EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

A combined priority scheduling method for distributed machine learning

TianTian Du[1], GongYi Xiao[1], Jing Chen[1*], ChuanFu Zhang[1], Hao Sun[1], Wen Li[1] and YuDong Geng[1]

*Correspondence:
jingchen94@163.com

[1] Shandong Provincial Key Laboratory of Computer Networks, Shandong Computer Science Center (National Supercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences), Jinan, China

Abstract

Algorithms and frameworks for distributed machine learning have been widely used in numerous artificial intelligence engineering applications. A cloud platform provides a large number of resources at a lower cost and is a more convenient method for such applications. With the rapid development of containerization, native cloud combinations based on Docker and Kubernetes have provided effective resource support for distributed machine learning. However, native Kubernetes does not provide efficient priority or fair resource scheduling strategies for distributed machine learning in computationally intensive and time-consuming jobs, which easily leads to resource deadlock, resource waste, and low job execution efficiency. Therefore, to utilize the execution order between multiple jobs in distributed machine learning as well as the dependencies between multiple tasks for the same job, considering intra- and inter-group scheduling priorities, a combined priority scheduling method is proposed for distributed machine learning based on Kubernetes and Volcano. Considering the user priority, task priority, longest wait time, task parallelism, and affinity and non-affinity between the parameter server and worker nodes, a combined priority scheduling model of inter- and intra-job priority is proposed, which is mapped into a scheduling strategy of inter- and intra-group priorities of pods, enabling the efficient scheduling and training of distributed machine learning. The experiment results show that the proposed method achieves preferential resource allocation for urgent, high parallelism, and high-priority jobs with high-priority users and improves the job execution efficiency. The affinity and anti-affinity settings among pods reduce the time of information interaction between the parameter server and worker nodes to a certain extent, thereby improving the job completion efficiency. This group scheduling strategy alleviates the problems of resource deadlock and waste caused by insufficient resources in cloud computing.

Keywords: Cloud computing, Distributed machine learning, Resource scheduling, Prioritization

1 Introduction

Machine learning generally involves a large number of iterative computations, requiring a large amount of resources to perform jobs or handle data [1–3]. The abundance of cloud computing resources promote machine learning services on the cloud platform. An efficient resource scheduling method for cloud computing is the key to ensuring the

execution efficiency of distributed machine learning. A cloud computing service model has been proposed for the on-demand scheduling and allocation of distributed computing resources and storage space [4, 5]. Resource management and scheduling methods for cloud computing have also been studied to improve the system efficiency [6–8]. The resource allocation process is divided into three parts, i.e., resource scheduling, resource mapping, and resource adjustment, and a new resource allocation management framework has been proposed [9]. Distributed machine learning can be classified into data [10] and model [11] parallel modes. Massive resources are scheduled in parallel for training a model based on large-scale data, improving the performance and privacy protection of large-scale data training models [12]. A dynamic trend prediction ant colony algorithm was proposed to effectively solve the problems of network space occupation and a slow response time [13]. To improve the resource utilization and service quality of cloud computing, a dynamic resource scheduling system based on particle swarm optimization (PSO) and a radial basis neural network (RBF) was designed [14]. Some effective virtual scheduling schemes have also been proposed to reduce the response time [15, 16]. Using k-means resource requirements to predict future tasks, a multi-objective resource scheduling algorithm was proposed to minimize the cost of virtual machines [17]. A cache-aware scheduling model was also proposed based on a neighborhood search, in which jobs are scheduled for nodes with similar capabilities to reduce the job execution time [18]. In addition, an elastic cluster and scheduling strategy was optimized to improve the resource utilization and reduce the resource costs while ensuring the quality of service (QoS) [19]. The rough set theory has been used to solve the load balance problem among servers and improve the overall performance of the cluster [20].

A max–min fair scheduling algorithm was proposed to schedule different types of resources [21]. An efficient priority task scheduling algorithm was designed to satisfy the different priorities and QoS requirements of users [22]. In addition, an efficient smallest-height-first dominant resource fairness scheduling algorithm was proposed to guarantee near-optimal scheduling and isolation without prior knowledge of the coflow size using the smallest-height-first and the monopolistic dominant resource fairness bandwidth allocation strategy [23]. A task scheduling method based on artificial bee foraging optimization that considers four QoS metrics was proposed [24]. A method applying a computational efficiency analysis and embedded vectors was proposed to solve the excessively high user data dimensions caused by mean shift clustering [25]. Furthermore, a leader election algorithm was proposed for improving the scalability of applications within containers and achieving a dynamic resource allocation. This algorithm selects leaders from various nodes in a cluster and manages them more effectively through leaders [26]. To achieve load balancing and solve the resource allocation issues in a cluster, leader election algorithms and leader-based consistency maintenance mechanisms have been analyzed, demonstrating the importance of leader distribution across nodes in a cluster for highly scalable load balancing [27].

There is an execution order between multiple tasks in distributed machine learning, and a certain degree of dependency exists between multiple tasks in the same job. To solve the problems of resource deadlock, resource waste, and low job execution efficiency caused by computationally intensive and time-consuming jobs without efficient priority and fair resource scheduling strategies, intra- and inter-group scheduling priorities should be considered to effectively cope with resource competition. To address these issues, a combination priority scheduling method for distributed machine learning is proposed herein.

The main contributions of this paper are summarized as follows.

(1) This study examines the factors influencing job and task execution priorities, including user priority, job priority, maximum wait time, task parallelism, and affinity and anti-affinity, to comprehensively design the model training of distributed machine learning from the job and task levels.
(2) Inter- and intra-group job priorities are constructed based on factors influencing the job priority.
(3) A combined priority scheduling method is proposed that not only realizes the prior scheduling of jobs and tasks by implementing inter- and intra-group priority scheduling strategies, but it also solves the problem of resource deadlock and resource waste in distributed machine learning by implementing the pod group scheduling strategy. Furthermore, the combined priority scheduling method improves the efficiency of resource allocation and shortens the job completion time.

The remainder of this paper is organized as follows. Section 2 describes the related studies. Section 3 presents the proposed combined priority scheduling method. Section 4 discusses the experiment results. Finally, Sect. 5 concludes the study.

2 Related work

Kubernetes includes numerous resource scheduling methods. A Kubernetes container scheduling system is proposed that considers central processing unit (CPU) utilization, memory utilization, disk utilization, power consumption, the time required for selecting images, and the number of running containers [28]. A scheduling strategy based on the Docker cluster of a self-defined Kubernetes scheduler was proposed to improve the cluster scheduling fairness [29]. A two-level scheduling framework, i.e., Kubernetes-on-EGO (a microservice-oriented management framework developed by Golang), combines Kubernetes and EGO scheduling systems for improving the resource utilization and scheduling efficiency [30]. A cloud computing resource management method based on container technology was proposed to balance the use of the overall resources [31]. A progress based on the ProCon (a progress-based container placement scheme) container placement scheme (a progress-based container placement scheme) considers the real-time and predicted future resource utilization for balancing the resource competition in a cluster and reducing the completion time [32]. A new preemptive scheduling strategy was proposed to solve the problem of a high scheduling

failure rate without sufficient resources [33]. To solve the problem of Kubernetes killing and discarding important low-priority containers, a scheduling strategy was proposed based on the run level obtained through the resource utilization and runtime [34]. A master–slave scheduler was proposed to solve the performance problem of a large-scale cluster scheduling system [35]. A scheduler was proposed to monitor the requirements and attributes of containers based on multiple prediction models and a scheduling framework that is more conducive for container management [36]. An edge scheduler was proposed to achieve low-latency resource allocation. The scheduler can schedule resources globally, significantly reducing the scheduling latency [37]. An effective controller was proposed to reduce the energy consumption of a cluster. The controller establishes a multi-objective function by considering such factors as the carbon footprint, interference, and emissions that occur through energy consumption. The best scheme was selected based on multiple objective functions used to achieve energy-saving resource allocation [38].

