KIPS Trans. Comp. and Comm. Sys.
Vol.11, No.7 pp.205~216
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2022.11.7.205

Dynamic Resource Adjustment Operator Based on Autoscaling for Improving Distributed Training Job Performance on Kubernetes

Jinwon Jeong[†] · Heonchang Yu[††]

ABSTRACT

One of the many tools used for distributed deep learning training is Kubeflow, which runs on Kubernetes, a container orchestration tool. TensorFlow jobs can be managed using the existing operator provided by Kubeflow. However, when considering the distributed deep learning training jobs based on the parameter server architecture, the scheduling policy used by the existing operator does not consider the task affinity of the distributed training job and does not provide the ability to dynamically allocate or release resources. This can lead to long job completion time and low resource utilization rate. Therefore, in this paper we proposes a new operator that efficiently schedules distributed deep learning training jobs to minimize the job completion time and increase resource utilization rate. We implemented the new operator by modifying the existing operator and conducted experiments to evaluate its performance. The experiment results showed that our scheduling policy improved the average job completion time reduction rate of up to 84% and average CPU utilization increase rate of up to 92%.

Keywords : Kubeflow, Kubernetes, Distributed Deep Learning Training, Resource Adjustment Operator

쿠버네티스에서 분산 학습 작업 성능 향상을 위한 오토스케일링 기반 동적 자원 조정 오퍼레이터

정 진 원[†] · 유 헌 창[††]

요 약

딥러닝 분산 학습에 사용되는 많은 도구 중 하나는 컨테이너 오케스트레이션 도구인 쿠버네티스에서 실행되는 큐브플로우이다. 그리고 큐브플로우에서 기본적으로 제공하는 오퍼레이터를 사용하여 텐서플로우 학습 작업을 관리할 수 있다. 하지만 파라미터 서버 아키텍처 기반의 딥러닝 분산 학습 작업을 고려할 때 기존의 오퍼레이터가 사용하는 스케줄링 정책은 분산학습 작업의 태스크 친화도를 고려하지 않으며 자원을 동적으로 할당하거나 해제하는 기능을 제공하지 않는다. 이는 작업의 완료 시간이 오래 걸리거나 낮은 자원 활용률로 이어질 수 있다. 따라서 본 논문에서는 작업의 완료 시간을 단축시키고 자원 활용률을 높이기 위해 딥러닝 분산 학습 작업을 효율적으로 스케줄링하는 새로운 오퍼레이터를 제안한다. 기존 오퍼레이터를 수정하여 새로운 오퍼레이터를 구현하고 성능 평가를 위한 실험을 수행한 결과, 제안한 스케줄링 정책은 평균 작업 완료 시간 감소율을 최대 84%, 평균 CPU 활용 증가율을 최대 92%까지 향상시킬 수 있음을 보여준다.

키워드 : 큐브플로우, 쿠버네티스, 딥러닝 분산 학습, 자원 조정 오퍼레이터

1. Introduction

The process of designing a deep learning model takes a lot of time. Therefore, it is very important to accelerate the training process of the model. To minimize the training time of the deep learning model, distributed deep learning training that utilizes the computing resources of multiple nodes is required [1]. Each of these nodes has a deep learning training (DLT) job consisting of several tasks executed in parallel.

This parallelism can be achieved with two kinds of methods, model parallelism and data parallelism. Model parallelism is a parallel processing method where input data is loaded evenly in multiple nodes. A large-scale training model is first divided and loaded, which then performs the training. On the other hand, data parallelism is a parallel processing method where the training model is loaded evenly on each node, large-scale input data is divided and

※ 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학ICT연구센터 지원사업의 연구결과로 수행되었음(IITP-2018-0-01405).
※ 이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(IITP-2019-0-01343).
† 준 회 원 : 고려대학교 컴퓨터학과 석사
†† 종신회원 : 고려대학교 컴퓨터학과 교수
Manuscript Received : December 31, 2021
First Revision : February 25, 2022
Accepted : March 18, 2022
* Corresponding Author : Heonchang Yu(yuhc@korea.ac.kr)

loaded, and then training is performed. Some studies [2-4] have achieved very high scalability efficiencies using data parallelism approaches, indicating that the communication overhead is not excessive. Nonetheless, some data parallelism applications may experience excessive communication overhead between devices, which seriously hinders the scalability of large model training [5]. An alternative to this is model parallelism, which is mainly used when the model is too large to fit in the memory of a single device, and data parallelism cannot be used. In general, an effective way to accelerate the training of large-scale deep learning models is to use data parallelism [6]. The main representative approach in data parallelism is the parameter server approach [7].

In a parameter server architecture, DLT job is divided into worker tasks that train the model and parameter server tasks that maintain the globally shared parameters. Each DLT job may have different requirements for different resources. In other words, while processing many DLT jobs in multiple time units, the demand for resources will have high volatility. If a job with high resource requirements is scheduled first, other jobs cannot be scheduled at the same time, which may decrease the overall resource utilization of the cluster. This may lead to a situation where the job completion time (JCT) of DLT jobs in each node increases. Therefore, when considering the minimized JCT for each DLT job in the cluster, a scheduling strategy that considers dynamic resource allocation and priorities is important.

One of the many tools used for distributed deep learning training is Kubeflow, which runs on Kubernetes, a container orchestration tool [8]. Kubeflow makes it simple to deploy machine learning workflows on Kubernetes. It also provides Kubernetes custom resources to simplify complex distributed training and supports various frameworks such as TensorFlow, PyTorch, and MXNet. Therefore, it is relatively easy to perform distributed deep learning training on Kubernetes using Kubeflow. To run distributed training jobs using TensorFlow, one of the most used deep learning frameworks in Kubernetes, the Kubernetes custom resource TFJob is used. And this TFJob is managed through the existing operator called tf-operator [9].

However, in the process of parameter server-based distributed deep learning training using tf-operator, Kubernetes's default scheduling policy cannot dynamically allocate or release cluster resources to jobs, so jobs cannot be scheduled in consideration of priority. In addition, since the task placement occurs without considering the task affinity of the distributed training job, communication overhead occurs and the JCT may be delayed or suffer from low resource utilization. Therefore, this paper proposes a new operator that efficiently schedules distributed deep learning training jobs by considering job priorities and task affinity to minimize JCT and increase resource utilization.

