information

MDPI

Article

Research and Implementation of Scheduling Strategy in Kubernetes for Computer Science Laboratory in Universities

Zhe Wang [1], Hao Liu [1], Laipeng Han [1], Lan Huang [2] and Kangping Wang [1,*]

[1] School of Computer Science and Technology, Jilin University, Changchun 130021, China; wz2000@jlu.edu.cn (Z.W.); liuhao18@mails.jlu.edu.cn (H.L.); hanlp15@mails.jlu.edu.cn (L.H.)
[2] Key Laboratory of Symbolic Computation and Knowledge Engineering, Jilin University, Changchun 130021, China; huanglan@jlu.edu.cn
* Correspondence: wangkp@jlu.edu.cn

Abstract: How to design efficient scheduling strategy for different environments is a hot topic in cloud computing. In the private cloud of computer science labs in universities, there are several kinds of tasks with different resource requirements, constraints, and lifecycles such as IT infrastructure tasks, course design tasks submitted by undergraduate students, deep learning tasks and and so forth. Taking the actual needs of our laboratory as an instance, these tasks are analyzed, and scheduled respectively by different scheduling strategies. The Batch Scheduler is designed to process tasks in rush time to improve system throughput. Dynamic scheduling algorithm is proposed to tackle long-term lifecycle tasks such as deep learning tasks which are hungry for GPU resources and have dynamically changing priorities. Experiments show that the scheduling strategies proposed in this paper improve resource utilization and efficiency.

Keywords: cloud computing; Kubernetes; scheduling strategy; resource management

check for updates

Citation: Wang, Z.; Liu, H.; Han, L.; Huang, L.; Wang, K. Research and Implementation of Scheduling Strategy in Kubernetes for Computer Science Laboratory in Universities. *Information* **2021**, *12*, 16. https://doi.org/10.3390/info12010016

Received: 20 November 2020
Accepted: 28 December 2020
Published: 3 January 2021

1. Introduction

In complicated computing context, cloud computing can significantly improve resource utilization rate. In cloud architecture, the exploitation of container technology [1], such as Docker, is becoming more and more widespread [2]. Accordingly, there has been a dramatic increase in the popularity of Container as a Service (CaaS) clouds. CaaS is a model that uses containers as the unit of resource division. Cloud providers will virtualize compute resources in the form of containers and deliver them to users as services [3,4]. Based on CaaS and containerization technique, a complex monolithic application could be divided into several microservices and these containerized services are distributed on multiple physical hosts [5,6]. Therefore, there may be many container-based heterogeneous tasks in a cloud platform. How to schedule these tasks is a key problem [7]. There is no 'silver bullet' for schedulers. Schedulers for different clouds in different circumstances perform diversely and it is meaningful to explore scheduling strategies for various requirements.

In our private cloud, these containerized tasks are deployed and managed by Kubernetes which is a lightweight and powerful orchestration tool [8]. Kubernetes provides lots of fundamental functions, such as scheduling, monitoring, resource allocation and and so forth. Besides, Kubernetes is an open source project and provides customized interface for plugins, so we can replace the scheduling strategy to meet our platform's requirements [9].

In the scenario of our computer science laboratory, the computing resources are relatively limited, especially about GPU, and the time for resource purchase is quite long. Moreover, there are greatly diverse requirements in tasks. For instance, there are some basic tasks with long life cycles that require high availability, and short tasks in traffic jam submitted by undergraduates for their class design which have relatively low reliability and usability, and some resource-consuming and time-consuming tasks such as image processing and deep learning which are vying for GPUs.

During deployment, we found that the default strategy in Kubernetes treats all tasks equally and all Pods are scheduled by the same scheduling process with default score methods. To achieve better deployment and resource utilization rate, different scheduling processes and constraint rules for different tasks should be employed. Moreover, in default scheduler of Kubernetes, tasks are scheduled from the priority queue in order statically. But the lifecycles of tasks are distributed in a wide time span. The tasks should be rescheduled to tackle the dynamical changes of the system. The default scheduling strategy can be improved in many ways to suit the needs of laboratory tasks much better.

In this paper, based on some case study analyses in our laboratory, we propose two new scheduling strategies and conduct some experiments. We have built a private cloud in our lab—Jilin Provincial Key Laboratory of Big Data Intelligent Computing. In the cloud, we have deployed basic services such as identity authentication, storage, and knowledge management in containers managed by Kubernetes. The users will submit tasks on web GUI or by REST API. Then, these tasks will be scheduled by our proposed algorithms. In this paper, our main contributions are listed below:

- Two Schedulers, Batch Scheduler and Dynamic Scheduler, are proposed and implemented. These Schedulers are exploited in appropriate situation for different tasks to improve system performance in different aspects.
- The Batch Scheduler contains two steps, greedy method step and local search step, which can search out a better layout in rush hour.
- The Dynamic Scheduler manages tasks by priorities which will dynamically change. These tasks are basically long-term deep learning tasks which the preemption and migration mechanism are mandatory in our system.
- To estimate the Batch Scheduler and the Dynamic Scheduler, some experiments are conducted in our cloud. Compared with the default strategy, the resource utilization rate and efficiency are improved.