When a cloud platform system adopts a priority scheduling method, the system first allocates resources to the highest priority task. There are two classes of priority scheduling methods. The first class is nonpreemptive priority scheduling in which the highest priority process allocates resources until it is implemented [39]. The second is preemptive priority scheduling in which the process with the highest priority is first allocated to the resource node that is interrupted, and resources are then allocated to other processes with a higher priority during the running process. The rationale for the priority scheduling approach is the assignment of different priorities to each process, and the process with the highest priority preferentially allocates resources. Priority scheduling can prioritize important processes, ensuring that important tasks obtain their requested resources first [40]. Priority scheduling can be classified into dynamic and static priority scheduling. The static priority remains constant during the process, which is simple and flexible, with a low system overhead. In addition, the priority setting is determined based on the scenario. Different influencing factors should be considered when prioritizing resource scheduling within a specific environment.

3 Methods

3.1 Factors impacting priority

Suppose that a distributed machine learning job to be scheduled in a cluster queue is denoted as $J = \{J_1, ..., J_i, ..., J_n\}$, where the i th job J_i contains one or more tasks $\{t_{i1}, t_{i2}, \ldots, t_{ij}, \ldots, t_{im}\}$ and task t_{ij} is the j th task of the i th job J_i. Job J_i cannot run normally until sufficient resources are successfully created for all tasks $\{t_{i1}, t_{i2}, \ldots, t_{ij}, \ldots, t_{im}\}$ of this job. These tasks can be executed on one or more pods as a pod group $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ij}, \ldots, p_{im}\}$ in a Kubernetes cluster. In this paper, task t_{ij} runs on pod p_{ij}. A Kubernetes cloud platform receives a queue of distributed machine learning jobs from different users, which require different resources determined by multiple impact factors including the user priority, job priority, job urgency, job parallelism, and affinity and anti-affinity.

(1) User priority

If a Kubernetes cloud platform receives a queue of distributed machine learning jobs $J = \langle J_1, J_i, J_n \rangle$, these jobs come from multiple users with different priorities. Job J_i comes from the i th user, and the user priority is $u_i(J_i)$. The following formula defines the user priority $U(J)$ for job queue J.

$$U(J) = \{u_1(J_1), \ldots, u_i(J_i), \ldots, u_n(J_n)\} \tag{1}$$

The user priority $u_i(J_i)$ has a relation $u_i(J_i) \in N$, where N^* is a positive integer.

(2) Job priority

Different jobs running on a Kubernetes cloud platform have different job priorities, as described in the following.

$$B(J) = \left\{b_1(J_1), \ldots, b_i(J_i), \ldots, b_n(J_n)\right\}, \tag{2}$$

where $B(J)$ is a priority of job queue J. Job priority $b_i(J_i)$ is related to $b_i(J_i) \in M^*$, where M^* is a positive integer.

(3) Maximum wait time

User jobs have different degrees of urgency, which determine the longest wait times of the jobs. Suppose that the maximum wait time $L(J)$ of the job queue is defined as follows:

$$L(J) = \{L_1(J_1), \ldots, L_i(J_i), \ldots, L_n(J_n)\} \tag{3}$$

There exists a relation $L_i(J_i) \in Q^*$, where Q^* is a positive integer within a certain range.

(4) Task parallelism

Distributed machine learning J_i contains multiple tasks that are executed using a parameter server architecture. This architecture involves a parameter server and a worker node. The worker nodes conduct training tasks and push the trained model parameters to the parameter nodes. The parameter servers store and update the model parameters. The worker nodes pull the updated model parameters and continue with the subsequent iterative training. When the distributed machine learning model involves massive parameters, model training requires a large number of work nodes to conduct tasks in parallel. The task parallelism $M(J)$ of the job queue J can be denoted as the number of worker nodes for each job J_i.

$$M(J) = \{m_1(J_1), \ldots, m_i(J_i), \ldots, m_n(J_n)\} \tag{4}$$

There exists a relation $m_i(J_i) \in X^*$, where X^* is a bounded positive integer.

(5) Affinity and anti-affinity assays

Numerous data interactions occur between the worker and parameter nodes when conducting job J_i using the TensorFlow parameter server architecture. The placement of the parameter server and worker nodes on the same server reduces the transmission time loss between them and improves the job execution efficiency. When possible, the same type of node should not be placed on the same server. Affinity and anti-affinity relationships exist among parameter server or worker nodes. The affinity and anti-affinity among the parameter server and work nodes were therefore set as the two factors affecting the resource scheduling. Suppose that the worker nodes running job J_i are represented by the following expression $w_{i1}, w_{i2}, \ldots, w_{ij}, \ldots, w_{im}$. The parameter nodes are denoted as variables $s_{i1}, s_{i2}, \ldots, s_{ij}, \ldots, s_{il}$. The storage unit $B_i = \{b_{i1}, b_{i2}, \ldots, b_{ij}, \ldots, b_{ir}\}$ is set to store tasks with an affinity relationship, such as a set of tasks with an affinity relation in storage unit b_{ij}.

3.2 Combined priority scheduling method

The involved factors of the combined priority scheduling (CPS) method is shown in Fig. 1. To improve the execution efficiency of different jobs and maximize the resource utilization, this method considers the user priority, job priority, job execution parallelism, maximum wait time of the job, and affinity and anti-affinity relations among the parameter server and worker nodes in the parameter server architecture. The priority of each job is influenced by the user priority, job priority, urgency, and task parallelism.

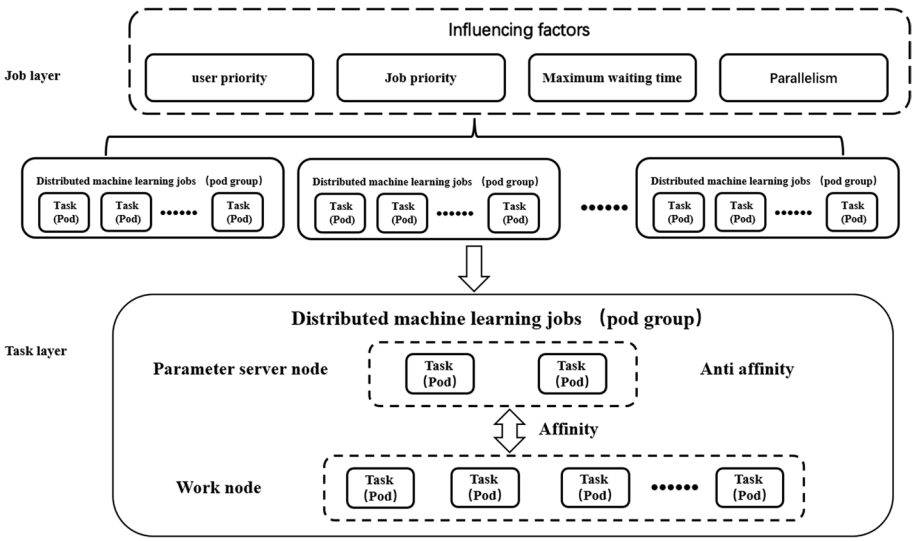

Fig. 1 Involved factors of CPS method. This method considers user priority, job priority, job execution parallelism, the maximum waiting time of a job and the affinity and anti-affinity relations among parameter server and worker nodes in the parameter server architecture to improve the execution efficiency of different jobs and maximize the resource utilization. The priority of each job is influenced by user priority, job priority, urgency, and task parallelism. Each job contains multiple tasks. The task priority is mapped to the priority of resource allocation, with anti-affinity between parameter server nodes and between parameter server and work nodes

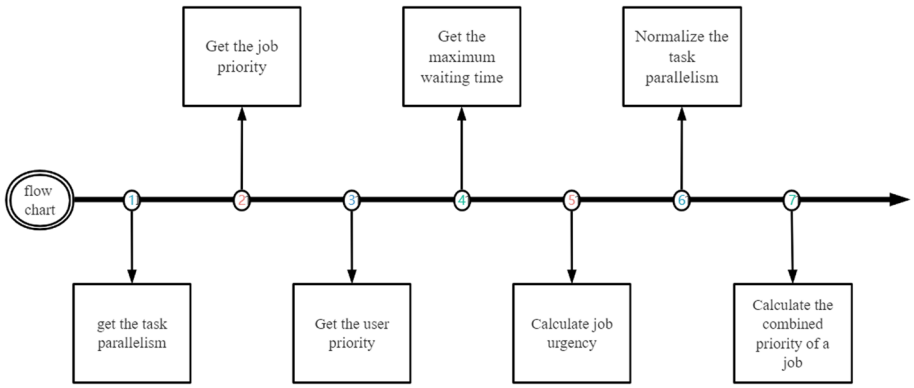

Fig. 2 Combined priority. The combined priority of the job is calculated by comprehensively considering the task parallelism, job priority, user priority, and maximum waiting time of the job

Each job is composed of multiple tasks. The task priority is mapped to the priority of resource allocation with anti-affinity between the parameter server nodes and between the parameter server and work nodes.