The main contributions of the paper are summarized as follows: 1) We propose and evaluate a resource adjustment operator that extends from the existing tf-operator of Kubernetes to increase resource utilization and minimize the job completion time of distributed training jobs. 2) We implemented a dynamic scaling algorithm that considers the priority of distributed training jobs based on weights. 3) We implemented a policy to place tasks in consideration of the affinity between worker tasks and parameter server tasks in distributed training jobs.

The remainder of this paper is organized as follows. Section 2 introduces iterativeness of the training process, parameter server architecture, and Kubernetes default scheduler. Section 3 describes the scheduling strategy designed for the proposed operator. Section 4 describes the proposed operator implementation. Section 5 presents results from deep learning experiments with various models and batch sizes. Section 6 concludes the paper.

2. Related Works

2.1 Iteration for Distributed Training

Deep learning model training is generally performed iteratively due to the large size of the training dataset. In general, the entire data set is divided into equal-sized data chunks and assigned to each worker. And each data chunk contains multiple mini-batches divided into equal sizes. Here, the size of the mini-batch is called the batch size.

Iteration means the number of repetitions of training using a given batch size. In each iteration, workers perform training through forward propagation and back propagation using a set batch size [10]. Forward propagation refers to the process of sequentially calculating and storing variables from the input layer to the output layer of the training model. Backward propagation refers to a process of sequentially calculating and storing gradients of parameters from the output layer to the input layer. In this case, hyperparameters such as batch size and learning rate are values that a neural network designer must determine in advance before training, and often has a great influence on the training result. If the batch size is too large, the training speed will be slowed because there is a large amount to be processed at one time, and in some cases, it may suffer from insufficient memory. On the contrary, if the batch size is too small, that can also be a problem. The reason is that the training is relatively unstable because the parameters are updated frequently by referencing too few samples. Therefore, it is important to set the appropriate batch size. And the learning rate should be determined to be an appropriate value that is neither too large nor too small to reach the local minimum.

2.2 Parameter Server Architecture

In distributed deep learning training, the parameter server architecture is mainly used for training by distributing the large computational load of deep learning training across multiple nodes [11]. In the parameter server architecture, a distributed training job consists of tasks with two roles: a parameter server (ps) and a worker. As shown in Fig. 1, the overall training is performed by the worker who trains the divided data and the ps that sums up the training results of each worker and manages them. Each worker has the same replicated training model and receives some training data. Since each worker's model trains using different data, it consequently has different local parameters. Therefore, there must be a global parameter that integrates these local parameters. There are asynchronous and synchronous training methods to create global parameters by integrating local parameters of all workers.

The asynchronous training is a training method where each worker completes one iteration and then proceeds with the next iteration independently without waiting for other workers. Each worker synchronizes new global parameters in the following way and proceeds with the next iteration. In the forward propagation process of each worker, the model that receives the data outputs the predicted value using the initialized weights (ω^{old}). Then, the loss function finds the loss between the predicted value and the correct answer. Loss is the penalty for a bad prediction. If the model's prediction is perfect, the loss is zero. The purpose of training the model is to find a set of weights that minimizes this loss. Lastly, the gradient ($\Delta\omega^{old}$) is calculated through an optimizer function during the backward propagation process. Each worker sends the computed gradient to ps. Then, ps immediately applies the gradient received from a specific worker to the global parameter along with the learning rate (α), and then synchronizes the new global parameter (ω^{new}) only to that worker. Therefore, workers receive synchronized global parameters that are not the same as each other and proceed with the next iteration. That is, the asynchronous training supports a fast-training speed because each worker performs training independently. However, as many 'fast workers' make significant progress in training, the delayed gradients of 'slow workers' are applied to global parameters, which has a limitation of degrading training accuracy.

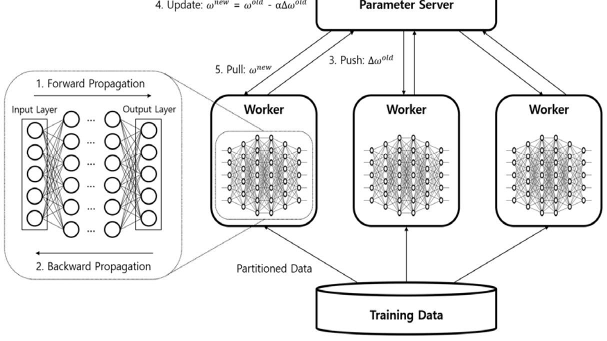


Fig. 1. Parameter Server Architecture

The synchronous training is a training method where 'fast workers' complete one iteration and then wait until the iteration of 'slow workers' is completed without independently proceeding with the next iteration. Each worker synchronizes new global parameters in the following way and proceeds with the next iteration. Each worker sends the computed gradient to ps. Then ps applies the gradients received from all workers to the global parameters in turn, and then synchronizes the new global parameters to all workers. Therefore, workers receive the same synchronized global parameters as each other and proceed with the next iteration. That is, the synchronous training can guarantee training accuracy as workers always receive the same synchronized global parameters from ps. However, there is a limitation in that the training speed decreases due to the delay caused by 'fast workers' waiting for 'slow workers' in each iteration. Due to these limitations of the synchronous training, if there many nodes, or if the jobs have different priorities or functions, the asynchronous training that prevents workers from waiting for each other may be a better choice. Therefore, this paper focuses on training the model using an asynchronous training in the parameter server architecture.

