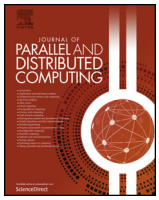
Journal of Parallel and Distributed Computing 183 (2024) 104776

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



Interference-aware opportunistic job placement for shared distributed deep learning clusters ☆

Hongliang Li [a,b], Hairui Zhao [a], Ting Sun [a], Xiang Li [a,*], Haixiao Xu [c], Keqin Li [d]

[a] College of Computer Science and Technology, Jilin University, Changchun, China
[b] Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, Changchun, China
[c] High Performance Computing Center of Jilin University, Changchun, China
[d] Department of Computer Science, State University of New York, New Paltz, NY, 12561, USA

ARTICLE INFO

Keywords:
Deep learning cluster
Interference
Online adjustment
Opportunistic sharing
Job placement

ABSTRACT

Distributed deep learning frameworks facilitate large deep learning workloads. These frameworks support sharing one GPU device among multiple jobs to improve resource utilization. Modern deep learning training jobs consume a large amount of GPU memory. Despite that, sharing GPU memory among jobs is still possible because a training job has iterative steps that its memory usage fluctuates over time. However, resource sharing also introduces the risk of job performance degradation. Co-located jobs sharing a GPU device may suffer from different levels of interference, mainly caused by memory oversharing. How to improve resource utilization while maintaining good job performance is a novel challenge for job placement strategies. This paper studies the job placement problem. We propose an opportunistic memory sharing model to describe the time-varying job memory requirements. Based on this model, we introduce an Opportunistic Job Placement Problem (OJPP) for shared GPU clusters that seek job placement configurations using a minimum number of GPU devices and guarantee user-defined performance requirements at the same time. We propose a greedy algorithm and a heuristic algorithm with computational complexities of $O(n \log n)$ and $O(n^2 \log n)$, respectively, to solve the problem. We also propose an online adjustment algorithm with the computational complexity of $O(n \log n)$ to perform updates to job placement configurations in runtime. A machine-learning-based interference prediction method is used to prepare accurate interference estimations. Extensive experiments are conducted on a GPU cluster to verify the correctness and effectiveness of our algorithms. Compared with standalone training jobs on dedicated clusters, the proposed approach reduces resource consumption by 46% in a shared cluster, while guaranteeing over 92.97% of the job performance, in terms of average job completion time.

1. Introduction

In recent years, rapidly developing artificial intelligence applications have sparked state-of-art Deep Learning (DL) technologies [19,13] to solve novel big data analysis problems, such as machine translation [14,15], computer vision [19,6], speech recognition [41,35], etc. In the meantime, deep learning networks keep developing, in terms of both model complexity and dataset size. It has spurred a new wave of Distributed Deep Learning (DDL) frameworks.

These frameworks are scalable in nature that they can efficiently host large DL jobs on GPU clusters [38]. Examples of such frameworks include TensorFlow [1], BigDL [11]. Large DDL training jobs are divided into two main parallel schemes, namely *data parallelism* and *model parallelism* [32,38]. The former divides large datasets into small subsets as chunks and trains models in SIMD fashion to deal with large datasets, while the latter splits a large model that is hard to host in one GPU device into subsets that are trained in MIMD fashion. These two schemes can be mixed to solve more complicated applications.

In *data parallelism* scheme, DL job starts multiple model instances, called *workers*. Each *worker* is fed a chunk of the dataset every iteration. *Workers* periodically perform *allreduce* communication to update their

☆ A preliminary version of the paper was published in 2020 IEEE 22nd International Conference on High Performance Computing and Communications (HPCC 2020) [29].

* Corresponding author.
E-mail addresses: lihongliang@jlu.edu.cn (H. Li), zhaohr21@mails.jlu.edu.cn (H. Zhao), sunting18@mails.jlu.edu.cn (T. Sun), lxiang@jlu.edu.cn (X. Li), haixiao@jlu.edu.cn (H. Xu), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.jpdc.2023.104776
Received 25 December 2022; Received in revised form 6 May 2023; Accepted 18 September 2023
Available online 25 September 2023

Table 1
Comparison of different distributed deep learning clusters.

System Tpe	1: Dedicated Cluster	2: Shared Cluster	3: Non-dedicated Cluster
Allocation	exclusive	shared	uncertain
Utilization	poor	medium	high
Availability	high	predictable or controlled	unpredictable
Performance	guaranteed	partial guaranteed	no guarantees

local parameters, using optimization algorithms, e.g., Stochastic Gradient Descent (SGD) [12]. Such *allreduce* operations can be performed with different communication patterns, e.g., with Parameter Server (PS) [32] or Ring All-Reduce [1], both synchronously and asynchronously. This paper focuses on *data parallelism*. It is widely applied in various application scenarios and supported by both commodity and commercial frameworks.

Deep learning algorithms are floating point computation intensive, these workloads rely on hardware accelerators like GPUs to achieve high training efficiency. In terms of how resources are shared and managed, DDL clusters can be categorized into three types, as listed in Table 1. (1) *Dedicated cluster* [16,25], that provides resources (mostly GPU) for one DL job or one user group exclusively. While it guarantees training performance and provides high resource availability at almost all times, it suffers from poor resource utilization. (2) *Shared cluster* [37,8], such as multi-tenant GPU clusters and GPU clouds, designed specifically for deep learning jobs. These clusters usually have multiple high-end GPU accelerators on a server to host multiple DL training jobs at once. Such infrastructures provide better resource utilization than *dedicated clusters*. The resource availability is usually predictable, therefore it can guarantee training performance in some scenarios. (3) *Non-dedicated clusters* [5], such as public clouds, usually host mixed workloads with both GPU-heavy DL jobs and other traditional CPU-heavy jobs. These clusters are designed in pursuit of high resource utilization but usually cannot provide performance guarantees for DL jobs. Overall, shared DDL clusters are able to achieve moderate resource utilization while providing decent training performance. It is the most efficient and widely applied environment for DL jobs on the market. However, it also introduces novel efficiency and performance challenges [24,40].

DDL cluster schedulers allocate resources, mostly GPU devices, for multiple DL jobs in a shared cluster, with the objectives of better resource utilization and higher job throughput than exclusive clusters. However, studies show that the resource utilization of individual GPUs can sometimes be quite poor, e.g., 52% on average in production systems [24]. This is mainly caused by the increased overhead brought by synchronizations among distributed training *workers*, and it can get even worse when jobs scale up. A straightforward solution is to pack more *workers* on one GPU device. The reason is (1) to achieve better *worker* locality and reduce synchronization cost, and (2) to improve device utilization. While it improves the system efficiency, it also introduces another significant problem.

It has been reported in recent works that *workers* sharing a single GPU may experience noticeable interference that adversely causes training performance downgrades [2,40,24]. As exampled in Fig. 1(a), we observe that the overall performance degradation can be as much as 113% when sharing a GPU device with *workers* of multiple deep learning jobs, compared with standalone resource allocation. It is observed for co-located *workers* from the same job and from different jobs [2,40,29]. Moreover, a DL job can suffer from different levels of interference [2] when paired with different DL jobs. For example, we observed that a ResNet *worker* suffers from significant performance slowdown (up to 122% slower) when sharing a host with a VGG19 *worker*. A LeNet *worker* achieves decent performance (18% slower) when sharing a device with an LSTM *worker*.

This is mainly caused by resource competition when sharing underlying resources besides GPU, such as CPU caches, disk I/O, network I/O, and buses (e.g., QPI, PCIe) [2,40]. Experiments in study [29] have shown that the dominant factor is GPU RAM overload. Specifically, DL jobs are iterative in nature. The GPU RAM usage of a *worker* of a DL training job usually fluctuates and follows a cyclic pattern [47]. Memory operations from multiple DL jobs can cause memory oversharing, as exampled in Fig. 1(b) when two memory usage curve peaks collide. We called this GPU *memory oversharing* among multiple jobs. We observe noticeable performance interference among co-located jobs under this circumstance. Given enough job and device data, the interference can be predicted [29], as demonstrated in Section 5.

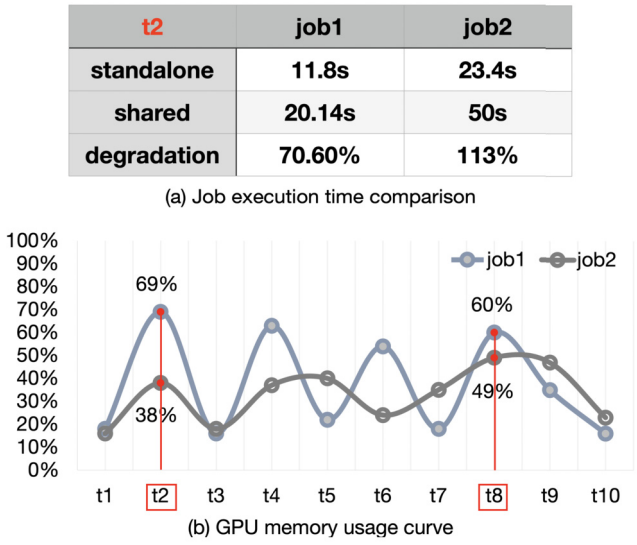


Fig. 1. Performance degradation and memory oversharing of co-located DL jobs (LeNet and ResNet).