The rest of the paper is organized as follows—Section 2 presents some related works. Section 3 introduces the private cloud platform and services of our laboratory. Then, Section 4 will describe the default scheduling strategy in Kubernetes and proposed improvements. In Section 5, we present the implementation details of our strategy. At last, Section 6 describes the results and analysis of the experiment and the paper is concluded in Section 7.

2. Related Work

The purpose of the scheduling is to place the Pod on a particular node. A lot of strategies have been proposed by scholars in recent years.

The default scheduling strategy in Kubernetes keeps updating new features such as Node affinity, Taints, Tolerations, Preemption and Eviction. Node affinity is a property of Pods, including Required and Preferred affinity. This property attracts Pods to a set of nodes. Taints are opposite to Node affinity and they allow a node to repel a set of pods. A Pod will be allowed to accept nodes with matching Taints according to Tolerations. Pods can also have static priority and that allows Pods to preempt or evict lower priority Pods [10].

There are some well-known schedulers in other orchestration tools. A fair algorithm called Dominant Resources Fairness (DRF) is exploited in Mesos [11,12] to solve the problem that each user may have different demands for resources. Binpack [13] applied in Swarm aims to achieve compression of nodes, but that will lead to instability of cluster. These algorithms are easy to implement and perform well in simple cases.

In addition, many complicated algorithms have been proposed to be combined with Kubernetes. For example, in Reference [14], the author proposes a strategy which based on a multi-criteria decision analysis algorithm. In Reference [15], a scheduling algorithm called Balanced-CPU-Disk-IO-Priority (BCDI) is mainly applied to improve the resource balance between IO and disk. In Reference [16], ant colony algorithm and particle swarm optimization algorithm are used to improve the balance of the resource allocation. Reinforcement learning algorithm is applied to the scheduling strategy in References [17,18].

However, the existing algorithms are not suitable for our cases. Lots of them do not concern batch tasks scheduler in which there is a tradeoff between throughput and waiting time. Some keep eyes on I/O and CPU resources which are not key points in deep learning tasks. In deep learning tasks, GPU cards are scarce resources and they are exclusive by a particular task at a time which leads to a different scheduling demand. Algorithms proposed in this paper improve performance in these areas and achieve good results.

3. Our Private Cloud Platform and Services

The architecture of our private cloud is shown in Figure 1. There are four layers which are the resource layer, management layer, application layer and presentation layer from bottom to top respectively [19]. The resource layer includes physical servers, storage, computing, and network resources [20]. In order to manage and monitor the use of resources, Kubernetes is deployed in management layer. The application layer consists of many services of the platform. Moreover, users can manage and monitor their own tasks and services through presentation layer [21,22].

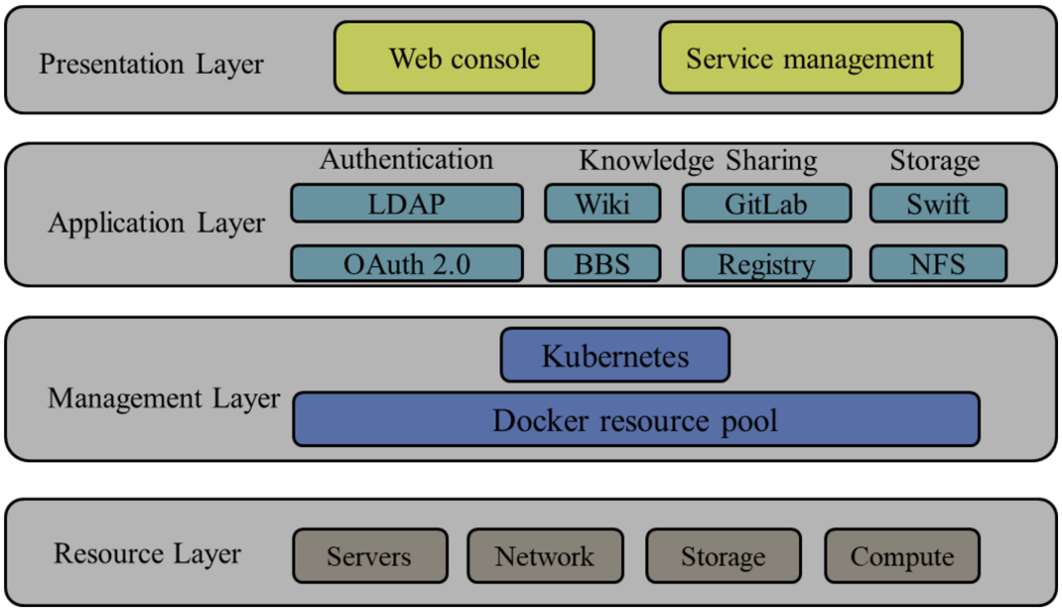

Figure 1. Architecture of the platform.