2.3 Kubernetes Scheduler

Kubernetes is a representative container orchestration

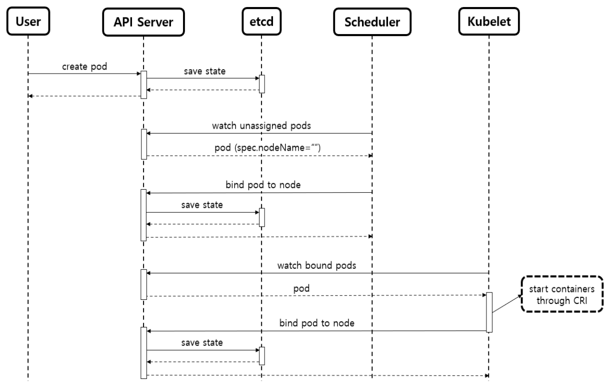


Fig. 2. Kubernetes Scheduling Lifecycle

tool for automating the deployment, scaling, and management of containerized applications. In Kubernetes, all resource allocation decisions are handled in a shared-state scheduling method, where all resource allocation decisions are made in the scheduler without a central resource allocator. One of the characteristics of Kubernetes is that containers are not deployed individually, but in units called a pod, which include one or more application containers. The Kubernetes scheduler is designed to create its own scheduling components as desired or needed and use them instead of the default scheduler [12]. That is, the pod can be scheduled in the Kubernetes cluster using the new scheduling policy. The scheduler monitors newly created pods with no node placement decisions. For every pod the scheduler finds, the scheduler is responsible for finding the best node on which that pod will run. All state information generated by scheduling is processed while recording and sharing the state in etcd, a data store. Fig. 2 shows the Kubernetes scheduler processing procedure when a new pod is assigned to a cluster.

1) When a user's pod creation request is submitted to the API Server through a command such as kubectl apply, the API Server stores the pod information in etcd. API Server is a server that provides APIs for users to access the cluster, and at the same time, it is a server that interacts with all components of Kubernetes.
2) In the pod spec, there is a nodeName field indicating the node on which the pod is scheduled. If the nodeName field is empty, the pod information is stored in etcd in an empty state.
3) The scheduler maintains and manages a FIFO-type pod queue to store pods that need to be scheduled. The newly created pod will be watched by the scheduler and added to the pod queue. After dequeuing a pod to schedule, the scheduler detects that the nodeName field of the pod is empty.
4) The scheduler finds the most suitable node for the pod after going through two major scheduling processes, predicates and priorities, based on the configuration of the pod (e.g., by metrics such as resource usage and affinity).
5) The scheduler writes the found node in the nodeName field of the pod and then sends the information to the API Server.
6) When the kubelet located in the node detects that a new pod is scheduled on the node, the local docker daemon is called to run the pod.
7) Update the status information of the pod to etcd.

The Kubernetes default scheduler, including schedulers such as YARN and Mesos used in Machine Learning (ML) systems, has a disadvantage in that it only cares about the number of resources allocated to each job without knowing the characteristics of the training job. In DLT jobs, there is no communication between worker tasks and worker tasks only communicate with the ps tasks. Therefore, the node placement of ps tasks and worker tasks can greatly affect the training speed [13]. That is, for faster training speed, the ps tasks and worker tasks of the DLT job should be scheduled on the same node. Also, the Kubernetes default scheduler is not sensitive to the affinity between ps tasks and worker tasks of DLT job during the distributed training process. Then, the tasks are scheduled in a random node in a FIFO manner without considering task affinity. Each job is assigned a fixed amount of resources based on the resource requirements specified by the job owner. In this case, cluster resources cannot be dynamically allocated or released to the job, and the ps task can become a bottleneck, slowing the job and reducing resource utilization. Therefore, to efficiently schedule a DLT job, a scheduling algorithm that considers the characteristics and the size of the job is required.

3. Proposed Scheduling Strategy

This chapter describes operator design with a scheduling strategy that minimizes JCT for concurrent DLT jobs and achieves high cluster resource utilization. The distributed training architecture of this paper is constructed based on Kubernetes. Kubeflow has an operator (tf-operator) that makes it easy to manage

DLT jobs, and Kubeflow uses Kubernetes default scheduler. When creating a new DLT job, the existing Kubeflow scheduling policy schedules each task on a random node with sufficient resources. It works well for most workloads, but not for ML workloads. Therefore, there is a need for a scheduler to help schedule ML workloads more efficiently.

Session 3.1 describes the overall cluster structure for the proposed operator, Session 3.2 describes the preemptive scheduling algorithm, and Session 3.3 describes the autoscaling algorithm that occurs during the scheduling process. The proposed scheduling algorithm provides better scheduling performance than the existing ones by combining resource allocation and task placement.

3.1 Overall Architecture

Kubernetes is a highly extensible open-source platform with a well-defined API structure. In addition to built- in resources (e.g., pods) provided by Kubernetes, resources required by users can be defined and used internally. Therefore, if a user defines a custom resource directly inside the Kubernetes system, it can be used with basic Kubernetes commands such as Kubectl. And it is possible to create and use custom controllers to manage those resources. The controller is a core concept of Kubernetes and is implemented as a software loop that runs continuously on the Kubernetes master node [14]. The controller then compares the object's current state with the desired state defined and reconciles if necessary. Objects are well-known resources such as Pod, Service, Namespace, or Volume.

The pattern of using custom resources and custom controllers together is called the operator pattern and allows users to maintain the desired state for custom resources. The operator pattern refers to a system derived from the concept of an operator in the real world. When managing their custom resources, the operator can define and handle how each resource should operate, how it should be distributed, and how to respond to problems by using the application domain. In other words, the operator acts as a custom controller for custom resources.

TFJob is a Kubernetes custom resource that enables distributed training jobs using TensorFlow on Kubeflow. That is, TFJob is designed to run distributed training jobs on Kubernetes. TFJob consists of several ps tasks and worker tasks, each deployed on a different node within the Kubernetes cluster. Each task is implemented on a container (e.g., pod) in the nodes. This paper will show that KOD2 (Kubernetes Operator for Distributed Deep learning training job), a new operator that replaces the tf-operator that previously managed TFJob, provides better scheduling performance.

