ NVIDIA RTX 3080 GPU is sufficient to train our MARL model with 32 parallel training environments for either synthetic or test-bed ML tasks.

Baselines. We consider two representative ML task scheduling algorithms and two heuristics as our baselines.

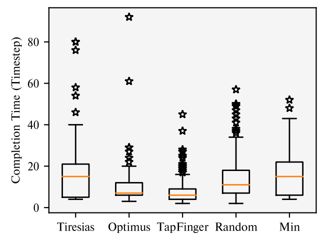
- Optimus. It uses curve-fitting performance models to estimate training speed as a function of the number of parameter servers and workers, and batch size in each task. It allocates 1 parameter server and 1 worker for each task initially, and then incrementally chooses the allocation decision with the largest marginal gain. To tailor Optimus to our problem, we consider the CPU and GPU as the two resource types and fix the batch size variable in the function to a constant.
- Tiresias. It is a preemptive scheduling algorithm based on the least-attained service and multi-level feedback queue. It divides the queue according to the GPU time thresholds and has the tasks of each queue sorted in a First-in First-out order of their start times. In our implementation, we use a 2-level feedback queue as the original paper recommended, and use the median GPU time as the threshold.
- Random/Minimum allocation. They randomly choose a task from the pending set and allocate a random/minimum but valid amount of resources to the tasks.



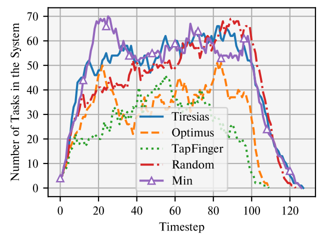
(a) 3 agents, 192 tasks.



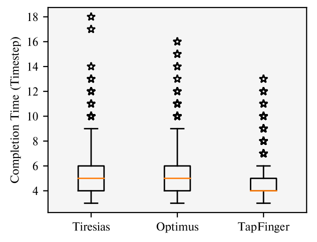
(b) 3 agents, 192 tasks.



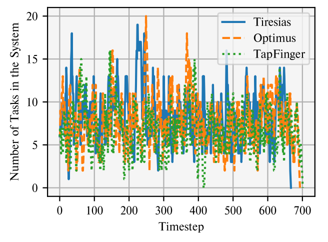
(c) 6 agents, 384 tasks.



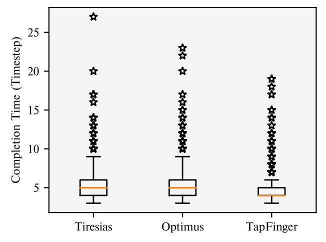
(d) 6 agents, 384 tasks.



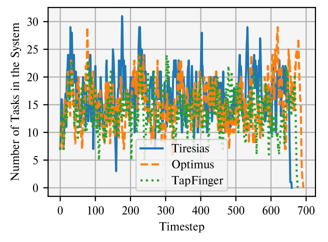
(e) 3 agents, 1000 tasks.



(f) 3 agents, 1000 tasks.



(g) 6 agents, 2000 tasks.



(h) 6 agents, 2000 tasks.

Fig. 4: Completion time and task accumulation comparison on synthetic (first row) and test-bed (second row) ML workloads.

Synthetic ML tasks. To capture the heterogeneous and non-stationary resource sensitivity of the ML tasks, we define two types of synthetic ML tasks that vary greatly in duration time and resource sensitivity. We fix the progress gain of the resource-insensitive tasks for every resource allocation. The resource-insensitive tasks can be accomplished in 4 timesteps as long as the minimum resource requirement is met. Conversely, the resource-sensitive tasks can be accomplished from 2 to 17 timesteps under different resource allocations. The speed of the resource-sensitive tasks grows sub-linearly with the number of GPUs. We use the hyperbolic tangent function as the speed function, which shares some functional properties with the formulation of Amdahl's law [46].

ML phase	Task	Model	Batch size	Duration
Training	Image classification	ConvNet	1024	34-262s
Training	Language modeling	Transformer	16384 (tokens) per GPU	75-264s
Inference	Speech recognition	Wav2Vec2	1	17-18s

TABLE I: Test-bed ML tasks.