Suppose that a job queue of distributed machine learning is denoted as $J = \{J_1, \ldots, J_i, \ldots, J_n\}$, where the number of jobs is larger than 1, that is, $n \geq 1$. Each job J_i contains h tasks $\{t_{i1}, t_{i2}, \ldots, t_{ij}, \ldots, t_{ih}\}$, where the number of tasks is greater than 1, that is, $h \geq 1$. Assume that job J_i has user priority $u_i(J_i)$, job priority $b_i(J_i)$, maximum job wait time $L_i(J_i)$, and task parallelism $m_i(J_i)$. The specific process combining these priorities is shown in Fig. 2.

(1) Obtain the task parallelism $m_i(J_i)$ of job J_i. Job J_i includes multiple tasks, each of which will be executed on one or more worker nodes simultaneously. The task parallelism $m_i(J_i)$, that is, the intra-group priority, can be denoted as the number of worker nodes N_i as follows:

$$m_i(J_i) = N_i. \tag{5}$$

(2) Obtain job priority $b_i(J_i)$ of job J_i [41]. The job priority can be set in the YAML file as high, medium, or low for job J_i as follows:

$$b_i(J_i) = \text{choice (hp, np, lp)} \tag{6}$$

The relations $hp \in N^*$, $np \in N^*$, and $lp \in N^*$ exist in which high job priority hp, medium job priority np, and low priority lp are expressed by different values from large to small, such as $\text{hp} = 10$, $\text{np} = 5$, and $\text{lp} = 1$.

(3) Obtain user priority $u_i(J_i)$ of job J_i. Kubernetes implements multitenant isolation using different names. Different names indicate different priorities. User priorities can be set based on the weight of the namespace. A YAML file of the ResourceQuota type is created, in which a weight value ranging from 1 to 10 indicates user priority.

(4) Obtain the maximum wait time $L_i(J_i)$ of job J_i and calculate the time-constraint priority $l_i(J_i)$ based on the maximum wait time. The shorter the wait time, the higher the time-constraint priority of the job. The time-constraint priority $l_i(J_i)$ of job J_i can be calculated based on the wait time $L_i(J_i)$ using Eq. (7).

$$l_i(J_i) = \left\lceil \frac{100}{L_i(J_i)} \right\rceil \tag{7}$$

The symbol "⌈⌉" is an upward integral function. The value of this variable is limited to within the range of $1 \leq L_i(J_i) \leq 60$. The maximum wait time is 60 min, and the minimum wait time is 1 min. The time-constraint priorities are defined separately as 1 and 100 for time frames of more than 60 and 1 min, respectively. The value of the executed priority $l_i(J_i)$ is limited to the range [1,100].

(5) Calculate the urgency $e_i(J_i)$ of job J_i. The job urgency $e_i(J_i)$ is calculated based on the user priority, job priority, and time-constraint priority of job J_i, as expressed in Eq. (8). The higher the user, job, and time-constraint priorities are, the higher the job urgency. This job should be executed preferentially.

$$e_i(J_i) = u_i(J_i) + b_i(J_i) + l_i(J_i) \tag{8}$$

There exists a relation $e_i(J_i) \in N^*$, where symbol N^* is a positive integer. The set $E(J) = \{e_1(J_1), \ldots, e_i(J_i), \ldots, e_n(J_n)\}$ is the urgency of all jobs.

(6) Normalize the task parallelism $m_i(J_i)$ and job urgency $e_i(J_i)$ of job J_i. Task parallelism is normalized using the following formula, where $M(J)_{\max}$ indicates the maximum value of task parallelism $M(J)$ for all jobs, $M(J)_{\min}$ indicates the minimum value of task parallelism $M(J)$ for all jobs, and $m_i(J_i)_n$ indicates the normalized value of task parallelism $m_i(J_i)$.

$$m_i(J_i)_n = (m_i(J_i) - M(J)_{\min})/(M(J)_{\max} - M(J)_{\min}) \tag{9}$$

The job urgency $e_i(J_i)$ is normalized as shown in Eq. (9), where $M(J)_{\max}$ indicates the maximum value of job urgency $E(J)$ for all jobs, $E(J)_{\min}$ indicates the minimum value of job urgency $E(J)$ for all jobs, and $e_i(J_i)_n$ indicates the normalized value of job urgency $e_i(J_i)$.

$$e_i(J_i)_n = (e_i(J_i) - E(J)_{\min})/(E(J)_{\max} - E(J)_{\min}) \tag{10}$$

(7) Calculate the combined priority of job J_i, that is, the inter-group priority of the pod group created for a job from the job queue using Eq. (11). Specifically, V_i is the combination priority of job J_i, k' is the expansion coefficient, $e_i(J_i)_n$ indicates the normalized job urgency, and $m_i(J_i)_n$ indicates the normalized task parallelism.

$$V_i = k' \times (m_i(J_i)_n + e_i(J_i)_n) \tag{11}$$

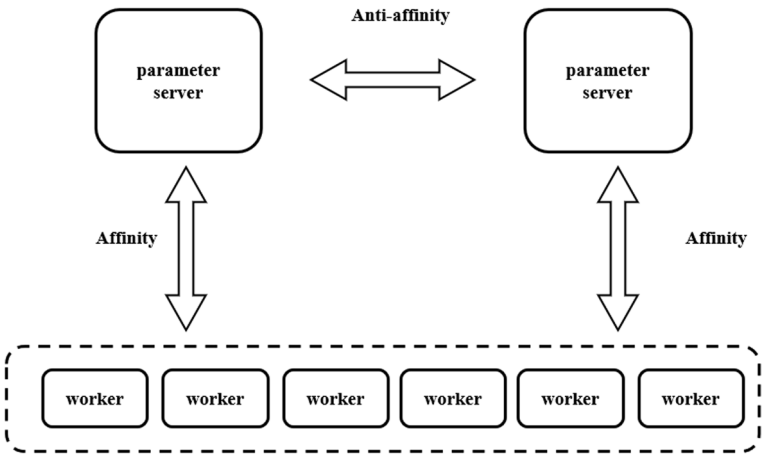

Fig. 3 Affinity and anti-affinity among pods as parameter and worker nodes. The affinity relation between the pods as parameter nodes and the pods as worker nodes and the anti-affinity relation among the pods as parameter nodes

(8) Map the combined priority of job J_i to the priority of the pod group $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ij}, \ldots, p_{ih}\}$, which corresponds to h tasks. The affinity relationships between the pods as parameter nodes and the pods as worker nodes, and the anti-affinity relationships among the pods as parameter nodes, are shown in Fig. 3.

If the priorities of the parameter server nodes p_{i1}, p_{i2} are higher than those of the worker nodes $p_{i3} \ldots p_{im}$, the resources of the parameter server nodes p_{i1}, p_{i2} are preferentially allocated to the worker nodes $p_{i3} \ldots p_{im}$. Affinity and anti-affinity relationships are not mandatory. If a server has insufficient resources for the parameter server nodes, other servers that may violate the relations can allocate resources to these parameter server nodes.