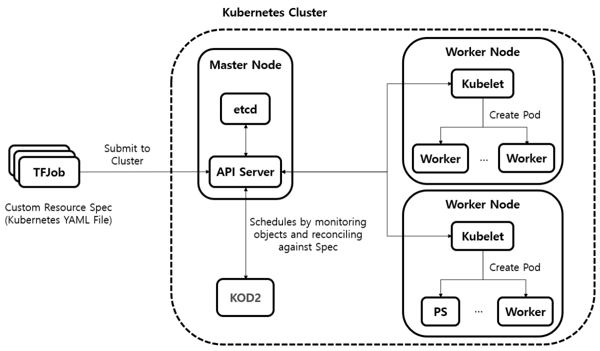


Fig. 3. Overall Architecture

The overall architecture of KOD2 running on Kubernetes is shown in Fig. 3. Just like running containerized applications, KOD2 runs outside the master node. When a TFJob where resource specs of ps tasks and worker tasks are defined is submitted to Kubernetes in the form of a YAML (YAML Ain't Markup Language) file, the API Server on the master node stores the pod allocation request status information in etcd. KOD2 recognizes that the TFJob has been created through the watch API, compares the current state with the desired state, and, if different, takes an action to reconcile to the desired state. In this process, efficient scaling of ps tasks and worker tasks and node placement decisions are made in units of pods. The operation of the internal structure of KOD2 is dealt with in section 4.2. Then, after kubelet creates a pod, the information that the pod is assigned to a specific node and is running is stored in etcd through the API server.

3.2 Priority Preemptive Scheduling

The core of KOD2 is a scheduling algorithm that minimizes JCT and increases resource utilization of the cluster. To achieve this goal, a preemptive scheduling algorithm is needed [15]. Although SJF (Shortest Job First) and SRTF (Shortest Remaining Time First), which are representative preemptive scheduling algorithms, are known to minimize JCT, there is a premise that the remaining time of a job needs to be known. In DLT jobs, it is not possible to accurately predict the remaining training time.

However, considering the two main factors that affect the training speed of a DLT job, the training time is somewhat predictable. As shown in Fig. 4 (experiment on four deep learning models when batch

Table 1. Notation of Algorithm

Notation	Description
J	number of jobs
N	number of nodes
RQ	running queue
WQ	waiting queue
w^j	replica of the worker type in job j
p^j	replica of the ps type in job j
t_j	time when job j first entered the waiting queue
ω_j	weight of job j
l_{dwj}	threshold count of workers that can be decreased in job j
l_{iwj}	threshold count of workers that can be increased in job j
c_{sij}	whether job j can be scheduled through scale-in
c_{soj}	whether job j can be scheduled through scale-out
ρ_j	node placement status for replicas in job j
ρ_{si}	scale-in node placement plan for replicas
ρ_{so}	scale-out node placement plan for replicas

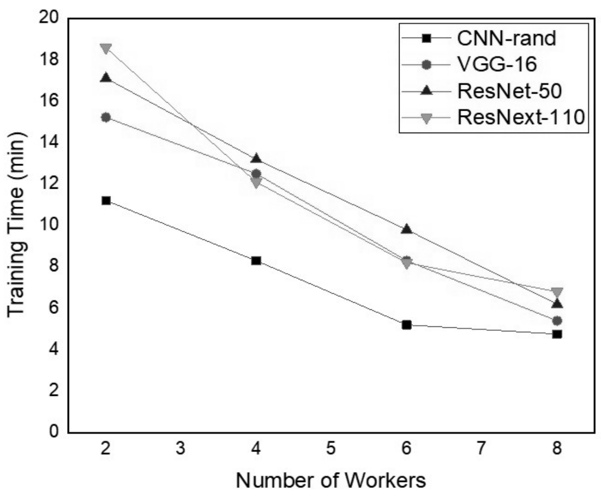


Fig. 4. Relationship between Training Time and Number of Workers

size is 256), in the parameter server architecture, when there is no bottleneck in the ps task, the training speed tends to be faster as the number of worker tasks increases. Also, in a cluster with limited resources, if one DLT job consumes too many resources, it may become a bottleneck, which may affect the training speed of other DLT jobs. If jobs that require fewer resources are scheduled first, the overall average JCT can be decreased. Since TFJob can specify the number of ps tasks and worker tasks and resource requirements, it can estimate which TFJob's JCT will be shorter by comparing the specs of TFJob.

Therefore, a heuristic priority-based preemption scheduling algorithm that gives a higher priority to a TFJob that is expected to take a shorter time can be considered. When several TFJobs enter the cluster with limited resources, applying this scheduling algorithm can lead to the effect of minimizing the average JCT and efficient utilization of resources.

Algorithm 1. Preemptive Scheduling

```
Algorithm 1 Preemptive Scheduling
Input: $j \in [J], WQ, RQ$
 1: for $j$ in $J$ do
 2:     $WQ$.enqueue($j$)
 3:     $\omega_j$ is calculated
 4: end for
 5: if $\max(\omega_j) \geq \max(\omega_{j^r})$ then
 6:     if $cluster^{idleresources} \geq$ min(|resources required by job $j$|) then
 7:         update placement of workers
 8:         $WQ$.dequeue($j$)
 9:         $RQ$.enqueue($j$)
10:     else
11:         attempt to preempt workers in $RQ$ by calling scale-in
12:         if scale-in is possible then
13:             update placement of workers
14:             $WQ$.dequeue($j$)
15:             $RQ$.enqueue($j$)
16:         end if
17:     end if
18: else
19:     $Sorted_{WQ}$ = sort jobs in $WQ$ by descending order based on the
    weight of jobs
20:     for $j^w$ in $Sorted_{WQ}$ do
21:         if $cluster^{idleresources} \geq$ min(|resources required by job $j^w$|) then
22:             update placement of workers
23:             $WQ$.dequeue($j^w$)
24:             $RQ$.enqueue($j^w$)
25:         end if
26:     end for
27:     if $j$ is waiting longer than STARVELIMIT then
28:         update $\omega_j$
29:     end if
30: end if
31: if number of workers in any running job < maximum number of workers
    then
32:     attempt to increase the number of workers in $RQ$ by calling scale-out
33:     if scale-out is possible then
34:         update placement of workers
35:     end if
36: end if
37: if all workers in the running job have finished training then
38:     $RQ$.dequeue($j^r$)
39:     $FQ$.enqueue($j^r$)
40: end if
```

Algorithm 1 is a pseudocode for the preemptive scheduling algorithm applied to KOD2. Each TFJob has information on the minimum and maximum required number of worker tasks that perform deep learning training.