Test-bed ML task traces. We run the test-bed ML training and inference workloads in a server with $4\times$ Intel Xeon Gold 6348 CPU and $8\times$ NVIDIA RTX 3090 GPU. We implement distributed data parallelism for 3 ML training or inference tasks with PyTorch, as shown in Table I. We run the tasks in an ML task workload with every resource allocation combination from the minimum resource requirement of 2 CPU cores and 1 GPU, to 16 CPU cores and 8 GPUs. We use the PyTorch interface to control the number of GPU devices for the tasks. To control the CPU allocation with low overhead, we set the thread affinity policy according to the CPU allocation to constrain the OpenMP threads and the data loader threads on the designated CPU cores. We downsize the running time of the tasks in the trace data. The tasks are accomplished when the validation loss reaches the preset target for the training tasks, or the outputs of all the samples are calculated for the inference tasks. We collect the validation loss of the tasks and the elapsed time for every epoch and later use these data to shape the progress functions of the tasks over time.

Task arrivals. Our MARL models are trained with finite-size ML task workloads. We set the number of tasks in the ML workloads to be 64 times the number of agents in the environment. The simulated workloads follow a Poisson distribution with an arrival rate $\lambda = 2$. We assign the bandwidth uniformly at random from $[0, 10]$ Mbps to each connection between the edge devices and the edge clusters. The edge device can be considered connectable to an edge cluster only if the bandwidth and the latency of their connection meet the minimum requirement for offloading, e.g., 5 Mbps.

B. *Performance*

We train our MARL models for 8 million steps and save the models that achieve the best evaluation results. The scheduling interval is set to be 10 seconds. We add ± 0.1 randomness to the normalized progress gains to simulate the unstable network condition between the edge devices and the edge clusters. The minimum resource requirements of both synthetic and test-bed ML tasks are 2 CPU cores and 1 GPU. Given that Fig.s 4 and 5 in [6] demonstrate at least 5 seconds in pausing and resuming a data-parallel ML task. We set the overhead of adjusting the resource allocation to be 5 seconds, which is equal to 0.5 timesteps. But even if we ignore the switching overhead, *TapFinger* can still outperform Tiresias and maintain comparable performance as Optimus.

1) Evaluation on synthetic data.: We first evaluate *TapFinger* on the synthetic ML tasks. We assume that each edge cluster has 16 CPU cores and 16 GPUs. Fig.s 4a and 4b show the completion times and the task accumulation of *TapFinger* as well as the baselines in a 3-agent environment. Because of the

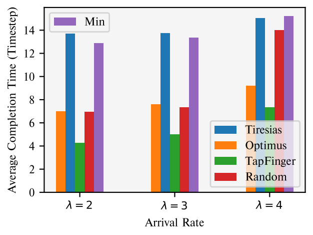


Fig. 5: Varying arrival rates.

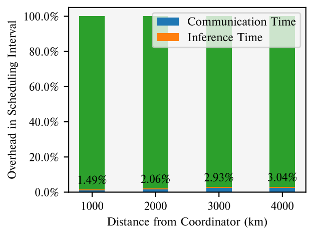
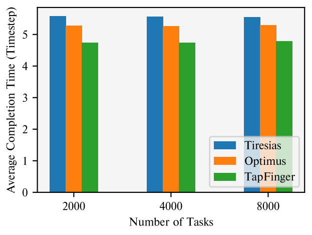
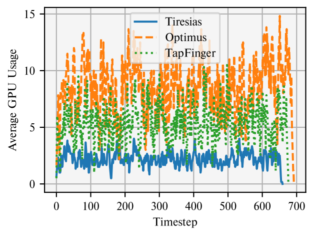


Fig. 6: Scheduling overhead.

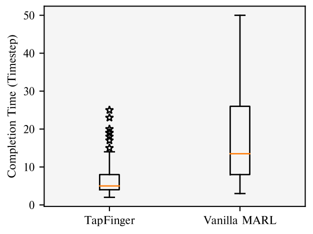


(a) Increasing workload scale.