3.3 Algorithm design for CPS method

The algorithm used by the CPS method (i.e., the CPS algorithm) is designed based on the Volcano scheduling framework, which provides basic scheduling modules and algorithms through plugins and can combine various scheduling algorithms. In this study, the algorithm design of the CPS method was completed, and to implement the algorithm, a plugin module called mypriority.go was developed. The algorithm design includes two main parts: inter- and intra-group priorities. Different priorities were defined for some YAML files. The mypriority.go plugin integrated into the Volcano scheduling framework calls these priorities to implement the inter- and intra-group priorities. The main steps of intra-group priority (i.e., job priority in a job queue) are as follows:

(1) Set and obtain the user priory of a job. Kubernetes enables multitenant isolation through different names. User priority can differ using different weights for the user names. The weight indicating the user priority of a job can be set within the range of (1, 10) in a ResourceQuota type YAML file.

(2) Obtain the job priority. The job priority is set to a high, medium, or low value in a job configuration YAML file and can be obtained by calling the "jobinfo" structure in the opensession function.
(3) Obtain the maximum wait time for a job. The SLA plugin supports the user in defining the maximum wait time to preferentially schedule resources for jobs in the Volcano scheduler. A user can assign an annotation called "sla-waiting-time" to define the maximum expected wait time for a job. The queue and allocated action plugins compare the difference between the actual and maximum expected wait times. The JobOrderFn function determines the ranking of the job scheduling based on the differences in the jobs. The maximum wait time of a job is obtained directly by calling the "jobinfo" structure in the opensession function.

The time-constraint priority $l_i(J_i)$ can be calculated using Eq. (7).

(4) Obtain the job task parallelism. Task parallelism is set as the number of worker nodes in the parameter server architecture, which can be acquired by rewriting the application programming interface (API) of the scheduling controller and the correlated functions and interfaces of the scheduler cache.
(5) Obtain the affinity and anti-affinity relationships among job tasks. First, the affinity and anti-affinity relations are observed according to the topology among pods in the parameter server architecture and are defined in the job configuration YAML file.
(6) Calculate the job urgency. The job urgency is calculated according to the obtained user priority, job priority, and time-constraint priority using Eq. (8).
(7) Calculate the combined priority of a job. Job urgency and task parallelism are normalized to calculate the combined priority of a job, that is, the intra-group priority.

The inter-group priority, that is, the priority among tasks from a job, considers the affinity between parameter server pods and worker pods, and the anti-affinity among parameter server pods. The main steps are as follows:

(1) Obtain the job task information. Task information, including the task name, is obtained by calling the functions and interfaces of the Volcano controller components.
(2) Determine whether it is running on a parameter server pod, that is, if it is marked as a "ps" type. If so, it is preferentially placed in the task queue. Otherwise, it enters into the priority judgment procedure.
(3) Place the task in a storage unit bucket. A bucket is created for the first task. If the second task has an affinity relationship with the task in the first bucket, it is placed in the first bucket. If an anti-affinity relationship exists between them, another bucket is created for the task. This process stops when all tasks are traversed. The time complexity of the intra-group priority of a job is assumed to be $O(h)$. Thus, the total time complexity of the intra-group priority of all jobs is $O(n \cdot h)$. The time complexity of the inter-group priority between two jobs is $O(n)$, and thus the time complexity of the inter-group priority for the job queue is $O(n \cdot (n-1))$.

Algorithm: CPS algorithm

Input: *QueueA*

Output: *QueueB*

Initialize *(up，jp，tp)* ;
while (True)do
  read *jobqueue* ;
  Get user info *up* ;
  Get job info *jp* ;
  Get wait time info J ;
  Calculate urgency J ;
  Get parallel info P ;
  Normalized J ;
  Normalized P ;
  Calculate Priority;
  Compare *job*1P, *job*2P ;
while (True)do
read *job* ;
Get task info;
Get task name;
if (*tasktype* ==ps')
  Schedule taskname;
  Compare *task1P,task2P* ;
endif
break;
  endwhile
break;
endwhile
return *QueueB*

4 Results and discussion

Experiments were conducted on the inter- and intra-group priorities and the group scheduling under different scenarios.

4.1 Preparation

A Kubernetes cloud environment is constructed using a virtual machine (VM) cluster for the experiment. The VM cluster has 10 VM nodes, including 1 master node and 9 computing nodes, as shown in Fig. 4. These nodes install Kubernetes components such as kubeadm, kubelet, kubectl, and docker. In the experiment, different numbers of computing nodes were used to validate the different scenarios. The node configurations are listed in Table 1.

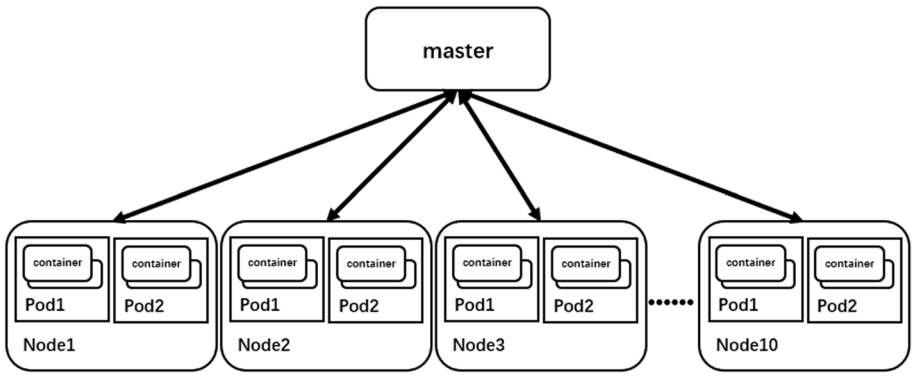

Fig. 4 Kubernetes cloud environment for the experiment. The Kubernetes cloud environment required by the experiment is built by virtual machine (VM) clusters. VM cluster has 10 VM nodes, including 1 master node and 9 computing nodes

Table 1 Experimental cluster node settings

Nodes	Master number	Node number	Master configuration	Node configuration
3 nodes	1	2	4CPU8GB	8CPU16GB
6 nodes	1	5	4CPU8GB	8CPU16GB
10 nodes	1	9	4CPU8GB	4CPU8GB

Table 2 Impact factor settings of multiple jobs

	Job priority	Maximum wait time	Task parallelism	User priority
S1	Difference	Same	Same	Same
S2	Same	Difference	Same	Same
S3	Same	Same	Difference	Same
S4	Same	Same	Same	Difference

The public MNIST dataset was used in the experiment. The TensorFlow and MindSpore frameworks were applied to train the model using the MNIST dataset. The experiment verified the priority scheduling between multiple jobs and the priority scheduling of tasks within a job. To analyze the effectiveness and applicability of the CPS algorithm under different impact factors, various experimental scenarios were established, as listed in Tables 2, 3 and 4. Table 2 presents the experimental scenarios with different impact factors for multiple jobs. In Scenario S1, multiple jobs have different job priorities but the same user priority, maximum wait time, and task parallelism. Scenario S2 indicates that multiple jobs have different maximum wait times but the same job priority, task parallelism, and user priorities. Scenarios S3 and S4 were similar to scenarios S1 and S2, respectively.

Table 3 presents the experimental setup for intra-group priority scheduling, including seven groups. Groups 1–4 use the same number of virtual machine nodes, training framework, and workers to verify the priority effectiveness of the CPS algorithm under different scenarios. Group 5 verifies whether the priority effectiveness of the

Table 3 Experiment setting for intra-group priority scheduling

	Training framework	Number of worker nodes	Number of VM Nodes	Scenarios
Group 1	TensorFlow	5	3	S1
Group 2	TensorFlow	5	3	S2
Group 3	TensorFlow	5	3	S3
Group 4	TensorFlow	5	3	S4
Group 5	MindSpore	5	3	S3
Group 6	TensorFlow	5	6	S3
Group 7	TensorFlow	5	10	S3

Table 4 Experiment setting for inter-group priority scheduling

	Framework	Number of pods	Number of VM nodes	Pod configuration
Group 1	TensorFlow	2ps6worker	6	2CPU2GB
Group 2	TensorFlow	2ps6worker	10	2CPU2GB
Group 3	MindSpore	2ps6worker	10	2CPU2GB
Group 4	TensorFlow	2ps10worker	10	2CPU2GB
Group 5	TensorFlow	2ps6worker	10	1CPU1GB
Group 6	TensorFlow	2ps6worker	10	4CPU4GB

CPS algorithm is affected by the training framework compared to group 3. Groups 6 and 7 verify whether the priority effectiveness is affected by the number of VM nodes.