In lines 1-4, when TFJobs (newJobs) with different specs enters the cluster, it is first placed in the waiting queue. After that, the weight value (ω'_j) is calculated based on the minimum required resources of the newJob. The minimum required resources refer to resources calculated by normalizing based on the number of minimal worker tasks, CPU, and memory required resources. In general, the more the number of worker tasks, the faster the training speed tends to be (see Fig. 4), and the fewer resources required, the more resources can be used by other tasks. That is, favoring TFJobs that need less resources has smaller opportunity cost since more resources are available for other TFJobs [16].

Therefore, ω'_j is calculated through a heuristic method proportion to the minimum number of worker tasks and inverse proportion to the sum of required CPU and memory resources. At this time, to normalize the weight value to be actually applied to a value between 0 and 1, the weight value (ω_j) is defined as Equation (1) in the form of a sigmoid function for ω'_j.

$$\omega_j = \frac{1}{1+e^{-\omega'_j}} \tag{1}$$

The larger the weight, the higher the priority of the TFJob, and the worker tasks of other TFJobs with relatively low priority can be preempted through scaling. In lines 5-17, it is determined whether a newJob with the largest weight in the current waiting queue will be placed first in the running queue. If the weight is greater than or equal to the TFJob with the largest weight in the running queue and there are resources more than the minimum required resources in the cluster, the newJob is placed in the running queue. If the cluster lacks resources, the scale-in function is called to preempt worker tasks of TFJobs (runJobs) running in the running queue.

If scale-in is possible, the placement status of worker tasks in the cluster is updated and the newJob is placed in the running queue thanks to the decreased worker tasks in other runJobs. In lines 27-29, if the waiting time is longer than STARVELIMIT for newJobs that are not placed in the running queue because the weight is small, the weight value is updated so that they can be scheduled first in the waiting queue. In lines 31-36, if the number of worker tasks being trained in runJob is less than the maximum number of worker tasks, the scale-out function is called to increase the number.

If scale-out is possible, the placement status of worker tasks in the cluster is updated and the number of worker tasks in the runJob is increased. In lines 37-40, when the worker task is trained as much as the maximum number of worker tasks, the runJob ends and enters the finished queue. As TFJob enters the cluster and is scheduled, worker tasks can have four states as shown in Fig. 5.

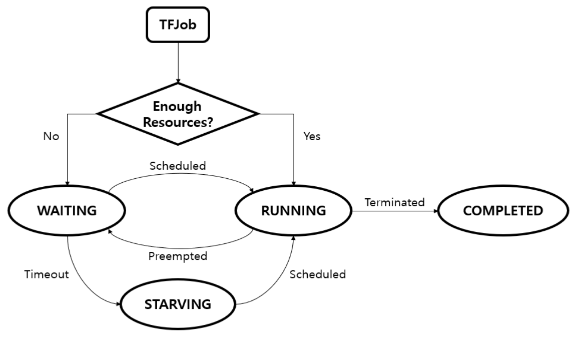


Fig. 5. State transition diagram of a TFJob

① RUNNING: When a new TFJob enters the cluster, if there are enough resources available, it is immediately enqueued in the running queue.
② WAITING: If the available resources are not enough, it waits in the waiting queue, and when the priority condition is met, worker tasks in the running queue are preempted and can be enqueued in the running queue.
③ STARVING: If the waiting time is longer than the threshold, the weight of the TFJob is updated.
④ COMPLETED: When all training is completed, the TFJob is enqueued in the finished queue.

3.3 Dynamic Auto Scaling

To accelerate job completion and make the most of the cluster's resources, the number of tasks in TFJob is dynamically adjusted using auto-scaling. There

Algorithm 2. Scale-In

```
**Algorithm 2** Scale-In
**Input:** $j \in [J], t_j, \omega_j, n \in [N], l_{dwj^r}, RQ$
**Output:** $c_{sij}, \rho_{si}$
 1: Initialize $c_{sij}$ = false
 2: Initialize $\rho_{si}$ = null
 3: $Sorted_{RQ}$ = sort jobs in $RQ$ by ascending order based on the weight of
    jobs
 4: for $j^r$ in $Sorted_{RQ}$ do
 5:     if $\omega_j == \omega_{j^r}$ and $t_{j^r}$.before($t_j$) then
 6:         $Sorted_{RQ}$.dequeue($j^r$)
 7:     end if
 8: end for
 9: for $j^r$ in $Sorted_{RQ}$ do
10:     Initialize scaleinCount = 0
11:     Initialize stop = false
12:     $Sorted_N$ = sort nodes based on where $p^{j^r}$ does not exist
13:     for $n$ in $Sorted_N$ do
14:         for $w^{j^r}$ in $n$ do
15:             if $node_n^{idleresources} \geq$ min(|resources required by job $j$|) then
16:                 $c_{sij}$ = true
17:                 return $c_{sij}$, $\rho_{si}$
18:             end if
19:             if scaleinCount $\geq l_{dwj^r}$ then
20:                 stop = ture
21:                 break
22:             end if
23:             $node_n^{idleresources}$ += |resources required by $w^{j^r}$|
24:             update $\rho_{si}$ by $\rho_{j^r}$.remove($w^{j^r}$)
25:             scaleinCount++
26:         end for
27:         if stop then
28:             break
29:         end if
30:     end for
31: end for
32: return $c_{sij}, \rho_{si}$
```

are maximum and minimum values for the number of tasks in the auto-scaling setting. The scale-in, scale-out algorithm that applies the scaling plan to the cluster when the corresponding condition is satisfied by performing the scaling test will be described.

The scale-in function is called when decreasing the number of worker tasks of the TFJob being trained. When the worker tasks of each TFJobs in the cluster are being trained, there will be the TFJob with the largest weight among them. At this time, when a TFJob (newJob) with the same weight or greater weight comes in, the scale-in function is called if the cluster has insufficient resources.