Sharing GPU devices among multiple DL jobs presents an opportunity for higher system efficiency and job throughput, while interference among DL jobs significantly affects their training performance. How to efficiently place DL training jobs on a shared distributed GPU cluster, guaranteeing training performance in spite of the interference, is a challenging problem. This paper seeks a comprehensive solution and makes four main contributions.

1. We introduce a model to describe memory oversharing on a shared device and based on this, we introduce a novel Opportunistic Job Placement Problem (OJPP) for distributed deep learning clusters.
2. We propose greedy and heuristic algorithms to compute efficient job placement configurations with memory oversharing and interference constraints to guarantee job performance.
3. We propose an online adjustment algorithm to dynamically update job configurations according to real-time status and job interference.
4. We conduct extensive experiments on a GPU cluster with typical deep learning training workloads to verify the correctness and effectiveness of our approach.

The rest of the paper is organized as follows. In the next section, we summarize related work. Section 3 presents problem formulation and analysis. We propose our approaches for static placement and online

adjustment in Section 4. Section 5 discusses the experimental results. Section 6 concludes the paper.

2. Related work

2.1. Distributed model training

A deep learning algorithm generally consists of two stages, the training stage, and the inference stage. A deep learning training job trains a learning model, such as a Deep Neural Network (DNN), with a large number of examples (e.g., images with labeled objects). The training process is an approximate function for input-output mapping. We use the quality of the mapping to measure how well the model maps input to the correct output. Deep learning training job works in an iterative way to explore feature validity and tune hyperparameters [49]. The inference stage uses the trained model to make predictions on future inputs.

Compared to the lightweight inference stage that runs in real-time, the training stage is highly resource-consuming. This is mostly caused by the complexity of DNN and the size of training datasets. A large dataset is usually divided into data chunks and then *mini-batches*. Each training *iteration* trains one mini-batch of dataset and tunes model parameters to improve the model. A training job may generate a low-quality model at the beginning and improve the model's quality through a sequence of training iterations (usually tens of thousands or even millions) until it converges or reaches user-defined termination conditions.

In recent years, distributed model training frameworks, such as TensorFlow [1], BigDL [11], and MXNet [7], are developed to shorten training time for both large dataset and large models, following *data parallelism* and *model parallelism* [32,38] respectively. *Data parallelism* is more popular in recent years, because of the large dataset modern DL applications are dealing with. *Data parallelism* divides large datasets into small subsets as chunks and trains models in SIMD fashion. Distributed frameworks launch multiple instances (*workers*) for a DL training job so that *workers* train the model with different data chunks in parallel to accelerate the process. *Workers* perform all-reduce fashion updates to maintain global model parameters in between training iterations. Multiple communication patterns are used to implement the all-reduce update operation, such as Parameter Server [32] and Ring All-Reduce [48].

This paper focuses on the more popular *data parallelism*, and the placement of *worker* instances in the DDL cluster. The proposed approaches can adapt to different all-reduce implementations.

2.2. Resource sharing strategies

A DDL cluster hosting DL jobs is a typical multi-job distributed system. Jobs placed on the same server share physical resources, such as GPU, CPU, caches, disk I/O, network I/O, etc. DL job users specify resource requirements, e.g., the numbers and types of resources required during job submission. A cluster scheduler, e.g., Mesos [21], Yarn [43], and Kubernetes [4], is in charge of managing resources and placing *workers* of jobs. Most existing works [24] [26] formulate the job placement problem in the DDL cluster considering some of the most important types of resource, such as GPU and CPU, for two reasons, (1) the problem complexity, and (2) the fact that GPU resource partition mechanism is not clear, or not fully supported. Most related works assign resources to DL jobs in a coarse-grain fashion that the scheduler focuses on the number of *workers* to be launched and their locations.

Optimus [37] is an online scheduler for DL training jobs that adjusts job placement based on real-time training progress. By allocating better resources for jobs in their early training stage, Optimus shortens the completion time of these jobs. This is implemented by dynamic adjustment of the numbers of *workers* in runtime. TetriSched [42] dynamically allocates resources and supports specified job deadlines. Tiresias [17], is another DL job scheduler adjusting resource allocation online based on running status. It performs well when the remaining training time of jobs is hard to estimate. It is quite important for exploratory jobs, such as hyper-parameter tuning jobs.

With the development of a large shared DDL cluster, recent related works started to focus on interference issues caused by co-located training jobs [24,29]. Xiao et al. observed performance drops caused by GPU memory oversharing in [47], and proposed a scheduling framework called Gandiva to enable time-share GPUs to avoid the oversharing issue. However, switching contexts between training jobs on a GPU device or migrating training jobs between GPUs can be very expensive. Harmony [2] is a deep learning-driven DDL cluster scheduler that places training jobs in a manner that minimizes interference and maximizes performance. The black-box approach uses deep reinforcement learning (DRL) to predict job interference, and make job placement decisions. Jeon et al. made a comprehensive analysis of Shared GPU clusters and studied the correlations between *worker* locality and interference to mitigate inter-job interference. These approaches either avoid or ignore the GPU RAM overload issue, which is the main reason for interference. To the best of our knowledge, our work is the first study to explore a model-based approach to address the GPU RAM overload issue that leads to interference among co-located training jobs.

2.3. Load balance

Load balance is a classic issue in the distributed system. DDL clusters also face this issue when hosting DL jobs. This is usually caused by the unbalanced load-to-resource ratio among *workers* and fluctuated resource availability. Load balance issues in asynchronous DL jobs may cause inaccurate results and eventually affect the model convergence. In synchronous DL jobs can cause long synchronization time, increase job completion time, and reduce job throughput and resource utilization.

There are both coarse-grain and fine-grain methods to solve the load balance problem. Chen et al. adjust parameter distribution among PS *workers* and dynamically scale in and out parameter servers to mitigate straggler issues [8]. Chen et al. [5] proposed an integrated *worker* coordination mechanism that adapts *worker* load at the synchronization barriers in between training iterations. They proposed weighted gradient aggregation to ensure the model training process still converges correctly even with inconsistent batch size.

Another important mechanism to deal with the online workload and resource availability fluctuations is job migration. The fundamental checkpoint/restore operations for DL jobs are supported by a few frameworks [1,24,45]. For example, TensorFlow [1] provides user-level *saver* to preserve and resume the training progress and data for a DL job. These operations can be triggered in between iterations and are not transparent to users.

2.4. Job placement model

The job placement problem in distributed systems is a classic issue. They are usually modeled as the Bin Packing Problem (BPP), a special case of 0-1 integer linear programming. In the classic Bin Packing Problem (BP), the goal is to pack a list of given items into a minimal number of bins, satisfying resource constraints, such as item dimensions. For the general BPP, please refer to surveys [9,10].

If we consider the GPU resources requirements of a DL job as the first dimension of the bin packing problem, and the GPU memory requirement as the second dimension, the problem we are facing is closely related to the two-dimensional Vector Bin Packing problem (2D VBP) [9]. 2D VBP problem is strongly NP-hard. All the heuristics for the geometric packing problem, such as *Next-Fit Decreasing Height* (NFDH), *First-Fit Decreasing Height* (FFDH), and *Best-Fit Decreasing Height* (BFDH), can be used as a reference to solve 2D VBP. These algorithms sort items according to their sizes, representing the packing difficulty. They differ

Table 2
Notations.

	Description
V	Set of n queued jobs, $V = \{v_i \mid i \in \{1,..,n\}\}$.
B	Set of m devices, $B = \{b_k \mid k \in \{1,..,m\}\}$.
v_{ij}	A *worker* of job v_i, where $v_i = \{v_{ij} \mid i \in \{1,..,n\}, j \in \{1,..,\omega_i\}\}$.
ω_i	Number of *workers* of job v_i.
α_i	Computing resource requirement of *workers* of job v_i.
β_i	Memory resource requirement of *workers* of job v_i.
Φ	Job Placement Configuration, $\Phi = \{\phi_i \mid i \in \{1,..,n\}\}$.
γ_i	Basic memory requirement of job v_i.
δ_i	Variable memory requirement of job v_i.
p_i	Memory usage volatility probability of job v_i.
θ_i	Interference of job v_i.
θ_k	Interference of a device b_k.
$\bar{\theta}$	User-defined upper bound of interference.
θ'	Difference upper bound of interference.
τ_k	Collision probability among jobs on device $b_k \in B$.
$\bar{\tau}$	User-defined upper bound of collision probability.
c_i	Migration cost of job v_i.
$\bar{c}$	User-defined upper bound of migration cost.

in the ways of choosing a bin for an item to be put in. For details of these heuristics, please refer to survey [33].