Table 4 presents the experimental setup used for inter-group priority scheduling, including six of the groups. Each group reflects the state of the job execution. For example, group 1 uses the TensorFlow training framework, two parameter server pods, and six worker pods to execute multiple tasks. The Kubernetes cloud environment operates on six nodes: one master node and five computing nodes. Experiments 1 and 2 were conducted to verify whether the priority effectiveness of the CPS algorithm is affected by the number of VM nodes on the Kubernetes cloud platform. The group 3 experiment verified whether the priority effectiveness of the CPS algorithm is affected by the training framework compared with the group 2 experiment. Similarly, the group 4 experiment was conducted to verify whether the priority effectiveness of the CPS algorithm is affected by different numbers of worker pods compared to the group 2 experiment. The built-in priority and CPS algorithm plugins in the Volcano scheduler were separately used to conduct resource scheduling experiments. Groups 2, 5, and 6 were used to verify whether the priority effectiveness of the CPS algorithm is affected by the pod configuration.

4.2 Results and comparison of intra-group job scheduling

The job parameters of distributed machine learning are listed in Table 5. By combining these parameters into different scenarios, the experiment results validate the idea that the final job priority is influenced by the four impact factors.

Because a job runs on a pod group in the Volcano scheduler, the priority among jobs is mapped to the scheduling order of the pod groups. The initial job priority is set to 1 for a low level, 50 for a normal level, and 100 for a high level. Under sufficient

Table 5 Job parameters of distributed machine learning

Sequence	Parameter	Job1	Job2	Job3	Job4	Job5
1	User priority	5	5	5	5	5
2	User priority	9	7	5	3	1
3	Job priority	Normal	Normal	Normal	Normal	Normal
4	Job priority	Normal	Normal	High	High	Low
5	Task parallelism	6	6	6	6	6
6	Task parallelism	8	6	4	3	2
7	Maximum wait time	20	20	20	20	20
8	Maximum wait time (min)	20	30	40	50	60

Table 6 Scheduling order of pod groups in group 1 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg4	podg3	podg2	podg1	podg5
	podg4	podg3	podg2	podg1	podg5
	podg3	podg4	podg2	podg1	podg5
	podg3	podg4	podg1	podg2	podg5
	podg4	podg3	podg2	podg1	podg5
Combined priority	podg4	podg3	podg2	podg1	podg5
	podg3	podg4	podg1	podg2	podg5
	podg4	podg3	podg2	podg1	podg5
	podg4	podg3	podg2	podg1	podg5
	podg3	podg4	podg2	podg1	podg5

resources, the built-in priority scheduling and CPS algorithm plugins of the Volcano scheduler are implemented for five jobs. The scheduling order of the pod groups created on the resource nodes is obtained using the system log and timestamp.

The group 1 experiment conducted five distributed machine learning jobs using the same MNIST dataset under the TensorFlow framework and a Kubernetes cluster with three nodes. These five jobs were configured as the combined conditions of rows 1, 4, 5, and 7 in Table 5, that is, the corresponding scenario S1 in Table 2 with different job priorities and other identical parameters. Table 6 lists the scheduling order of the pod groups for these five jobs using the built-in priority and combined priority algorithms, for which each algorithm is implemented five times.

It can be seen from Table 6 that pod groups 3 (podg3) and 4 (podg4) were preferentially created owing to their higher job priority compared to other jobs. The pod group podg5 is created last because it has the lowest job priority. The results show that both the built-in and CPS algorithms can be prioritized according to their job priorities.

The results of group 2 were similar to those of group 1. Five jobs were set as the combined conditions for rows 1, 3, 3, 5, and 8 in Table 5. The maximum wait time for the jobs differed, and the other parameters were the same, corresponding to scenario S2 in Table 2. To ensure sufficient resources, the built-in priority and combined priority algorithms were tested five times. Table 7 lists the scheduling orders for the pod groups.

Table 7 Scheduling order of pod groups in group 2 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg4	podg3	podg1	podg2	podg5
	podg4	podg2	podg1	podg3	podg5
	podg3	podg5	podg1	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg5	podg3	podg1	podg4	podg2
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

Table 8 Scheduling order of pod groups in group 3 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg5	podg3	podg1	podg2	podg4
	podg4	podg2	podg1	podg3	podg5
	podg3	podg1	podg5	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg2	podg3	podg4	podg1	podg5
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

The maximum wait time increases from pod groups podg1 to podg5. The built-in priority algorithm of the Volcano scheduler cannot allocate resources in advance for jobs with a long wait time, whereas the CPS algorithm can preferentially allocate resources to jobs with the shortest wait time. Pod group podg1 preferentially allocates resources because it has the shortest wait time, followed by pod groups podg2, podg3, podg4, and podg5, the latter of which has the longest wait time.

The five jobs in group 3 were configured using the combined conditions of rows 1, 3, 6, and 7 in Table 5. These jobs have a different task parallelism and identical parameters, which correspond to scenario S3 in Table 2. The task parallelism of the five jobs decreased from job1 to job5. Thus, the pod group for job1 will be allocated resources first, followed by pod groups podg2, podg3, podg4, and podg5 in the CPS algorithm. The experiment results show that the CPS algorithm starts the pod groups from large to small according to the task parallelism, whereas the built-in priority scheduling order is random and unaffected by the task parallelism, as shown in Table 8.

The five jobs in the group 4 experiment were set as the combined conditions of rows 2, 3, 6, and 7 in Table 5 with different user priorities and other identical parameters, corresponding to scenario S4 in Table 2. The scheduling order of the pod groups under an experimental procedure similar to those of groups 1 and 2 is shown in

Table 9 Scheduling order of pod groups in group 4 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg5	podg1	podg3	podg2	podg4
	podg4	podg2	podg1	podg3	podg5
	podg5	podg1	podg3	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg1	podg2	podg4	podg3	podg5
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

Table 10 Scheduling order of pod groups in group 5 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg4	podg3	podg1	podg2	podg5
	podg4	podg2	podg1	podg3	podg5
	podg3	podg5	podg1	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg5	podg3	podg1	podg4	podg2
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

Table 9. The scheduling order of the built-in priority algorithm is random and unaffected by the user priority, whereas the scheduling order of the CPS algorithm is arranged from large to small according to the user priority.

For group 5, the MNIST dataset was used to conduct five jobs of distributed machine learning under the MindSpore framework, which were configured using the combined conditions in rows 1, 3, 5, and 8 in Table 5. That is, the maximum wait times of the five jobs were different, and the other parameters were identical. It can be seen from the experiment results in Table 10 that the scheduling order of the pod groups for the five jobs is random, whereas pod group podg1 with the shortest wait time is preferentially scheduled, followed by pod groups podg2 and podg5, with the longest wait time created last. These results are similar to the scheduling of the TensorFlow framework, which indicates that the scheduling priority is unaffected by the distributed machine learning framework.

To conduct the experiments in the TensorFlow framework, groups 6 and 7 used 6 and 10 VM nodes, respectively. Except for the training framework, the five jobs were configured under the same conditions as in group 5. The experiment results are presented in Tables 11 and 12. The experiment results are similar to those for the cluster

Table 11 Scheduling order of pod groups in group 6 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg4	podg3	podg1	podg2	podg5
	podg5	podg2	podg1	podg3	podg4
	podg3	podg5	podg1	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg5	podg3	podg1	podg4	podg2
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

Table 12 Scheduling order of pod groups in group 7 experiment

Algorithm	Order1	Order2	Order3	Order4	Order5
Built-in priority	podg3	podg4	podg1	podg2	podg5
	podg4	podg2	podg1	podg3	podg5
	podg3	podg5	podg1	podg4	podg2
	podg1	podg5	podg2	podg4	podg3
	podg5	podg3	podg1	podg4	podg2
Combined priority	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5
	podg1	podg2	podg3	podg4	podg5

with three VM nodes, indicating that the built-in priority and CPS algorithms are unaffected by the cluster size.