In line 3, running TFJobs (runJobs) are first sorted in ascending order by the weight value. Then, the scale-in test is performed starting with the worker tasks of the runJob with the smallest weight. In lines 4-8, if newJob and runJob have the same weight, but runJob first entered the waiting queue before newJob, worker tasks of this runJob are excluded from the scale-in test. The ps tasks and worker tasks of runJob selected as the scale-in test target will already be placed on different nodes in the cluster. In lines 12-13, search the nodes to which rubJob's worker tasks belong. At this time, to prepare for the case where the ps task and the worker task belong to the same node, the node with the ps task is searched last. This is because the more the ps task and the worker task are on the same node, the less communication overhead between them. Also, ps tasks are excluded from the scale-in test. In lines 15-18, if the node to which rubJob's worker task belongs has idle resources to satisfy the minimum required resource of newJob, it returns a schedulable state and scale-in placement plan. And the scale-in function is terminated. In lines 19-23, if there are no idle resources in the node, one worker task of runJob is decreased within the threshold count. At this time, the number of worker tasks in runJob is guaranteed to maintain at least the minimum number of worker tasks. This is to prevent a certain degree of starvation by allowing the minimum worker tasks to train. As one worker task is decreased, the idle resources of the node increase. In line 24, update the scale-in placement plan of the worker task in the runJob. In line 32, if there are no nodes with sufficient idle resources, a state that scheduling is impossible is returned and the scale-in function is terminated. When scheduling of newJob is possible through scale-in, the scale-in placement plan is applied to the cluster.

Algorithm 3. Scale-Out

```
Algorithm 3 Scale-Out
Input: $n \in [N], l_{iwj^r}, RQ$
Output: $c_{soj^r}, \rho_{so}$
 1: Initialize $c_{soj^r}$ = false
 2: Initialize $\rho_{so}$ = null
 3: $Sorted_{RQ}$ = sort jobs in $RQ$ by descending order based on the weight
    of jobs
 4: for $j^r$ in $Sorted_{RQ}$ do
 5:     Initialize scaleoutCount = 0
 6:     Initialize stop = false
 7:     for $n$ in $N$ do
 8:         while $node_n^{idleresources} >$ |resources required by $w^{j^r}$| do
 9:             if scaleoutCount $\geq l_{iwj^r}$ then
10:                 stop = ture
11:                 break
12:             end if
13:             $node_n^{idleresources}$ −= |resources required by $w^{j^r}$|
14:             $c_{soj^r}$ = true
15:             update $\rho_{so}$ by $\rho_{j^r}$.add($w^{j^r}$)
16:             scaleoutCount++
17:         end while
18:         if stop then
19:             break
20:         end if
21:     end for
22: end for
23: return $c_{soj^r}, \rho_{so}$
```

The scale-out function is called when increasing the number of worker tasks of the TFJob being trained. When the worker tasks of each TFJobs in the cluster are being trained, if the number of worker tasks being trained is less than the maximum number of worker tasks, the scale-out function is called.

In line 3, runJobs are first sorted in descending order by the weight value. Then, the scale-out test is performed starting with the worker tasks of the runJob with the largest weight. And search all nodes that can accommodate the scale-out of worker tasks in runJob. In lines 7-8, if the idle resources of the node cannot satisfy the required resources of one worker task in runJob, the next node is searched. In lines 9-12, the scale-out test of the worker task in the runJob is performed on the node within the threshold count. At this time, the worker task cannot increase more than the maximum number of worker tasks specified in runJob. If there are idle resources of the node, one worker task is increased and the idle resources are decreased. In line 15, update the scale-out placement plan of the worker task in the runJob. In line 23, it returns whether scale-out of the worker tasks in runJob is possible and the scale-out placement plan and terminates the scale-out function. When scheduling of runJob is possible through scale-out, the scale-out placement plan is applied to the cluster.

4. Implementation

This chapter introduces the implementation of KOD2. Section 4.1 describes the task placement that occurs in the distributed deep learning training process, and section 4.2 describes the KOD2 structure that manages TFJob in a Kubernetes cluster.

4.1 Distributed Deep Learning Tasks Placement

In a distributed computing environment, the main reason for slow operations in TensorFlow is the overhead of gRPC communication between ps tasks and worker tasks. To reduce this overhead, ps tasks and worker tasks should be placed using the fewest number of nodes. That is, the training speed can be improved by reducing the time required for parameter exchange between the worker task and the ps task. The Kubernetes default scheduler's FIFO-based random scheduling algorithm can incur significant communication and synchronization overhead. Therefore, KOD2 tries to place all TFJob's tasks on as few nodes as possible to minimize this overhead. The key point is to search for nodes with idle resources that can accommodate all tasks of TFJob. If all tasks of TFJob cannot be placed together on one node, it tries to place ps tasks and worker tasks together in order of nodes with the idlest resources. When it is decided on which node to place the tasks in KOD2, the node is specified in the nodeName field of the pod spec. If a node is explicitly specified in the nodeName field in advance, it can be scheduled to that node without having to undergo a separate scheduling process in the Kubernetes default scheduler.

4.2 Operator for Preemptive Scheduling

The master node manages the records of all objects in the cluster and stores the state desired by the user in etcd. Sync loop is performed in the operator to manage the state of the object through the records. Sync loop means that whenever there is a change in the internal state of the cluster, it is detected and if the user's desired state does not match the current state, it operates to match it. That is, the main task of KOD2 is to match the desired state and current state of the Object (TFJob). This reconcile process is performed by the reconciler. Fig. 6 shows the internal operating structure of KOD2.