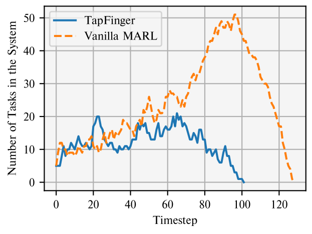


(b) GPU usage.

Fig. 8: *TapFinger* in long-running test-bed workloads.



(a) Completion time.



(b) Task accumulation.

Fig. 7: Compared with the vanilla MARL on synthetic data.

pre-run estimation of the progress function, Optimus performs better than other baselines. Tiresias assumes unknown task characteristics and only schedules tasks given the minimum resource requirements, resulting in inferior performance. We observe a 28.6% reduction in the median completion time and 25.1% reduction in the average completion time of *TapFinger*, in comparison with Optimus. Fig. 4b also shows that *TapFinger* achieves the least task accumulation.

We scale *TapFinger* to an environment with 6 edge clusters and an arrival rate of 4. As shown in Fig.s 4c and 4d, *TapFinger* still shows at least 20.5% improvement on the completion time compared with the baselines. We then decrease the arrival rate from 4 to 2, while keeping other settings unchanged. Fig. 5 shows that the task accumulation of *TapFinger* also outperforms other algorithms significantly, demonstrating that *TapFinger* is robust to varied environments once trained with sufficient workloads. We also observe a performance degradation of Optimus and Tiresias in these less heavy workloads. It can be explained as that frequently reassigning resource allocation for each task leads to severe overhead due to preemptions.

We compare *TapFinger* with a vanilla MARL algorithm without HAN and the pointer network, which uses stacking state features and chooses the task from the queue in a first-in-first-out manner and predicts the resource allocation with our invalid action masking design. As implied in Fig. 7, the vanilla MARL algorithm struggles to recognize the task characteristics and avoid task selection conflicts.

2) Evaluation on test-bed data.: We further conduct several long-running experiments on test-bed ML workloads and evaluate the overhead incurred by running the MARL algorithm. We have measured that the total data size that needs to be transmitted in each timestep as follows. The state observation and the action information take up 2KBs and 800Bytes respectively for each agent. We simulate the communication overhead using NetEm Linux kernel module [47]. We set the bandwidth between the coordinator and edge clusters as 1Mbps, and vary the network latency by increasing geographical distances according to the measurements of Alibaba Cloud edge clusters [35]. Fig. 6 shows that the inference and communication time of *TapFinger* is negligible compared to the scheduling interval.

The 3-agent and 6-agent *TapFinger* are trained with an arrival rate of 4 and 6, respectively. Each edge cluster has 16 CPU cores and 8 GPUs. We evaluate the algorithms in longer workloads than that in the training stage, and raise the resource capacity of each edge cluster to 32 CPU cores and 16 GPUs per agent. The arrival rates of the 3-agent and 6-agent test-bed environments are 1.5 and 3, respectively. Fig.s 4e-4h show that *TapFinger* still holds a scalable performance and a considerable average completion time reduction of 13.7% and 10.4% over the baselines in the corresponding environment settings. *TapFinger* is also applicable to various lengths of workloads. As shown in Fig. 8a, with the increasing workload scale, the 6-agent *TapFinger* maintains 14.5% and 10% reduction on average completion time compared with Tiresias and Optimus. The result indicates that trained *TapFinger* can be stably deployed on online edge clusters. We also look into the GPU usage of the 6-agent *TapFinger* in the 2000-task workload. Fig. 8b shows that *TapFinger* strikes a good balance between resource efficiency and completion time performance.

VI. Conclusion

We propose *TapFinger*, a distributed scheduling framework that jointly optimizes task placement and fine-grained multi-resource allocation for ML tasks in distributed edge clusters. *TapFinger* uses a MARL method based on HAN to encode the states of the edge components and their interrelation semantics. We integrate the pointer network and the conflict resolution module into our actor network to decompose our actions. To mitigate the decision conflicts problem in MARL and to yield a valid resource allocation decision, we combine Bayes' theorem and masking schemes to construct the loss function for our model training. Our experiments show that *TapFinger* can reduce average completion times by up to 28.6% and 14.5% compared with the state-of-the-art scheduling algorithms on synthetic and test-bed ML workloads, respectively.