In conclusion, the CPS algorithm can effectively solve the priority scheduling problem among multiple jobs by preferentially allocating resources to jobs with shorter wait times, large task parallelism, higher user priority, and higher job priority. The training framework of distributed machine learning and the cluster size cannot affect the effectiveness of a priority scheduling algorithm.

4.3 Results and comparison of inter-group job scheduling

Six groups of experiments were conducted using the MNIST dataset, and each group was executed 20 times. The experiment results are listed in Table 13.

The inter-group priority scheduling success rate of the CPS algorithm exceeded 80%. The two task priorities were compared before the pods were created. The task information was obtained before comparing the priorities. If a task belongs to the "ps" type, this parameter sever pod will be unconditionally scheduled. Otherwise, it enters a normal-priority comparison program. Owing to the randomness of the task priority comparison, the parameter sever pod cannot be scheduled first every time. The CPS algorithm first increases the scheduling probability of the parameter server pods

Table 13 Experiment results of inter-group job scheduling

Experimental group	Inter-group priority scheduling success rate of CPS algorithm (%)	Average job implementation time of built-in algorithm (s)	Average job implementation time of CPS algorithm (s)
Group 1	80	257.4	241.2
Group 2	90	250.5	233.9
Group 3	90	189.0	182.8
Group 4	85	221.2	209.2
Group 5	95	472.1	445.7
Group 6	85	37.8	36.3

by determining the pod type created for a task. Second, this algorithm places worker nodes in the corresponding storage unit according to their affinity with the parameter server nodes when it is placed in a storage unit, which can prevent many worker nodes from being placed into a storage unit owing to a lack of affinity or anti-affinity relations between worker nodes. This ensures that the placement of pods is more reasonable in the cluster server and further improves the efficiency of job completion by optimizing the distance of the information interaction between the parameter server and worker nodes. The average job implementation time of the CPS algorithm is less than that of the built-in algorithm. The job implementation time of the CPS algorithm is more than 6% shorter than that of the built-in algorithm. It is unaffected by the cluster size or pod configuration. The job completion time of the group 5 experiment with 2ps6worker pods and all pods with a 1CPU1GB memory configuration is nearly double that of groups 1 and 2. The job completion time of the group 6 experiment with 2ps6worker pods and all pods with the 4CPU4GB memory configuration was greatly reduced to only 36.3 s. To further analyze the effectiveness of parameter server priority scheduling and job execution in detail, the experiment results of the randomly selected group 2 experiment are shown in Fig. 5. The scheduling order of the parameter server pods in the CPS algorithm clearly precedes the built-in priority algorithm in the Volcano scheduler. In 20 of the experiments, the CPS algorithm achieved the later scheduling order of the parameter server pods only three times and the same order as the built-in priority algorithm five times. In the ten experiments that recorded the job completion time shown in Fig. 6, the job completion time of the CPS algorithm was lower than that of the built-in priority algorithm 9 times. The remaining time gap was extremely small, i.e., 224 s for the CPS algorithm and 222 s for the built-in priority algorithm. The job completion time is reduced by setting the affinity or anti-affinity relations between the parameter server and worker pods in the CPS algorithm.

Group 5 conducted distributed machine learning on 2ps6worker pods under the TensorFlow framework, where each pod was allocated one CPU core and 1 GB of memory, and there were 10 VM nodes in the Kubernetes cloud environment. Group 6 uses a pod with four CPU cores and 4 GB of memory. The experiment results are presented in Figs. 7 and 8.

The left sides of Figs. 7 and 8 show that the scheduling order of most of the parameter sever pods of the CPS algorithm was lower than that of the built-in priority

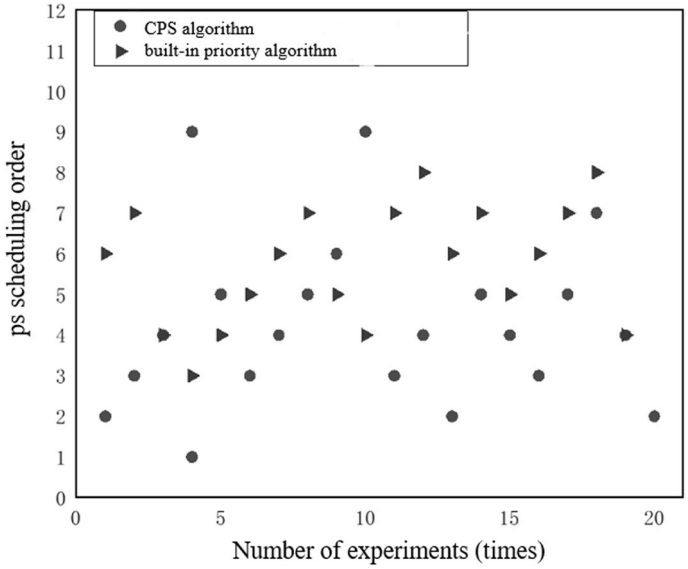

Fig. 5 Scheduling order of parameter server pods. In order to further analyze the effectiveness of priority scheduling and job execution of the parameter server, the experimental results of the second group of randomly selected experiments show that the scheduling order of the parameter server pod in the CPS algorithm is obviously better than the built-in priority algorithm in the Volcano scheduler

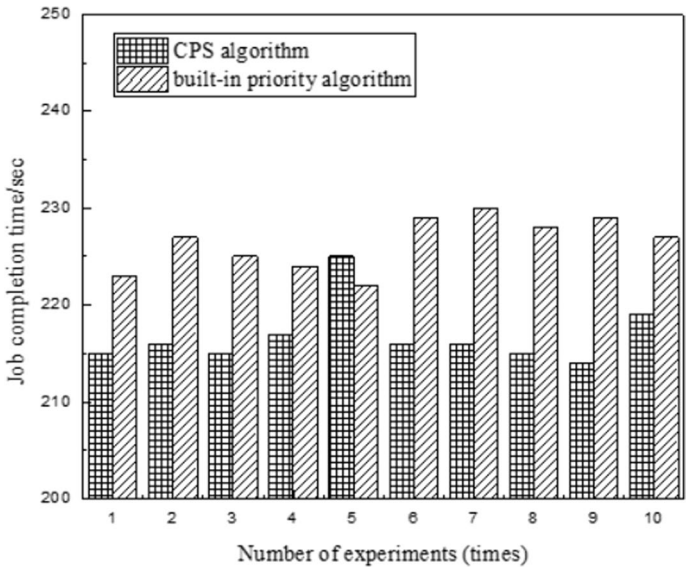

Fig. 6 Job completion time. In ten experiments of recording job completion time, the job completion time of CPS algorithm is 9 times shorter than that of the built-in priority algorithm

algorithm. The job completion time of the CPS algorithm was shorter than that of the built-in priority algorithm in most of the experiments. These two situations indicate that the CPS algorithm is more effective than the built-in priority algorithm for priority scheduling. This phenomenon is more evident when a task requires a smaller pod resource. For example, when the tasks request one CPU core and a 1-GB memory pod resource, the job completion time increases when applying the CPS or built-in

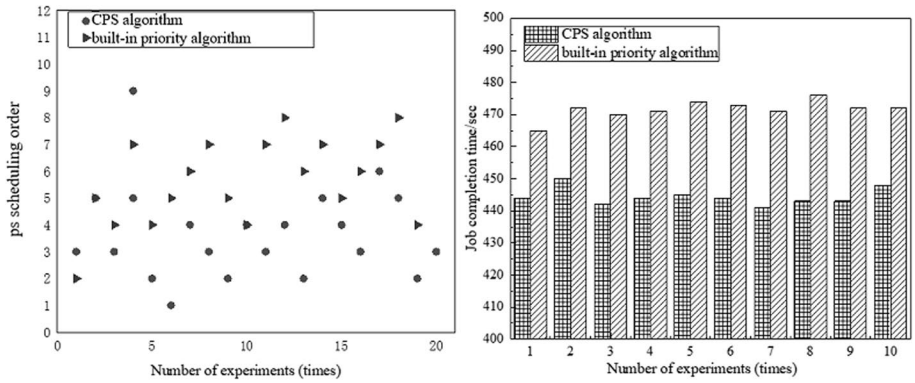

Fig. 7 Experiment result of group 5. The group 5 experiment performs a job of distributed machine learning on 2ps6worker pods under TensorFlow framework, where each pod is allocated 1 CPU core and 1 GB memory and the number of VM nodes in Kubernetes cloud environment is 10. It can be seen that the scheduling order of most parameters of the CPS algorithm server pod is lower than the built-in priority algorithm