If an object creation request is received from the user, the Kubernetes API server stores the object information in etcd and delivers the object event to KOD2. In KOD2, there are Kubernetes cache and work

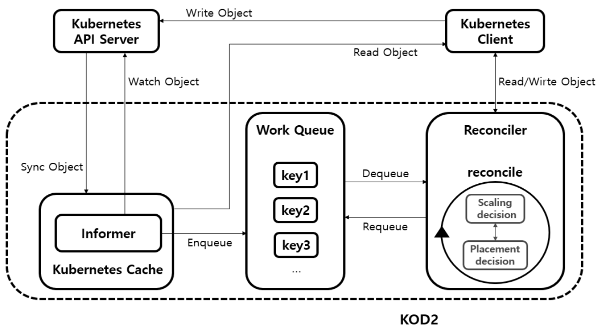


Fig. 6. KOD2 Architecture

queue. The cache serves to reduce the load on the API server by caching the information retrieved from the API server. And the informer exists inside the cache. Informer is responsible for monitoring objects to be managed by KOD2 and receiving object creation, change, and deletion events. Only the object name and namespace information, which are the object event information received by the informer, are extracted and enqueued in the work queue. In the work queue, there are keys containing the name and namespace information of each object. Reconciler uses the Kubernetes client to perform reconcile. The client's object write request is delivered directly to the API server, but the client's object read request is delivered to the cache, not the API server. This is to read data from the cache that exists inside KOD2 to prevent overload of API Server. Reconciler dequeues the name and namespace information of each object stored in the work queue. Then, the reconcile process of each object is performed using the client. In this process, scaling and placement of each object are determined using Algorithm1. The reconciler discards the object information in the case of an object that has succeeded in reconciling. However, in the case of an object that the reconciler has failed to reconcile, the name and namespace information of the object is requeued. After a specific time has passed, the reconciler tries to reconcile again by dequeueing the object information that has failed. If object reconciliation fails again, it is requeued and this process is repeated until reconciliation succeeds.

5. Evaluation

This chapter shows the performance evaluation of the proposed operator (KOD2) compared to the existing operator (tf-operator) in a real environment. Section 5.1 describes the experimental procedure and section 5.2 describes the experimental results.

5.1 Experimental Procedure

For the experiment, we built a small cluster of Kubernetes version 1.20.5 and Kubeflow version 1.0.0 consisting of 1 master node and 3 worker nodes as shown in Table 2. Each node is equipped with a 10-cores CPU (Intel Core i9-10900K), 32 GB of RAM, and 1 GTX 3090 GPU with 24 GB of device memory.

We deployed tf-operator and KOD2 in a Kubernetes cluster to schedule each training job. The TFJob used in the experiment performed the task of training the CIFAR-10 dataset using four CNN models, CNN-rand [17], VGG-16 [18], ResNet-50 [19], and ResNext-110 [20]. Training job configuration is shown in Table 3. TFJob consists of one ps task and several worker tasks, and the worker task can be scaled-in or scale- out based on the calculated weight. Also, the resource requirements of each worker task are different. One TFJob among 10 TFJobs was submitted to the cluster every 30 seconds, and average JCT and resource utilization were measured and compared when using KOD2 and using tf-operator. JCT means the elapsed time from when the TFJob enters the waiting queue until it enters the finished queue. Since TFJobs are entered in units of time, if there are not enough resources in the cluster, the worker tasks of TFJobs with smaller weight can be preempted after comparing their weight with each other.

5.2 Experimental Results

Fig. 7 shows the comparison of average JCT values of 10 TFJobs measured by each operator according to model and batch size. The tf-operator uses the Kubernetes default scheduler, so the tasks are placed on random nodes without considering task affinity.

In addition, even if there are insufficient resources in the cluster since tasks are continuously scheduled by the FIFO scheduling algorithm, these tasks fall into a state of continuously requesting resources. For this reason, the JCT of TFJob is slowed down, and some tasks fall into a deadlock state and are terminated without completion. On the other hand, KOD2 performed in a stable state because it scheduled tasks only when there were sufficient resources and performed efficient scaling and placement. Therefore, as shown in Fig. 7, the average JCT is relatively faster for all models when using KOD2 than tf-operator. Among them, when the batch size of Fig. 7(b) is 128, the time reduction rate is 84%, showing the highest performance.

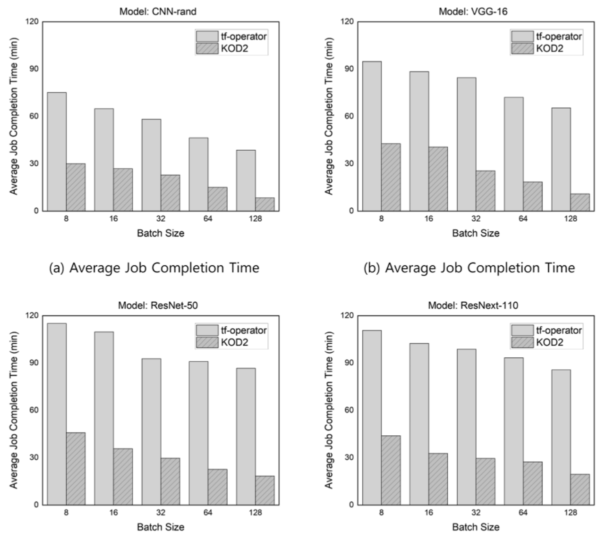
Fig. 8 shows the average CPU utilization of 3 worker nodes according to the model when the batch size is

Table 2. Cluster Configuration

Kubernetes Version	v1.20.5
Kubeflow Version	v1.0.0
OS	Ubuntu 18.04
CPU Cores	10
CPU Type	i9-10900K
Memory	32GB
GPU Type	GTX 3090
Device Memory	24GB
Num of Master Node	1
Num of Worker Node	3

Table 3. Training Job Configuration

Model	CNN-rand, VGG-16, ResNet-50, ResNext-110
Network Type	CNN
Application Domain	Image Cassification
Dataset	CIFAR-10
Dataset Size	60,000
Batch Size	8, 16, 32, 64, 128
Learning Rate	0.01



(a) Average Job Completion Time (b) Average Job Completion Time

(c) Average Job Completion Time (d) Average Job Completion Time

Fig. 7. Average Job Completion Time

64. CPU utilization was measured using Prometheus [21] and Grafana [22]. Since the TFJob enters every 30 seconds, the initial average CPU utilization may be small because the job is performed on only one of the three nodes. After that, the CPU utilization of the three nodes increases as the TFJob continues to enter.