3. Problem formulation

3.1. Resource allocation

We consider a typical application scenario for distributed model training. Multiple DL training jobs are queued in a shared distributed cluster of GPU servers. Each DL job has various resource requirements [2]. GPU resource is the most important for a DL training job. Therefore, we simplify the resource requirement of a DL job into two main aspects that affect the training performance the most, namely computing resource (e.g., GPU) and memory resource (GPU memory). Each server in a DDL cluster can provide one or multiple GPU cards. GPU cards are the resource unit in our model.

The placement of jobs is essentially the placement of *workers* in the DDL cluster since they are the most resource-consuming part of a DL job and the smallest unit of scheduling. Let $V = \{v_i \mid i \in \{1,..,n\}\}$ be a queue with n DL training jobs, and job v_i has ω_i *workers*, $v_i = \{v_{ij} \mid j \in 1,..,\omega_i\}$.

Let (α_i, β_i) be the resource requirements of job v_i, where α_i and β_i are the computing resource and memory resource requirements, respectively. Note that *workers* of a job have identical resource requirements in a data-parallel scheme. Additionally, $B = \{b_k \mid k \in \{1,..,m\}\}$ represents m candidate devices (e.g., GPU cards) to host DL jobs. Table 2 summarizes important notation.

Definition 1 *(Job Placement Configuration, JPC)*. Given a set of n deep learning training jobs $V = \{v_i \mid i \in \{1,..,n\}\}$, job v_i has ω_i *workers* $v_i = \{v_{ij} \mid j \in \{1,..,\omega_i\}\}$ with identical resource requirement (α_i, β_i), and a set of m GPU devices $B = \{b_k \mid k \in \{1,..,m\}\}$, a JPC Φ is a set of n mappings from *workers* to devices as $\phi_i(v_i \to B)$.

When we assign a *worker* v_{ij} to a device b_j, v_{ij} allocates a portion of resource from b_j. Without loss of generality, we normalize the device capacity to 1 and set $\alpha_i \in (0,1]$, $\beta_i \in (0,1]$. We seek a valid JPC that satisfies basic resource allocation principles that the total resource assigned to all *workers* on device b_k do not exceed the capacity of its total resources,

$$\sum_{\phi_i(v_{ij})=b_k} \alpha_i \le 1, \quad i \in \{1,..,n\}, j \in \{j,...,\omega_i\}, k \in \{i,...,m\}, \tag{1}$$

$$\sum_{\phi_i(v_{ij})=b_k} \beta_i \le 1, \quad i \in \{1,..,n\}, j \in \{j,...,\omega_i\}, k \in \{i,...,m\}. \tag{2}$$

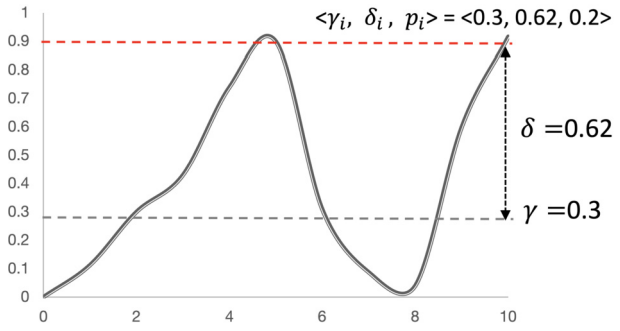


Fig. 2. Time-varying memory usage of a *worker* of ResNet training job.

3.2. Opportunistic memory sharing

DL training process is iterative, and we observe *workers* of DL jobs trained on the device have time-varying memory usage. Each training iteration causes a memory usage peak. Gaps between iterations leave memory idle. The GPU memory usage of a *worker* usually fluctuates and follows a cyclic pattern [47]. As exampled in Fig. 2, the memory usage of a *worker* of ResNet shows a clear cyclic pattern. However, current resource allocation models use fixed memory requirements which are not suitable for clearly time-varying memory usage. Moreover, using the minimum or maximum resource usage as a requirement would mean either under-allocation or over-allocation, respectively. It is not suitable to use average usage value either.

This issue becomes more serious when sharing a GPU among multiple DL jobs. Even if the total memory occupied by all *workers* on the device is within the limit. Memory usage surges from one or more *workers* on the same device may cause GPU memory oversharing, and potentially slow down all co-located jobs.

We present a stochastic model to formulate the time-varying memory requirement of *workers* of training jobs. In this model, the memory requirement of *workers* of job v_i consists of two parts, a basic sub-requirement γ_i, which represents the average memory usage of *workers* of job v_i, and a variable sub-requirement δ_i, which occurs with a probability p_i. We therefore replace the β_i in formulation (2) with tuple $\beta_i' =< \gamma_i, \delta_i, p_i >$. Take the *worker* in Fig. 2 for example, we model the memory requirement as $< \gamma_i, \delta_i, p_i > =< 0.3, 0.62, 0.2 >$. It means the *worker* requests 30% and 92% of the total capacity of a device with a probability of 0.8 and 0.2, respectively.

According to the time-varying memory requirement model, we modify the memory allocation constraint (2) as

$$\sum_{\phi_i(v_{ij})=b_k} \gamma_i + \max_{\phi_i(v_{ij})=b_k} \delta_i \le 1, \quad i \in \{1,..,n\}, j \in \{j,...,\omega_i\}, k \in \{i,...,m\}. \tag{3}$$

The stochastic memory sharing model benefits the DDL scheduler with more opportunities to share multiple jobs on one device. This would dramatically increase the resource utilization of a DDL cluster. However, there is also a risk that *workers* of multiple jobs would use the variable part of their resource requirements at the same time, and possibly lead to memory usage collisions [50]. This will significantly drag down the training performance of the affected jobs. Let D_k be the set of *workers* with time-varying memory requirements on device b_k, X_i indicates whether the variable sub-requirement of *workers* of job v_i occurs, i.e., $Pr[X_i = 1] = p_i$. Then, the collision probability of *memory* on D_k is denoted by τ_j:

$$\tau_k(D_k) = \tau\left[\sum_{v_{ij} \in D_k} X_i > 1\right] = 1 - \prod_{v_{ij} \in D_k} (1 - p_i) - \sum_{v_{ij} \in D_k} \left(p_i \prod_{v_{sg} \in D_k, v_{sg} \neq v_{ij}} (1 - p_s) \right). \tag{4}$$

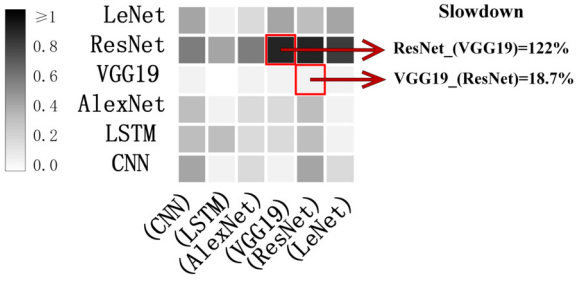


Fig. 3. Examples of DL training job interference.

In order to guarantee training performance, we introduce a user-defined collision probability threshold $\bar{\tau}$ to manage the risk of memory oversharing when sharing a GPU device among DL jobs,

$$\tau(D_k) \leqslant \bar{\tau}, \quad \forall b_k \in B. \tag{5}$$

3.3. Interference among co-located training jobs

Interference is defined as performance degradation when sharing resources among *workers* of training jobs, compared to standalone execution. Memory oversharing is one of the main causes of interference. However, co-located *workers* experience different levels of interference [40], even without memory oversharing. We have tested *workers* of different DL training jobs with different combinations, and recorded various levels of interference, as shown in Fig. 3.

Moreover, the interference effects on co-located *workers* are not necessarily equal. For example, when a ResNet *worker* shares a GPU card with a VGG19 worker, the ResNet *worker* would experience a 122% slowdown in completion time, compared to when using a GPU device alone, while the VGG19 *worker* would be less affected and perform 18.7% slower than a standalone execution.

Let t_i be the standalone completion time for *workers* of job v_i, t_i' the influenced co-located completion time, we use the slowdown percentage θ_k to represent the interference on a GPU device,

$$\theta_k := \max_{\phi(v_{ij})=b_k} \left\{ \frac{t_i' - t_i}{t_i} \right\}. \tag{6}$$

The interference among co-located workers of jobs can be obtained by testing ahead of time as discussed earlier. However, this method is best suited for scenarios with only a few types of DL training jobs submitted and limited combinations. As the scale of job submissions increases, the number of possible combinations becomes infinite and it becomes impractical to obtain interference predictions through testing. To overcome this challenge, we introduce an adaptive interference prediction algorithm (AIP) [31], which is discussed in detail in Section 4.5.

Interference among co-located workers can be predicted ahead of time by testing, as discussed earlier. However, this method is best suited for scenarios with only a few types of DL training jobs submitted and limited combinations. As the scale of job submissions increases, the number of possible combinations becomes infinite and it becomes impractical to obtain interference predictions through testing. To overcome this challenge, we introduce an adaptive interference prediction algorithm (AIP), which is discussed in detail in Section 4.