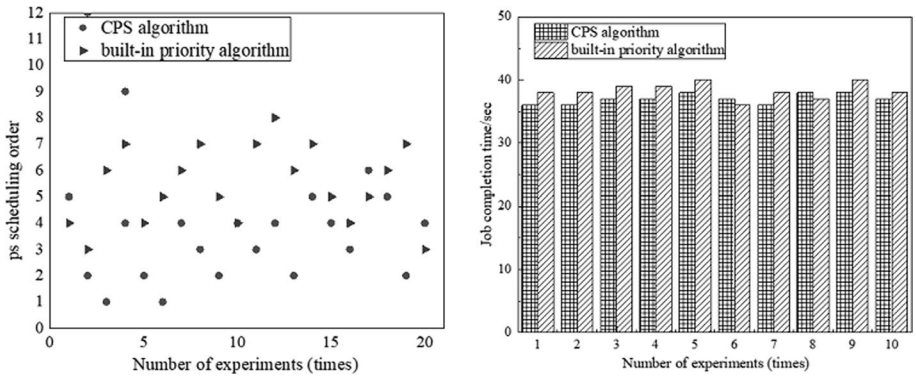

Fig. 8 Experiment result of group 6. The group 6 experiment uses the pod with 4 CPU cores and 4 GB memory. It can be seen that the scheduling order of most parameters of the CPS algorithm server pod is lower than the built-in priority algorithm

priority algorithm, and their difference is large, with a maximum value of 32 s, as shown in Fig. 7. When the tasks requested four CPU cores and a 4-GB memory pod resource, the job completion time reduced the performance of the CPS or built-in priority algorithm, and their difference was small, with an average value of 1.5 s, as shown in Fig. 8.

4.4 Results and comparison of group scheduling

To verify the effectiveness of the CPS algorithm in avoiding resource deadlock and waste, an experiment was conducted on five distributed machine learning jobs using the parameters listed in Table 14. Here, job5 was configured with 2ps6worker pods, and job1, job2, job3, and job4 were configured with 2ps4worker pods. Each pod requested two CPU cores and 2 GB of memory. The number of VMs for the Kubernetes cloud environment was set to three. The experiment results are shown in Figs. 9 and 10.

Table 14 Job parameters of distributed machine learning

Parameter	Job1	Job2	Job3	Job4	Job5
User priority	1	3	5	7	9
Job priority	High	High	Normal	Normal	Low
Task parallelism	4	4	4	4	6
Maximum wait time (min)	60	50	40	30	20

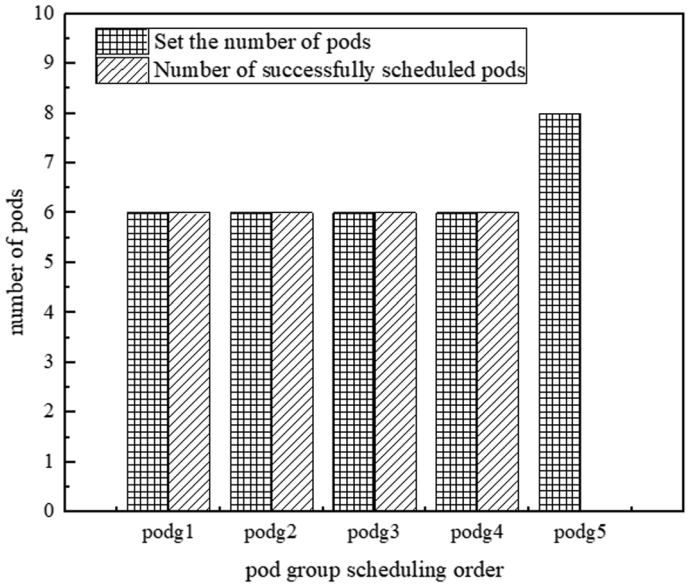

Fig. 9 Scheduling order of pod groups of the built-in priority algorithm. The experimental results of scheduling order of pod groups of the built-in priority algorithm under the experimental conditions that the 2ps6worker pod is configured in job 5, and the 2ps4worker pod is configured in job 1, job 2, job 3, and job 4

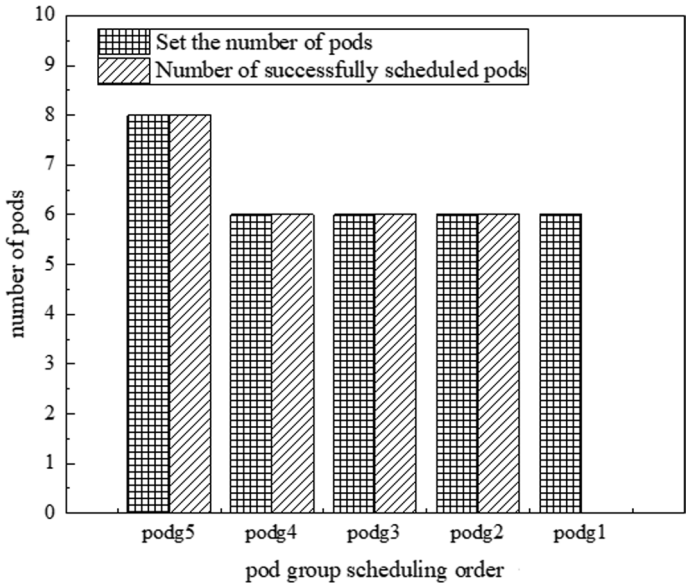

Fig. 10 Scheduling order of pod groups of the CPS algorithm. The experimental results of scheduling order of pod groups of the CPS algorithm under the experimental conditions that the 2ps6worker pod is configured in job 5, and the 2ps4worker pod is configured in job 1, job 2, job 3, and job 4

The scheduling of pod groups under the built-in priority algorithm in the Volcano scheduler was podg1, podg2, podg3, podg4, and podg5. Pod group podg5 with a higher user priority cannot be created because of insufficient resources, which can result in a longer job completion time and resource usage. However, job5 had a higher combined priority owing to its high user priority, high task parallelism, and shortest wait time. Therefore, podg5 preferentially allocates resources to conduct job5. Pod group podg1 does not allocate resources to conduct job1 owing to its low combined priority. This experiment verifies that the CPS algorithm works together with the group scheduling strategy for distributed machine learning jobs.

5 Conclusions

Model training used in distributed machine learning is generally computationally intensive and time-consuming, which can waste resources and generate deadlock problems owing to an inefficient priority scheduling. A combined priority scheduling method for preferentially scheduling jobs or tasks with high priorities was proposed herein. This method comprehensively considers the impact factors of user priority, job priority, maximum wait time, task parallelism, and affinity and anti-affinity for constructing a combined priority that includes intra- and inter-group scheduling priorities. These priorities are mapped to the priorities of the pod groups and pods. A plugin for the combined priority scheduling method was developed and integrated into the Volcano scheduling framework. The experiment results show that the CPS method not only realizes the affinity, anti-affinity, and group scheduling mechanisms for alleviating resource waste and deadlocks to a certain extent, but it also preferentially allocates resources to jobs with high parallelism, user priority, and urgency, thereby improving the job execution efficiency. This method is suitable for deep learning using the TensorFlow framework. At present, many jobs involving machine learning workflows are more complex, with different resource requirements, more priorities, and more complex correlations between tasks. It is therefore necessary to further study and establish a more complex combinatorial priority model for realizing an efficient resource scheduling of workflow tasks.

Author contributions

JC is a major contributor in proposing the method. TD designed this method and draft this manuscript. GX developed the script of this method. CZ and HS carried out the partial experimental work. WI and YG contributed to the partial data analysis. All author(s) read and approved the final manuscript.