References

[1] S. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge-computing," in *Proc. of ACM SIGMOBILE GetMobile*, 2020.

[2] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 119–135.

[3] "Aws outposts," https://nvidia.com/en-us/data-center/products/egx/.

[4] "Azure stack edge," https://azure.microsoft.com/en-us/products/azure-stack/edge/.

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

[6] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *Proc. of NSDI*, 2019.

[7] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park, "Elastic resource sharing for distributed deep learning," in *Proc. of NSDI*, 2021.

[8] V. K. V. et al., "Apache hadoop yarn: Yet another resource negotiator," in *Proc. of SoCC*, 2013.

[9] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[10] "Kubernetes native edge computing framework (project under cncf)," https://github.com/kubeedge/kubeedge.

[11] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 2019, pp. 505–513.

[12] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks," *IEEE Transactions on Mobile Computing*, 2020.

[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Meng, and W. Lin, "Dl2: A deep learning-driven scheduler for deep learning clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1947–1960, 2021.

[14] X. Zhao and C. Wu, "Large-scale machine learning cluster scheduling via multi-agent graph reinforcement learning," *IEEE Transactions on Network and Service Management*, 2021.

[15] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[16] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proc. of SC*, 2021.

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

[18] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework," in *Proc. of INFOCOM*, 2012.

[19] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, vol. 16, pp. 285–286, 2005.

[20] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2430–2439.

[21] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[22] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *Proc. of The world wide web conference*, 2019, pp. 2022–2032.

[23] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

[24] W. Kool, H. Van Hoof, and M. Welling, "Attention, learn to solve routing problems!" *arXiv preprint arXiv:1803.08475*, 2018.

[25] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[26] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[27] W. Wang, B. Li, B. Liang, and J. Li, "Multi-resource fair sharing for datacenter jobs with placement constraints," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 1003–1014.

[28] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-Resource fairness for correlated and elastic demands," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 407–424.

[29] J. Khamse-Ashari, I. Lambadaris, G. Kesidis, B. Urgaonkar, and Y. Zhao, "An efficient and fair multi-resource allocation mechanism for heterogeneous servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2686–2699, 2018.

[30] W. Gao, Q. Hu, Z. Ye, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision," *arXiv preprint arXiv:2205.11913*, 2022.

[31] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng, "A case for task sampling based learning for cluster job scheduling," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 19–33.

[32] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *Proc. of OSDI*, 2021.

[33] M. Cheong, H. Lee, I. Yeom, and H. Woo, "Scarl: Attentive reinforcement learning-based scheduling in a multi-resource heterogeneous cluster," *IEEE Access*, vol. 7, pp. 153 432–153 444, 2019.

[34] H. Lee, J. Lee, I. Yeom, and H. Woo, "Panda: Reinforcement learning-based priority assignment for multi-processor real-time scheduling," *IEEE Access*, vol. 8, pp. 185 570–185 583, 2020.

[35] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu, "From cloud to edge: a first look at public edge platforms," in *Proc. of the 21st ACM Internet Measurement Conference*, 2021, pp. 37–53.

[36] L. Luo, P. West, J. Nelson, A. Krishnamurthy, and L. Ceze, "Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud," in *Proc. of Machine Learning and Systems*, vol. 2, 2020, pp. 82–97.

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

[38] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, and M. Smelyanskiy, "Check-n-run: a checkpointing system for training deep learning recommendation models," in *Proc. of NSDI*, 2022.

[39] Z. Fang, D. Hong, and R. K. Gupta, "Serving deep neural networks at the cloud edge for vision applications on mobile platforms," in *Proc. of the 10th ACM Multimedia Systems Conference*, 2019, p. 36–47.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[41] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *arXiv preprint arXiv:2006.14171*, 2020.

[42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[43] J. Weng, H. Chen, D. Yan, K. You, A. Duburcq, M. Zhang, H. Su, and J. Zhu, "Tianshou: A highly modularized deep reinforcement learning library," *arXiv preprint arXiv:2107.14171*, 2021.

[44] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[45] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[46] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[47] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, vol. 5. Citeseer, 2005, p. 2005.