To keep the interference on a device below a certain level, we set a pre-defined upper bound of interference as a constraint when placing multiple *workers* onto a device b_k as

$$\max_{b_k \in B} \{\theta_k\} \leqslant \bar{\theta}. \tag{7}$$

Moreover, for multiple workers of a single job, we must also consider the challenge of balancing worker performance. Significant different training speeds of different workers in a job would jeopardize the overall job performance. Unbalanced *workers*, especially stragglers, would slow down training speed for jobs using synchronous parameter update scheme [3,30]. With an asynchronous scheme, [51], unbalanced workers would cause model state inconsistency that puts training accuracy at risk. Thus, in addition to limiting maximum worker interference, we also limit performance discrepancies between workers in the same job to ensure optimal training outcomes. To leave the option open for users when they can tolerate some level of interference (e.g. asynchronous model updates in data-parallel DL jobs), we use another constraint as

$$\max(v_i) - \min(v_i) \leqslant \theta', \quad \forall v_i \in V. \tag{8}$$

Here, we use $\max(v_i)$ and $\min(v_i)$ to represent the maximum and minimum interference among all *workers* of job v_i.

3.4. Job placement problem with opportunistic sharing

The opportunistic memory sharing model describes the time-varying memory usage of a GPU device shared by multiple DL training jobs. It helps us to control the risk of sharing GPU devices among DL jobs. More importantly, it provides opportunities in pursuit of the high device and system utilization, while still maintaining a certain level of performance guarantee. We introduce a novel DL training job placement problem based on the opportunistic memory sharing models and job interference matrix.

Definition 2 *(Opportunistic Job Placement Problem, OJPP).* Given a set of n deep learning training jobs $V = \{v_i | \{i \in \{1,..,n\}\}$, job v_i has ω_i *workers* ($v_i = \{v_{ij} | j \in \{1,..,\omega_i\}\}$) with identical resource requirement (α_i, β_i), and a set of m GPU devices $B = \{b_k | k \in \{1,..,m\}\}$, find a valid JPC ($\Phi$) that uses a minimum number of devices.

$$minimum \qquad \sum_{k=1}^{m} x_k \tag{9}$$

subject to:

$$\sum_{k=1}^{m} z_{ijk} = 1, \quad i \in \{1,..,n\}, j \in \{1,..,\omega_i\} \tag{10}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{\omega_i} \alpha_i z_{ijk} \leq 1, \quad k \in \{1,..,m\} \tag{11}$$

$$\sum_{\phi_i(v_{ij})} \gamma_i z_{ijk} + \max_{\phi_i(v_{ij})} \delta_i z_{ijk} \leq 1, \tag{12}$$

$$i \in \{1,..,n\}, j \in \{1...,\omega_i\}, k \in \{1,..,m\}$$

$$\tau(D_k) \leqslant \bar{\tau}, \quad k \in \{1,..,m\} \tag{13}$$

$$\max_{b_k \in B} \{\theta_k\} \leqslant \bar{\theta}, \quad k \in \{1,..,m\} \tag{14}$$

$$\max(v_i) - \min(v_i) \leqslant \theta', \quad i \in \{1,...,n\} \tag{15}$$

$$x_k = 0/1, \quad k \in \{1,..,m\} \tag{16}$$

$$z_{ijk} = 0/1, \ i \in \{1,..,n\}, j \in \{1...,\omega_i\}, k \in \{1,..,m\} \tag{17}$$

The decision variable $z_{ijk} = 1$ when *worker* v_{ij} is placed onto device b_k. Binary variable x_k represents the usage of a device. $x_k = 1$ indicates that there is at least one *worker* of job placed on device b_j. The objective function seeks to minimize the number of the occupied device while satisfying the collision probability and interference constraints. Note that we focus on computing resource requirements and do not consider communication cost between *workers* in this paper. We will take into account the communication issues in future work.

3.5. Online job placement problem adjustment with opportunistic sharing

When the real-time interference to a DL job exceeds the predefined threshold, as Eq. (7), we cannot guarantee training performance any-

more. In this case, it is necessary to adjust JPC and rearrange resource allocations.

Frequent adjustments of JPC may cause high system overhead, such as migration cost and *worker* synchronization cost. According to the observation of hardware configuration, some "violations"(i.e., interference exceeds the pre-defined upper bound) are instantaneous and then back to the normal level due to the abnormal use of some resources (e.g., startup of jobs). Thus, we wish to use Exponentially Weighted Average (EWMA) [23] to capture the continuously "violations" in order to avoid overreactions to accidental fluctuations.

Let $a \in [0,1]$ be a coefficient and $\theta_{k,t}$ be the observed interference on b_k at a moment, the exponentially weighted average of interference is

$$\theta'_{k,t} = a\,\theta'_{k,t-1} + (1-a)\,\theta_{k,t}. \tag{18}$$

The EWMA of interference $\theta'_{k,t}$ can be used as a trigger to perform JPC adjustment and calculate adjustment plans. Another important issue we need to consider is the migration cost.

A few DDL frameworks provide job migration mechanisms, such as checkpoint/restore operations for DL jobs [1,24,45]. For example, TensorFlow [1] provides user-level *saver* API to preserve and resume the training progress and data for a DL job. These operations can be triggered in between iterations but are not transparent to users. Besides the fact the checkpoint/restore functions are user-level, both operations introduce noticeable overhead, depending on the size of model parameters. Since the parameter size of a model does not change in runtime, the migration cost of a *worker* of a job is relatively stable, and it can be estimated offline. We assume for *workers* of job v_i, the migration cost c_i is estimated beforehand.

We add another constraint to the online adjustment problem to control the overall system overhead caused by job migrations as follows,

$$\sum c_i \leqslant \bar{c}. \tag{19}$$

Besides controlling individual job interference to guarantee user-defined training performance, the proposed online adjustment problem also seeks load-balanced JPC to avoid frequent adjustments.

Definition 3 *(Online Opportunistic Job Placement Adjustment Problem, OJPAP).* Given a set of n deep learning training jobs $V = \{v_i | i \in \{1,..,n\}\}$, job v_i has ω_i *workers* ($v_i = \{v_{ij} | j \in \{1,..,\omega_i\}\}$) with resource requirement (α_i, β_i), a set of available GPU devices $B = \{b_k | k \in 1,..,m\}$, and a JPC Φ, find a valid JPC (Φ') that causes the least device-level interference.

4. Opportunistic job placement algorithms

4.1. Overall discussion

The proposed job placement problems are closely related to the 2D-VBP problem. Besides GPU resource and GPU memory constraints as two dimensions in the 2D-VBP problem, the solution to the proposed problems also needs to satisfy both collision probability and interference thresholds, and try to balance the interference between *workers* of the same job. As exampled in Fig. 4, Case I and II are both valid JPCs, from the resource stand of point, Case I is the better one. However, more training jobs suffer from severe performance slowdown in Case I than in Case II. Therefore, Case II is a better solution for OJPP.

The proposed memory sharing model provides more opportunities to pack multiple *workers* on a device, and improve device utilization, but also introduces new risks of memory oversharing and interference that affects training performance. The main objective of our approach is to seek a JPC that satisfies the collision probability, interference constraint, and balance constraints to guarantee training performance, using as few devices as possible.

In this section, we first propose a best-fit-based greedy algorithm for OJPP. It is an efficient algorithm that we can use as a baseline. Then we propose a comprehensive heuristic algorithm considering all constraints to solve the problem.

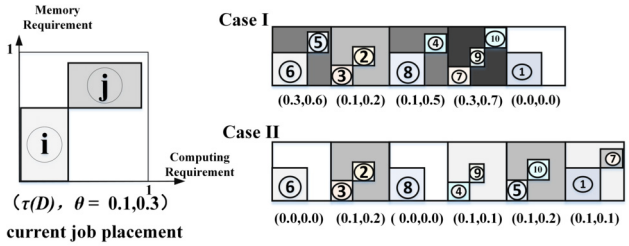


Fig. 4. Example of the different job placement configurations.

Algorithm 1 *Greedy* $(V, B, \bar{\theta}, \bar{\tau}, \theta')$.

Input: Job set V, device set B, collision probability $\bar{\tau}$, and interference bounds $\bar{\theta}$ and θ'
Output: JPC Φ

1: sort jobs V according to the resource requirements of their *workers* in descending order, and put jobs into a queue;
2: **while** there is job remaining in queue; **do**
3: take the front job;
4: $\phi = Placement(v_i, B, \bar{\theta}, \bar{\tau}, \theta')$
5: add ϕ into Φ;
6: remove v_i from job queue;
7: **end while**
8: **return** Φ

4.2. Greedy algorithm

We propose a greedy algorithm based on the classic BPP algorithm Best-fit Decreasing (BFD). In Algorithm 1, the algorithm sorts jobs in descending order according to the computing resource requirements of their workers. Then, it takes the first job from the queue and calls the function *Placement* to place this job. Function *Placement* returns the placement configuration of *workers* of job v_i. The algorithm ends until there are no jobs in the queue, and returns the overall JPC Φ (steps 2-8). This greedy algorithm ensures user-defined thresholds to guarantee training performance. As traditional best-fit algorithms, the greedy algorithm has a time complexity of $O(mn)$.