Funding

This work was supported by Project of Key R&D Program of Shandong Province (2022CXGC20106), Shandong Provincial Natural Science Foundation (ZR2020MF034), Qilu University of Technology (Shandong Academy of Sciences) pilot major innovation project of integrating science, education and industry (2022JBZ01-01), China.

Availability of data and materials

The datasets used during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 30 November 2022 Accepted: 17 May 2023
Published online: 29 May 2023

References

1. A. Mahmoodzadeh, H.R. Nejati, M. Mohammadi et al., Prediction of mode-I rock fracture toughness using support vector regression with metaheuristic optimization algorithms. Eng. Fract. Mech. **264**, 108334 (2022)
2. F. Chen, C.Y. Yang, K. Mohammad, Diagnose Parkinson's disease and cleft lip and palate using deep convolutional neural networks evolved by IP-based chimp optimization algorithm. Biomed. Signal Process. Control **77**, 103688 (2022)
3. Y. Guo, M. Khishe, M. Mohammadi et al., Evolving deep convolutional neural networks by extreme learning machine and fuzzy slime mould optimizer for real-time sonar image recognition. Int. J. Fuzzy Syst **24**, 1371–1389 (2022)
4. M. You, W. Luo, M. He, Resource scheduling of information platform for general grid computing framework. Int. J. Web Grid Serv. **16**(3), 254–272 (2020)
5. Q. Shi, F. Li, M. Olama et al., Network reconfiguration and distributed energy resource scheduling for improved distribution system resilience. Int. J. Electr. Power Energy Syst. **124**, 106355 (2021)
6. R. Gatti, S. Shankar, Bidirectional resource scheduling algorithm for advanced long term evolution system. Eng. Rep. **2**(7), e12192 (2020)
7. N. Malarvizhi, S.G. Priyatharsini, S. Koteeswaran, Cloud resource scheduling optimal hypervisor (CRSOH) for dynamic cloud computing environment. Wirel. Pers. Commun. **115**(1), 27–42 (2020)
8. M. Wang, J. Guo, W. Wang et al., Smart grid network resource scheduling algorithm based on network calculus. Integr. Ferroelectr. **199**(1), 1–11 (2019)
9. D. Jiang, H. Lin, Review on key technologies of resource allocation in cloud computing environment. J. China Acad. Electron. Sci. **13**(3), 308–314 (2018)
10. Y. Gong, B. Li, B. Liang. Chic: experience-driven scheduling in machine learning clusters. in *Proceedings of the International Symposium on Quality of Service*, (Phoenix, 2019), pp. 1–10
11. J. Zhou, Q. Cui, X. Li, System. PSMART: parameter server based multiple additive regression trees system. in *Proceedings of the 26th International Conference on World Wide Web Companion*, (2017), pp. 879–880.
12. Y.S.L. Lee, M. Weimer, Y. Yang, et al. Dolphin: Runtime optimization for distributed machine learning. *International Conference on Machine Learning ML Systems Workshop*, (New York City, 2016), pp. 1–14
13. K.B. Dewangan, A. Agarwal, M. Venkatadri et al., Self-characteristics based energy-efficient resource scheduling for cloud. Proc. Comput. Sci. **152**, 204–211 (2019)
14. H. Zhao, D. Shen, L. Tian, Research on resource demand forecasting and scheduling method in cloud computing environment. Small Micro Comput. Syst. **37**(4), 659–663 (2016)
15. J. Jang, J. Jung, J. Hong, An efficient virtual CPU scheduling in cloud computing. Soft. Comput. **24**(8), 5987–5997 (2019)
16. J.G. Mirobi, L. Arockiam, DAVmS: distance aware virtual machine scheduling approach for reducing the response time in cloud computing. J. Supercomput. **6**, 1–12 (2021)
17. L.K. Devi, S. Valli, Multi-objective heuristics algorithm for dynamic resource scheduling in the cloud computing environment. J. Supercomput. **77**(8), 8252–8280 (2021)
18. C. Li, Y. Zhang, Y. Luo, Neighborhood search-based job scheduling for IoT big data real-time processing in distributed edge-cloud computing environment. J. Supercomput. **77**(2), 1853–1878 (2021)
19. S. He. *Research on Constructing Elastic Cluster Based on Docker resource prescheduling strategy*. (Zhejiang Sci-tech University, 2017)
20. L. Chen, J. Wang, Research on load balancing algorithm based on rough set. Comput. Eng. Sci. **31**(1), 101–104 (2010)
21. A. Asadpour, A. Saberi, An approximation algorithm for max–min fair allocation of indivisible goods. SIAM J. Comput. **39**(7), 2970–2989 (2010)
22. H.A. Ben, S.A. Ben, A. Ezzati et al., A novel multiclass priority algorithm for task scheduling in cloud computing. J. Supercomput. **77**(10), 11514–11555 (2021)
23. C. Li, H. Zhang, W. Ding et al., Fair and near-optimal coflow scheduling without prior knowledge of coflow size. J. Supercomput. **77**(7), 7690–7717 (2021)
24. G. Muthsamy, S.C. Ravi, Task scheduling using artificial bee foraging optimization for load balancing in cloud data centers. Comput. Appl. Eng. Educ. **28**(4), 769–778 (2020)
25. J. Liu, T. Yang, J. Bai et al., Resource allocation and scheduling in the intelligent edge computing context. Futur. Gener. Comput. Syst. **121**, 48–53 (2021)
26. D.N. Nguyen, T. Kim, Balanced leader distribution algorithm in Kubernetes clusters. Sensors **21**(3), 869 (2021)
27. N. Nguyen, T. Kim, Toward highly scalable load balancing in Kubernetes clusters. IEEE Commun. Mag. **58**(7), 78–83 (2020)
28. T. Menouer, KCSS: Kubernetes container scheduling strategy. J. Supercomput. **77**(5), 4267–4293 (2021)
29. G. Zheng, Y. Fu, T. Wu, Research on docker cluster scheduling based on self-define Kubernetes scheduler. J. Phys. Conf. Ser. **1848**(1), 012008 (2021)
30. H. Tai. *Design and Implementation of Two-Level Resource Scheduler Based on Kubernetes-on-EGO*. (Xidian University, 2017)
31. A. Shu, X. Peng, W. Zhao, Cloud computing resource adaptive management method based on container technology. Comput. Sci. **44**(7), 120–127 (2017)
32. Y. Fu, S. Zhang, J. Terrero et al., Progress-based container scheduling for short-lived applications in a Kubernetes cluster. IEEE Int. Conf. Big Data (Big Data) **2019**, 278–287 (2019)
33. J. Marcelin. *Research on Resource Scheduling Strategy based on Kubernetes Container Cluster*. (Xi 'An University of Science and Technology, 2019)
34. Q. Liu. *Resource Scheduling Method and Device Based on Kubernetes System*. (Guangdong Province: CN110515704A, 2019-11-29)
35. K. Xu. *Design and Implementation of Scalable Distributed Resource Scheduler Based on Kubernetes*. (Xidian University, 2017)
36. A. Havet. *GENPACK: A Generational Scheduler for Cloud Data Centers. IEEE International Conference on Cloud Engineering*. (2017), pp. 95–104

37. L. Toka, Ultra-reliable and low-latency computing in the edge with Kubernetes. J. Grid Comput. **19**(3), 1–23 (2021)
38. K. Kaur, S. Garg, G. Kaddoum et al., KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem. IEEE Internet Things J. **7**, 4228–4237 (2019)
39. X. Sun, Research on resource scheduling algorithm based on business priority in 5G vehicle network scenario. Chongqing Univ. Posts Telecommun. **23**, 668 (2019)
40. M. Umi, D. Jakobovi, Ensembles of priority rules for resource constrained project scheduling problem. Appl. Soft Comput. **110**(1), 107606 (2021)
41. Y. Sun. *Research on Preemptive Scheduling Strategy of Multi-Dag Workflow in Cloud Computing*. (Xinjiang University)

Publisher's Note