When using tf-operator, since tasks are scheduled even when there are insufficient resources in the cluster, some tasks fall into a deadlock state, indicating that each node cannot utilize resources properly. In the case of Fig. 8(b), (c) and (d), there is a section show-

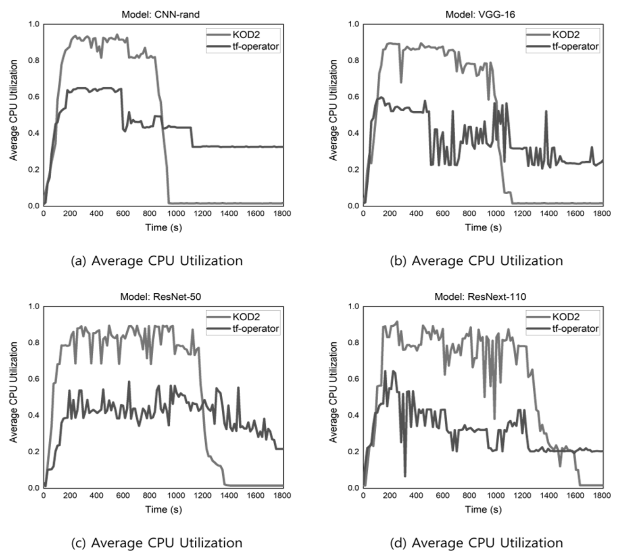


Fig. 8. Average CPU Utilization

ing the CPU utilization rate increases rapidly, unlike the rest of the cases. In this case, it seems that due to some TFJob that is terminated without completion, idle resources are created and the utilization has increased rapidly. However, this utilization cannot be maintained for a long time and it can be seen that the utilization is lowered again. And in the case of each model, the entire work was finished after about 30 minutes to over an hour in addition to the time shown in the Figure. On the other hand, when using KOD2, Fig. 8(b) shows the average CPU utilization increase rate of up to 92%. In other words, the result of KOD2 shows that each node utilizes resources better than the case of tf-operator for each model.

6. Conclusion

KOD2 is an operator proposed to manage TFJob more efficiently than existing operator in Kubernetes Cluster. Existing operator use scheduling policy that cannot allocate resources dynamically and do not consider the task affinity of the training jobs, which can result in longer job completion time and lower resource utilization. In KOD2, a weight-based preemption scheduling algorithm was applied by reflecting the job priority. A high priority is given to a TFJob that is heuristically predicted that the job will be completed faster based on resource usage and the number of worker tasks. If there are not enough resources in the cluster, but the priority of the waiting TFJob is higher than that of the running TFJob, task preemption occurs through auto-scaling. In addition, the task placement of the training job in consideration of task affinity was also made. As a result, when using KOD2 through an experiment, it showed the effect of minimizing the average JCT and leading to a higher resource utilization than before.

References

[1] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," In *ACM Computing Surveys (CSUR)*, Vol.52, No.4, pp.1-43, 2019.

[2] P. Goyal, et al., "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[3] X. Jia, et al., "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[4] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng, "Image classification at supercomputer scale," *arXiv preprint arXiv: 1811.06992*, 2018.

[5] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[6] C. J. Shallue, J. Lee, J. Antognini, J. Sohl-Dickstein, R. Frostig, and G. E. Dahl. "Measuring the effects of data parallelism on neural network training," *arXiv:1811.03600*, 2018.

[7] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D.G. Andersen, and A. J. Smola. "Parameter server for distributed machine learning," In *Big Learning NIPS Workshop*, 2013.

[8] Kubeflow 2021, accessed 1 September 2021 [Internet], https://www. kubeflow.org.

[9] TensorFlow Operator 2021, accessed 1 September 2021 [Internet], https://www.kubeflow.org/docs/components/training/tftraining/#installing-tensorflow-operator.

[10] S. Li, et al., "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv: 2006.15704*, 2020.

[11] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[12] Kubernetes 2021, accessed 1 September 2021 [Internet], https:// kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler.

[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. "Optimus: An efficient dynamic resource scheduler for deep learning clusters," In *Proceedings of ACM EuroSys*, 2018.

[14] Operator 2021, accessed 1 September 2021 [Internet], https://cloud. redhat.com/learn/topics/operators.

[15] J. Gu, "Tiresias: A GPU cluster manager for distributed deep learning," In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

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

[17] Y. Chen, "Convolutional neural network for sentence classification," MS thesis, University of Waterloo, 2015.

[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[20] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[21] Prometheus 2021, accessed 1 September 2021 [Internet], https://prometheus.io.

[22] Grafana 2021, accessed 1 September 2021 [Internet], https://grafana.com

[23] J. Geng, D. Li, and S. Wang. "Accelerating distributed machine learning by smart parameter server," In *Proceedings of 3rd Asia-Pacific Workshop Networking*, 2019.

[24] E. Gebremeskel, "Analysis and comparison of distributed training techniques for deep neural networks in a dynamic environment," 2018.

[25] W. Xiao, "Gandiva: Introspective cluster scheduling for deep learning," In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[26] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, 2018.

[27] M. Khalil-Hani and S. Liew, "A-sdlm: an asynchronous stochastic learning algorithm for fast distributed learning," In *13th Australasian Symposium on Parallel and Distributed Computing*, 2015.

Jinwon Jeong
https://orcid.org/0000-0002-6882-6074
e-mail : jin4812@korea.ac.kr

He received a M.S. degree in computer science and engineering from Korea University, Seoul, Korea, in 2022. His research interests include cloud computing, distributed computing, distributed deep learning.

Heonchang Yu
https://orcid.org/0000-0003-2216-595X
e-mail : yuhc@korea.ac.kr

He received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Korea University, Seoul, Korea, in 1989, 1991, and 1994, respectively. He has been a Professor of computer science and engineering with Korea University since 1998. From January 2015 to December 2020, he was the Vice President of Korea Information Processing Society, Korea. His research interests include cloud computing, virtualization, and distributed computing.