The function *Placement* as shown in Algorithm 2 handles *worker* placement for a job. The first question *Placement* needs to answer is whether all *workers* of job v_i can be placed within active devices (lines 1- 3). If not, a flag is set to False, permitting the use of new bins.

The function tries to allocate resources for *worker* v_{ij} on active devices. It checks all the constraints (lines 5 - 6), including the available resources (Eq. (11)), (Eq. (12)), the maximum slowdown ratio on device b_k (Eq. (13)), the collision probability of jobs co-located on b_k (Eq. (14)), and the difference between the maximum and minimum interference of *workers* of job v_i (Eq. (15)). Otherwise, *Placement* tries to finish *worker* placement with both active and inactive devices (line 17). Finally, *Placement* returns the placement of all *workers* of job v_i.

4.3. Heuristic algorithm

A *worker* of a DL training job occupies a basic memory space to store the model and data at the beginning after execution. Then subsequent operations, including calculating intermediate variables and passing parameters, will cause the memory usage to change regularly during the training process. We consider the varying memory requirement Eq. (3) according to condition Eq. (12).

As observed in our experiments, memory oversharing among co-located jobs significantly affects the execution time of each DL training job. The proposed heuristic seeks to reduce the collision probability of co-located jobs, in order to reduce memory oversharing.

Let $\tau(i,q)$ be the *collision probability* of *workers* of job v_i and job v_q, should they be assigned to the same device. We define the *gregariousness*

Algorithm 2 *Placement* $(v, B, \bar{\theta}, \theta', \bar{\tau})$.

Input: Job v, device set B, collision probability bound $\bar{\tau}$, and interference bounds $\bar{\theta}$ and θ'
Output: ϕ Placement of *workers* of job v_i

Flag = True, $\phi = Null$
loop: { *label for jumping* }
while ϕ is Null **do**
  for all *workers* of job v_i **do**
    for all active $b \in B$ **do**
      find device b that satisfies the following:
      $(\alpha_i, \beta_i) \geq$ R && $\tau(D_k) \leqslant \bar{\tau}$ &&
      potential θ_k is the lowest among active devices &&
      $\theta_k \leq \bar{\theta}$ && max(v_i) - min(v_i) $\leqslant \theta'$ &&
      place *worker* v_{ij} on device b_k does not break the balance of other *workers* on b_k;
      if Flag == True **then** {*only use active devices*}
        if b is not found **then**
          Flag = False
          $\phi = Null$ {*reset the mapping*}
          min(v_i) = 0
          goto loop;
        end if
        assign v_{ij} to b and add $v_{ij} \rightarrow b$ into ϕ;
      else {*active devices combined with new devices*}
        if b is not found **then**
          open a new bin b', add b' into B;
        end if
        assign v_{ij} to b and add $v_{ij} \rightarrow b$ into ϕ;
      end if
    end for
  end for
end while
return ϕ

of a job v_i in a group V as

$$\tau(i, V) = \max_{v_q \in V} \{\tau(i, q)\}. \tag{20}$$

The fitness of a job indicates its potential correlations to the other jobs in a group. In Algorithm 3, we first sort the job queue according to $\tau(i, V)$ in ascending order. The first job in order would be the most easygoing one, it has the least potential to affect the other jobs. This gives the other queuing jobs more opportunities to be assigned in an active device, instead of a vacant one. Then *Heuristic* performs the same steps as *Greedy* to place *workers* of this job. In each round, the queue is changed and therefore requires an update. Adding an extra sorting procedure with a computational complexity of $O(n \log n)$, the computational complexity of this heuristic is $O(n^2 \log n)$.

Algorithm 3 *Heuristic* $(V, B, \bar{\theta}, \bar{\tau}, \tau(i, V), \theta')$.

Input: Job set V, device set B, collision probability $\bar{\tau}$, gregariousness $\tau(i, V)(i \in \{1, .., n\})$, interference bounds $\bar{\theta}$ and θ'
Output: JPC Φ

sort job queue according to the resource requirements in descending order;
while there is job remaining in queue; **do**
  take the front job;
  sort job queue according to collision potential $\tau(i, V)$ in ascending order;
  $\phi = Placement(v_i, B, \bar{\theta}, \bar{\tau}, \theta')$
  add ϕ into Φ;
  remove v_i from job queue;
end while
return Φ

4.4. Online job placement adjustment

Interference among co-located *workers* of jobs fluctuates over time. On one hand, the iterative nature of DL jobs makes the resource usages change in the runtime. The initial interference may not be accurate all the time. On the other hand, resource availability of a DDL cluster changes dynamically, caused by completed jobs leaving the system and the arrival of new jobs. Therefore, an online adjustment mechanism is needed to rearrange JPC. The OJPAP problem seeks to adjust job placement configuration and balance interference among devices. Besides all the constraints in OJPP, OJPAP adds another one to keep the migration cost within control as Eq. (19).

First, we keep two device queues for the adjustment process. The first queue, named *priority queue*, keeps devices in descending order according to their interference θ_k (Eq. (6)). Devices suffering from the most severe performance degradation will have the highest priority in the adjustment process. The second queue, named *resource queue*, keeps devices in descending order of available resources.

Then we pick a *worker* of job v_i on the first device in *priority queue* with the least migration cost among co-located jobs. Then, we try to place this *worker* onto a new device, in order from *resource queue*. Finally, we update JPC if such a migration plan can be found.

All the constraints, including resource requirements, interference, collision probability, and balance between *workers* of job and migration cost should be satisfied for an updated JPC.

The process finishes when the first half of devices in *priority queue* get rearranged. The algorithm with computational complexities of $O(n \log n)$.

Algorithm 4 *Adjustment* $(V, B, \bar{\theta}, \bar{\tau}, \bar{c}, \theta', c_i^m, \Phi)$.

Input: Job set V, device set B, constraints $\bar{\tau}$, $\bar{\theta}$, θ', $\bar{c}$, migration cost for jobs c_i, and original JPC Φ
Output: JPC Φ'

sort devices B according to their interference in descending order, and put the first half of the devices into *priority queue*;
sort devices B according to their available resource in descending order, and put them into *resource queue*;
$\Phi' = \Phi$;
while there has devices remaining in *priority queue*; **do**
  take the *worker* v_{ij} with least c_i in the front device and remove the device from *priority queue*;
  if $c_i > \bar{c}$ **then**
    continue;
  end if
  for all $b \in$ *resource queue* **do**
    find device b that satisfies the following:
    $(\alpha_i, \beta_i) \geq$ R && $\tau(D_k) \leqslant \bar{\tau}$ &&
    potential θ_j is the lowest among active devices &&
    $\theta_k \leq \bar{\theta}$ && max(v_i)-min(v_i) $\leqslant \theta'$ &&
    place *worker* v_{ij} on device b_j does not break the balance of other *workers* on b_k;
  end for
  assign v_{ij} to b and add $v_{ij} \rightarrow b$ into ϕ;
  add ϕ into Φ';
  Update priority and resource queue;
end while
return Φ'

4.5. Interference analysis and prediction

To compute efficient job placement configurations, we need to obtain good perspectives of the potential interference among co-located *workers*. The interference is mainly caused by resource competition of shared resources, such as CPU caches, disk I/O, network I/O, and buses (e.g., QPI, PCIe) [2,40], and the interference level closely depends on the job types as well as the underlying system parameters.

We apply an adaptive interference prediction algorithm AIP [31] to prepare accurate interference estimations. AIP is our previous work, and it is a machine-learning-based interference prediction method. Given a set of jobs, AIP firstly collects the system configurations (GPU info, main memory, disk I/O, network I/O) as input, then chooses proper prediction technology by going over a set of methods (Linear Regression [36], SVR [44], Decision Tree Regression [34] and K-Neighbors [18]) to predict interference. The mean absolute error between the real interference and the predicted interference is taken as the metric to evaluate each method. Extensive experiments show that the average error of interference prediction of AIP is under 7% [31].

Table 3
Testing deep learning models and datasets.

DATA	CIFAR10 [20], MNIST [46]
DL Network	LeNet [28] , ResNet [19], VGG [39], AlexNet [27], LSTM [22], CNN

Table 4
Comparing approaches.

Algorithm	Description
Best-fit	Best-fit strategy.
I-Greedy	Greedy algorithm with interference constraint.
B-Greedy	Greedy algorithm with balance constraint.
C-Greedy	Greedy algorithm with collision constraint.
OJPP	Heuristic with opportunistic job placement.
OJPAP	Online adjust with opportunistic job placement.

5. Evaluations

5.1. Evaluation methodology

Training Model Setting. We generate workloads from six representative DL networks which are trained by two image datasets. The models and datasets we used are listed in Table 3.

Collision Probability Calculation. We set up a TensorFlow [1] system to test and collect data for formulating collision probabilities. First, the stand-alone execution time is obtained when each job uses a GPU (NVIDIA Tesla P40 with 24 GB memory) exclusively. In this process, we randomly choose 100 sampling points to gather memory usage and performance info of a job. These data are used to calculate collision probability as Eq. (4) for candidate training jobs and as the input parameter for our algorithms.

Interference Constraint. We use the TensorFlow system to gather interference data for different training jobs. We have tested 6 types of training jobs and put the data into an interference matrix, as shown in Fig. 3. Moreover, we also use AIP [31] to predict the interference. The collision probability and interference thresholds are user-defined to suit different application scenarios. We will discuss some options later.

Job Migration Mechanism. We use the TensorFlow framework to collect the migration cost of different training jobs. The checkpoint and resume operations are implemented with user-level interface *saver*. The data collected are used as an input value in the scheduling algorithm. Dynamic JPC adjustment process is triggered by users in runtime, which leaves room to explore more comprehensive migration triggering strategies.

5.2. Experimental settings

To evaluate the effectiveness of the placement and adjustment algorithm proposed by us, we simulate the environment of the cluster system. The capacity of each GPU is 11 GB (according to ASUS TURBO-GTX1080Ti GPUs), and for the jobs submitted to the simulated cluster, we use the info collected from real-world measurements.

Workloads. We generate four workload queues, each with 20 DL training jobs, and each job has 1-10 *workers*. In addition, we set the constraint parameters as $(\bar{\tau}, \bar{\theta}, \theta') = (0.1, 0.2, 0.1)$. Detailed information about the DL jobs is listed in Table 3. The workloads are divided into three types: (a) ***Mixed*** workload that 50% jobs either have high collision potential or high interference potential; (b) ***C-high*** workload that jobs have high collision potential (e.g., ResNet); (c) ***I-high*** that jobs have high interference potential (e.g., LeNet) (d) ***Average*** workload that jobs have neither high collision potential nor high interference potential).

Baseline. As listed in Table 4, we compare the proposed two opportunistic algorithms with the following approaches,

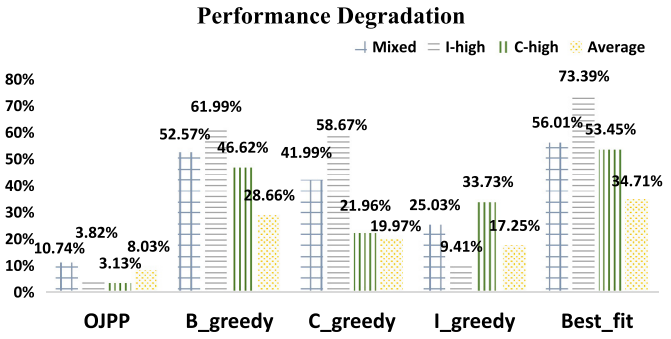


Fig. 5. Performance degradation.

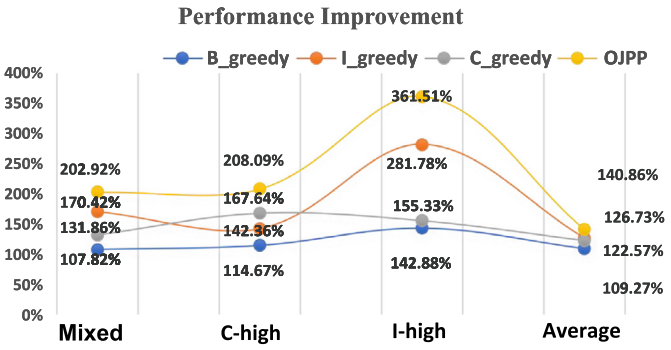


Fig. 6. Performance improvement.

- Best-fit strategy. Traditional Best-fit two-dimensional bin packing algorithm as a baseline.
- I-greedy strategy. Greedy job placement algorithm based on best-fit, only considering interference constraint.
- B-greedy strategy. Greedy job placement algorithm based on best-fit, only considering balance constraint between *workers* of job.
- C-greedy strategy. Greedy job placement algorithm based on best-fit, only considering collision probability constraint.

5.3. Results and analysis

5.3.1. Static placement

Job completion time Job completion time is the most important metric for user experience and service quality. We have tested all four different workloads with five job placement strategies. In all cases, the average co-located job completion time is longer than those of stand-alone execution. Compared to the job completion time stand-alone, the performance degradation and performance improvement are shown in Fig. 5 and Fig. 6.

The strategy that performs the worst is the traditional Best-fit algorithm, which causes a 54.39% slowdown in execution time on average. Compared to the baseline, I-Greedy, B-Greedy, and C-Greedy introduce additional constraints to ensure performance quality and achieved a smaller average job completion time. The proposed heuristic algorithm performs the best, it ensures training performance with all four types of workloads. It has over 228.34% of improvements compared to the baseline on average.

We illustrate detailed data with six types of training jobs. Fig. 7 shows the performance degradation of each type of job in mixed workload. We could see that some algorithms performed poorly dealing with ResNet jobs. It is because ResNet jobs are susceptible to other training jobs. Only I-greedy and OJPP deal with ResNet jobs well due to their constraint for interference, which causes ResNet jobs to end up being placed on a device alone, and they do not have performance degradation. On the other hand, by using the C-greedy algorithm, all types of jobs except ResNet have a low-performance degradation. It indicates

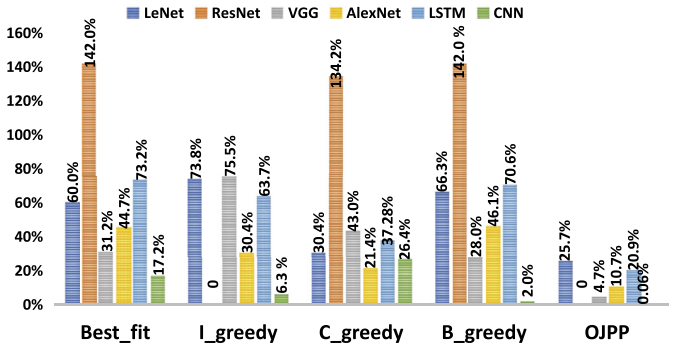


Fig. 7. Different job performance degradation comparison.

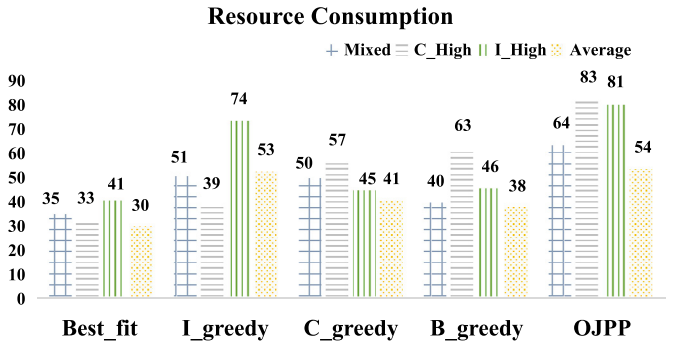


Fig. 8. Resource consumption.

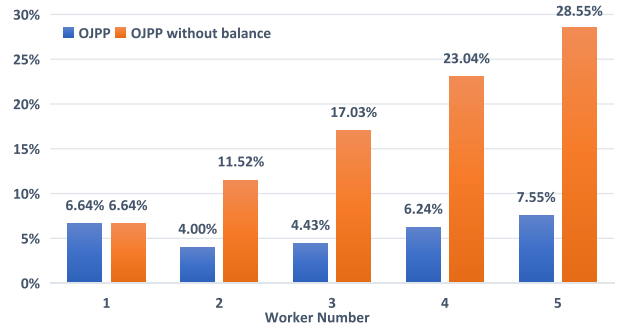


Fig. 9. Performance degradation comparison between OJPP and OJPP without balance constraint.

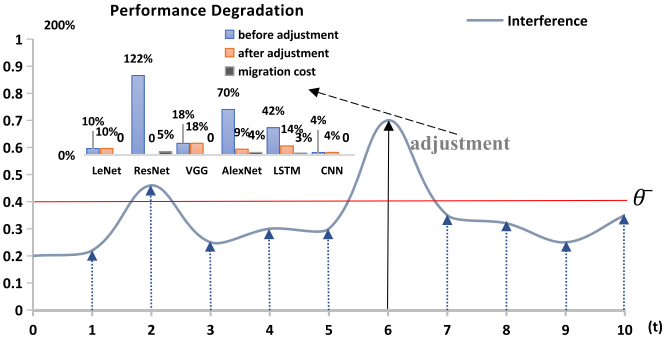


Fig. 10. Interference fluctuation.

that the opportunistic memory sharing model proposed by us is effective. But because there is no interference constraint in C-greedy, which causes ResNet jobs co-located with other training jobs and suffer terrible performance degradation. Here, we could see that all algorithms cannot handle LeNet and LSTM well except C-greedy and OJPP. This is because they are high memory collision jobs. The proposed heuristic sorts jobs according to their collision potential and takes all constraints into account. This greatly improved the job completion time. Note that B-greedy has little improvement compared to the baseline because it only uses the balance strategy and it results in *workers* of the same job being placed together regardless of interference and collision.

Number of bins This test illustrates the performance of the proposed algorithms in terms of the number of devices consumed in Fig. 8. Among all five algorithms, the traditional Best-fit algorithm uses the least number of devices. This is because it tries to pack all the jobs together, without considering any constraints, which leads to poor execution performance, as mentioned earlier. On the other hand, the proposed algorithms use 82.9% extra devices to improve 202.9% job performance in the mixed type of workload.

Influence of balance To verify the effectiveness of balance in the case of multi-workers jobs, we compare OJPP with OJPP without considering the balance between *workers* of jobs. Fig. 9 shows that the performance degradation of these two strategies with the number of *workers* of jobs changes. We could see as the number of *workers* of jobs increases, OJPP without balance constraint performs worse and worse, while the performance of OJPP is relatively steady. It is because the cost of synchronization between *workers* of a job is expensive if ignoring the balance constraint. As the number of *workers* of a job increases, the synchronization cost is getting higher. Here, to make results intuitive, we assume the communication between *workers* is synchronous. Even in asynchronous communication, a large gap between *workers* will also result in a loss of accuracy of the model.

5.3.2. Online adjustment

Dynamic change First, we tested the interference fluctuation in the runtime with *average* workload. As shown in Fig. 10. During the first half part of the test, we observed interference fluctuation and a minor violation of the interference threshold $\bar{\theta}$. There is no adjustment at time $t2$ due to the condition (EWMA of interference θ'_{tj}) of the trigger of the adjustment algorithm not being met. Then completed jobs kept leaving the system and new jobs are submitted to the cluster, we experienced some large interference raise. We performed an online adjustment algorithm OJPAP at time $t6$ and brought the interference level back to healthy.

Benefits and costs The subfigure of Fig. 10 illustrates the detailed results of the adjustment algorithm. As shown, the proposed adjustment algorithm significantly reduced the average interference level among jobs in the cluster. Compared with high performance degradation (122% at most), the migration cost is negligible (5%).

5.3.3. Interference prediction

This test is designed to verify the effectiveness of AIP [31] as an interference prediction method. As mentioned in Section 5.1, we prepare two interference matrices, in Fig. 11, the first matrix records the interference between jobs in real-world cluster, and the anther interference matrix is predicted by AIP. The predicted matrix has an average 7% error. As shown in Fig. 11, we use these two matrices to complete the job placement independently, and compare the results. AIP works well with OJPP in that the predicted matrix only suffers from 7.8% performance drop on average, compared with ideal application setting which use the first matrix. The performance of other algorithms has a difference by using both matrices, while the difference is negligible, especially for OJPP.

5.3.4. Trade-off

The proposed job placement algorithms are controlled by three user-defined parameters, which are collision probability $\bar{\tau}$ and interference $\bar{\theta}$ and θ'. These parameters ensure user-defined service quality and

Table 5
Algorithm performance under different control parameters.

$(\bar{\tau}, \theta', \bar{\theta})$	(0.05,0.1,0.1)	(0.05,0.2,0.1)	(0.05,0.4,0.1)	(0.1,0.2,0.1)	(0.1,0.2,0.2)	(0.1,0.4,0.2)
Guaranteed Performance	99.96%	94.23%	84.37%	92.97%	87.96%	83.17%
Reduced Resource Consumption	17%	39%	57%	46%	52%	61%

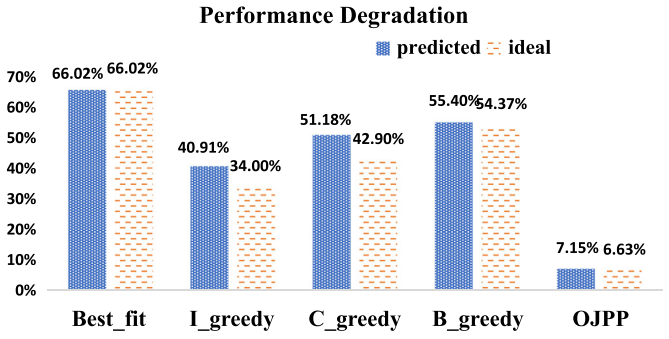


Fig. 11. Performance degradation comparison among algorithms by using predicted and measured interference.

provide guaranteed performance. Changing the parameter settings will affect both job performance and resource utilization. We list six representative parameter settings in Table 5. The corresponding guaranteed performance and reduced resource consumption are compared with jobs in the dedicated cluster which execute stand-alone. When we adjust the parameter constraint to make them larger, fewer resources are used and a lower performance is guaranteed (e.g., $(\bar{\tau}, \theta', \bar{\theta}) = (0.05, 0.1, 0.1)$, 17% resources consumption is reduced and 99.96% performance is guaranteed, when we adjust the θ' to 0.2, 39% resources consumption is reduced and 94.23% performance is guaranteed). Thus, for different workloads and requirements, we need to adjust to find a proper set of parameters. Specifically, when we set the parameters to $(\bar{\tau}, \theta', \bar{\theta}) = (0.1, 0.4, 0.2)$, the heuristic uses 61% less resources but only guarantees 83.17% of the training performance. This setting can be used to reserve resources. On the other hand, with $(\bar{\tau}, \theta', \bar{\theta}) = (0.05, 0.1, 0.1)$, the algorithm achieves 99.96% training performance with less 17% resources. It is the most aggressive setting that is suitable for users seeking high performance. In this experimental setting, we think the most efficient and balanced setting would be $(\bar{\tau}, \theta', \bar{\theta}) = (0.1, 0.2, 0.1)$, the heuristic maintains over 92.97% of the standalone performance and uses more than 46% less resources. Therefore, there is a trade-off between resource consumption and performance, users could set these parameters to meet their own requirements.

5.4. Application scenario

The proposed job placement algorithm OJPP can be used as a scheduling strategy for deep learning clusters. OJPP is capable of computing job placement configurations with high device utilization while satisfying user-defined performance bounds (collision probability $\bar{\tau}$ and interference $\bar{\theta}$). Our heuristic algorithm opportunistically puts multiple jobs on one device to improve resource utilization and keep the risk of performance downgradation in control. In order to deal with online interference fluctuations, an OJPAP algorithm is proposed to rearrange job placement configurations. OJPAP can be performed periodically or triggered when the device interference level reaches a certain level.

6. Conclusions

This paper focuses on novel challenges for distributed deep learning clusters, especially shared GPU clusters. In this type of environment, training jobs share one GPU device to improve resource utilization. However, interference among these co-located jobs brings significant performance slowdowns. We first analyze the interference issue in shared GPU clusters and identify the main reason, GPU memory oversharing caused by training jobs' fluctuating memory usage. We propose an opportunistic memory sharing model to formulate the time-varying memory usage for co-located jobs. With this model, we introduce an Opportunistic Job Placement Problem (OJPP) for distributed DL clusters. We seek opportunities to place jobs on shared GPU devices in pursuit of high device utilization while managing the risk of interference by supporting user-defined parameters. We propose a greedy algorithm and a heuristic algorithm with computational complexities of $O(n \log n)$ and $O(n^2 \log n)$, respectively. Moreover, we propose an online adjustment algorithm with computational complexities of $O(n \log n)$ to update job placement configurations according to the same principle and constraints as OJPP. We conduct extensive experiments on a GPU cluster in the HPC Center of Jilin University to verify the correctness and the efficiency of our approach. Compared with standalone training jobs on dedicated clusters, the interference-aware opportunistic job placement approach reduces resource consumption by 46% in a shared cluster, while guaranteeing over 92.97% of the job performance, in terms of average job completion time. We also make suggestions on how to set user-defined parameters for different application scenarios to achieve better system efficiency.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgment

This work was supported by the Natural Science Foundation of Jilin Province (Grant 20230101062JC), the National Key Research and Development Plan of China (Grant 2017YFC1502306), by the National Natural Science Foundation of China (NSFC) (No. 61602205).

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, et al., TensorFlow: large-scale machine learning on heterogeneous distributed systems, in: USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.
[2] Y. Bao, Y. Peng, C. Wu, Deep learning-based job placement in distributed machine learning clusters, in: IEEE International Conference on Computer Communications (INFOCOM), 2019.
[3] S. Basu, V. Saxena, R. Panja, A. Verma, Balancing stragglers against staleness in distributed deep learning, in: IEEE International Conference on High Performance Computing (HiPC), 2018.
[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, Omega, and Kubernetes, Commun. ACM (2016).
[5] C. Chen, Q. Weng, W. Wang, B. Li, B. Li, Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments, in: ACM Symposium on Cloud Computing (SoCC), 2020.
[6] M. Chen, A. Radford, J. Wu, H. Jun, P. Dhariwal, Generative pretraining from pixels, in: International Conference on Machine Learning (ICML), 2020.
[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, 2015, MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems, ArXiv.

[8] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, C. Guo, Elastic parameter server load distribution in deep learning clusters, in: ACM Symposium on Cloud Computing (SoCC), 2020.
[9] H.I. Christensen, A. Khan, S. Pokutta, P. Tetali, Approximation and online algorithms for multidimensional bin packing: a survey, Comput. Sci. Rev. (2017).
[10] E.G. Coffman, J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin packing approximation algorithms: survey and classification, in: Handbook of Combinatorial Optimization, 2013.
[11] J.J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C.L. Zhang, Y. Wan, Z. Li, et al., BigDL: a distributed deep learning framework for big data, in: ACM Symposium on Cloud Computing (SoCC), 2019.
[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al., Large scale distributed deep networks, in: Annual Conference on Neural Information Processing Systems (NeuralPS), 2012.
[13] J. Devlin, M.W. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2019.
[14] C. Fan, Y. Tian, Y. Meng, N. Peng, X. Sun, F. Wu, J. Li, Paraphrase generation as unsupervised machine translation, in: Computational Linguistics (COLING), 2021.
[15] C. Federmann, O. Elachqar, C. Quirk, Multilingual whispers: generating paraphrases with translation, in: Conference on Empirical Methods in Natural Language Processing (EMNLP), 2019.
[16] P. Goyal, P. Dollár, R.B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, 2017, Accurate, large minibatch SGD: training ImageNet in 1 hour, ArXiv.
[17] J. Gu, M. Chowdhury, K.G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, C. Guo, Tiresias: a GPU cluster manager for distributed deep learning, in: USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2019.
[18] G. Hamerly, C.P. Elkan, Learning the k in k-means, in: Annual Conference on Neural Information Processing Systems (NeuralPS), 2003.
[19] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[20] K. He, X. Zhang, S. Ren, J. Sun, Identity mappings in deep residual networks, in: European Conference on Computer Vision (ECCV), 2016.
[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, I. Stoica, Mesos: a platform for fine-grained resource sharing in the data center, in: Symposium on Network System Design and Implementation (NSDI), 2011.
[22] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (1997) 1735–1780.
[23] A. Ingolfsson, E. Sachs, Stability and sensitivity of an EWMA controller, J. Qual. Technol. (1993).
[24] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, F. Yang, Analysis of large-scale multi-tenant GPU clusters for DNN training workloads, in: USENIX Annual Technical Conference (ATC), 2019.
[25] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, X. Chu, Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes, 2018, ArXiv.
[26] H. Jiang, Y. Chen, Z. Qiao, T.H. Weng, K.C. Li, Scaling up MapReduce-based big data processing on multi-GPU systems, in: IEEE International Conference on Cluster Computing (CLUSTER), 2015.
[27] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Annual Conference on Neural Information Processing Systems (NeuralPS), 2012.
[28] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (1998) 2278–2324.
[29] H. Li, T. Sun, X. Li, H. Xu, Job placement strategy with opportunistic resource sharing for distributed deep learning clusters, in: IEEE International Conference on High Performance Computing and Communications, 2020, in: International Conference on Smart City, 2020, in: International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2020.
[30] H. Li, D. Xu, Z. Xu, X. Li, Hybrid parameter update: alleviating imbalance impacts for distributed deep learning, in: IEEE International Conference on High Performance Computing and Communications (HPCC), 2022.
[31] H. Li, N. Zhang, T. Sun, X. Li, Performance interference analysis and prediction for distributed machine learning jobs, J. Comput. Appl. 42 (6) (2022) 1649–1655.
[32] M. Li, D.G. Andersen, J.W. Park, A.J. Smola, A. Ahmed, V. Josifovski, J. Long, E.J. Shekita, B.Y. Su, Scaling distributed machine learning with the parameter server, in: Symposium on Operating Systems Design and Implementation (OSDI), 2014.
[33] A. Lodi, S. Martello, M. Monaci, Two-dimensional packing problems: a survey, Eur. J. Oper. Res. (2002).
[34] W.Y. Loh, 2008, Classification and regression tree methods.
[35] N. Moritz, T. Hori, J.L. Roux, Triggered attention for end-to-end speech recognition, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2019.
[36] J. Neter, W. Wasserman, M.H. Kutner, Applied Linear Regression Models, 1983.
[37] Y. Peng, Y. Bao, Y. Chen, C. Wu, C. Guo, Optimus: an efficient dynamic resource scheduler for deep learning clusters, in: European Conference on Computer Systems (EuroSys), 2018.
[38] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M.P. Reyes, M.L. Shyu, S.C. Chen, S.S. Iyengar, A survey on deep learning: Algorithms, techniques, and applications, ACM Comput. Surv.
[39] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: International Conference on Learning Representations (CoRR), 2015.
[40] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, H. Yang, Owl: performance-aware scheduling for resource-efficient function-as-a-service cloud, in: ACM Symposium on Cloud Computing (SoCC), 2022.
[41] Z. Tian, J. Yi, Y. Bai, J. Tao, S. Zhang, Z. Wen, Synchronous transformers for end-to-end speech recognition, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2020.
[42] A. Tumanov, T. Zhu, J.W. Park, M.A. Kozuch, M. Harchol-Balter, G.R. Ganger, TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters, in: European Conference on Computer Systems (EuroSys), 2016.
[43] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache Hadoop YARN: yet another resource negotiator, in: ACM Symposium on Cloud Computing (SoCC), 2013.
[44] M. Wauters, M. Vanhoucke, Support vector machine regression for project control forecasting, Autom. Constr. (2014).
[45] Y. Wu, K. Ma, X. Yan, Z. Liu, Z. Cai, Y. Huang, J. Cheng, H. Yuan, F. Yu, Elastic deep learning in multi-tenant GPU clusters, IEEE Trans. Parallel Distrib. Syst. (2021).
[46] H. Xiao, K. Rasul, R. Vollgraf, Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017, ArXiv.
[47] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al., Gandiva: introspective cluster scheduling for deep learning, in: USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.
[48] M. Yu, B. Ji, H. Rajan, J. Liu, On scheduling ring-all-reduce learning jobs in multi-tenant GPU clusters with communication contention, in: MobiHoc, 2022.
[49] H. Zhang, L. Stafman, A. Or, M.J. Freedman, SLAQ: quality-driven scheduling for distributed machine learning, in: ACM Symposium on Cloud Computing (SoCC), 2017.
[50] S. Zhang, Z. Qian, J. Wu, S. Lu, L. Epstein, Virtual network embedding with opportunistic resource sharing, IEEE Trans. Parallel Distrib. Syst. (2013).
[51] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z. Ma, T.Y. Liu, Asynchronous stochastic gradient descent with delay compensation, in: International Conference on Machine Learning (ICML), 2016.

Hongliang Li is an associate professor of College of Computer Science and Technology (CCST), Jilin University in Changchun, China. He received the PhD degree in computer application technologies from CCST, JLU, in 2012. He was a visiting scholar with the Department of Computer and Information Sciences, Temple University, Philadelphia. His research current interests include resource scheduling and fault tolerance for distributed system, big data computing, user-level and system-level HPC optimization.

Hairui Zhao received his B.S. degree from the College of Computer Science and Technology, Jilin University, Changchun, China, in 2019. He is currently pursing his Ph.D. degree in the College of Computer Science and Technology, Jilin University. His primary research interests include cloud computing, high performance computing, resource management and job placement.

Ting Sun received the master degree from College of Computer Science and Technology, Jilin University, Changchun, China, in 2021. Her research interests include cloud computing and high performance computing.

Xiang Li is an engineer and a Ph.D student of College of Computer Science and Technology (CCST), Jilin University in Changchun, China. Her research interests include computer network, distributed deep learning system.

Haixiao Xu is a senior engineer in HPC center of Jilin University and a Ph.D student of College of Computer Science and Technology (CCST), Jilin University, Changchun, China. His research interests include high performance computing, resource scheduling and fault-tolerant.

Keqin Li received the B.S. degree in computer science from Tsinghua University in 1985 and the Ph.D. degree in computer science from the University of Houston in 1990. He is currently a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University (China). He has authored or coauthored more than 940 journal articles, book chapters, and refereed conference papers. He received several best paper awards from international conferences including PDPTA-1996, NAECON-1997, IPDPS-2000, ISPA-2016, NPC-2019, ISPA-2019, CPSCom-2022. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top five most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He was a 2017 recipient of the Albert Nelson Marquis Lifetime Achievement Award for being listed in Marquis Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in American Education for more than twenty consecutive years. He received the Distinguished Alumnus Award of the Computer Science Department at the University of Houston in 2018. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He is a Member of SUNY Distinguished Academy. He is an AAAS Fellow, an IEEE Fellow, and an AAIA Fellow. He is a Member of Academia Europaea (Academician of the Academy of Europe).