In the application layer, we have deployed some tasks which are adapted open source software to service our members in research and education, including Wiki, Gitlab, and so forth. And a unified identity authentication system is deployed to authenticate and authorize computing and data resources [23–25]. These tasks should be scheduled at the initial stage and always keep alive.

There are also lots of tasks created by users which include researchers and undergraduate students in our lab course. These tasks such as deep learning training and some short-term coursework have widely distributed lifecycle. For coursework tasks, they will be submitted in batches and their response time can be relatively longer and unstable in busy hour. For deep learning training tasks, some tasks may be starved because GPU resource was occupied by some long running tasks if the Scheduler has not deal with these.

4. The Default Scheduling Strategy and Improvements

There is a default scheduler in Kubernetes to find node for a newly created Pod. The scheduler exploits a priority queue in which head Pod will be scheduled firstly to a selected node. The default scheduling strategy includes three steps: Predicate, Priority and

Select. In the Predicate step, the Scheduler checks all available nodes, and filters out nodes that meet the Pod requirements according to some Predicates, such as CheckNodeConditionPred and PodFitsResources; then the default Scheduler will calculate score of each available node in the second step; LeastRequestedPriority described in Formula (1) is used to pick the host with the most free resources, and BalancedResourceAllocation described in Formula (2) targets to allocate resources in more balanced way. Finally, the node with the highest score is selected to run the Pod [10,26].

$$score = \sum (type((capacity - \sum requested) * 10)/capacity)/3, type \in \{gpu, cpu, mem\} \quad (1)$$

$$score = 10 - sub(fraction_{gpu}, fraction_{cpu}, fraction_{mem}) * 10. \quad (2)$$

The default Scheduler uses priority queues to schedule containers in order. It cannot forecast the resource requested by the subsequent containers, so there is some space to improve the scheduling method. As depicted in Figure 2, there are four Pods A, B, C, and D in the priority queue and two 4-core hosts E and F. The colored boxes indicate the number of CPU cores they require respectively, which are 2, 1, 2, and 3. The Pods in the priority queue are scheduled in order one by one, and we can find that container D cannot be scheduled in this situation. However, there is an available way to schedule all containers if we had a good strategy depicted in desired result.

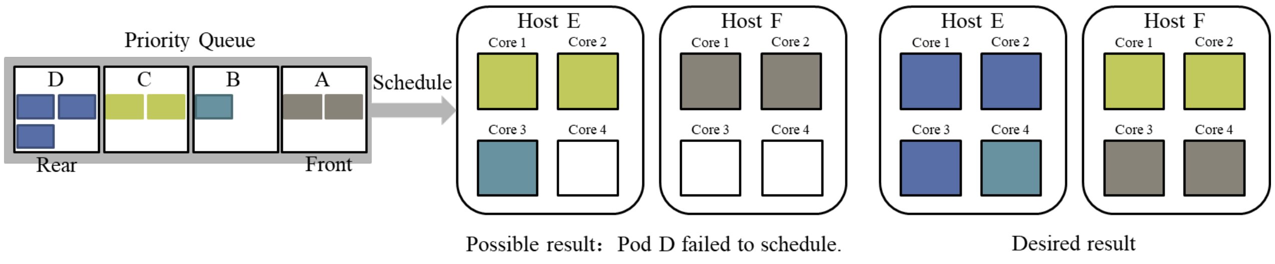

Figure 2. A case of schedule.

In addition, if there were fierce competition for resources, dynamic migration is more effective. There is a case depicted in Figure 3 containing five Pods A, B, C, D, and E and two 4-core hosts G and H. These Pods require 3, 1, 2, 1, 1 CPU cores respectively and they have been ordered in the queue according to the required resources. After scheduling, the resource utilization rate of our current layout is 100%, and there are no Pods waiting to be scheduled. However, when the Pod B and E finished and released the resources, a new Pod F which requires two cores is submitted. In this situation, the Pod F cannot be scheduled. In fact, if Pod D was migrated to host G dynamically, Pod F can be scheduled on host H.

To tackle these problems, a Batch Scheduler and Dynamic Scheduler are proposed and implemented. The Batch Scheduler will schedule tasks all together which are collected in a time span. For these batch tasks, initial deployment is calculated by greedy algorithm after they were sorted firstly. Then, we will adjust the previous deployment to get a better layout.

The Dynamic Scheduler will keep track of all tasks scheduled by itself through all their lifecycle. For these tasks, active tasks are stored in a task buffer where the Dynamic Scheduler will dispatch tasks dynamically. For tasks supporting the dynamic migration, there are some prerequisites. First, an estimated running time which will be contained in scheduling decision must be submitted in Pod. In our private cloud, every user is regarded as an honest user who will submit approximate estimated running time. A regular audit will be taken on the submitted running time by managers in hand to ensure this mechanism is work. Second, these tasks should support Pause and Resume operations. The pausing tasks has 10 min to store their states in our distributed file system. In this way, dynamic strategy is proposed to schedule tasks. The implementation details are presented in next section.

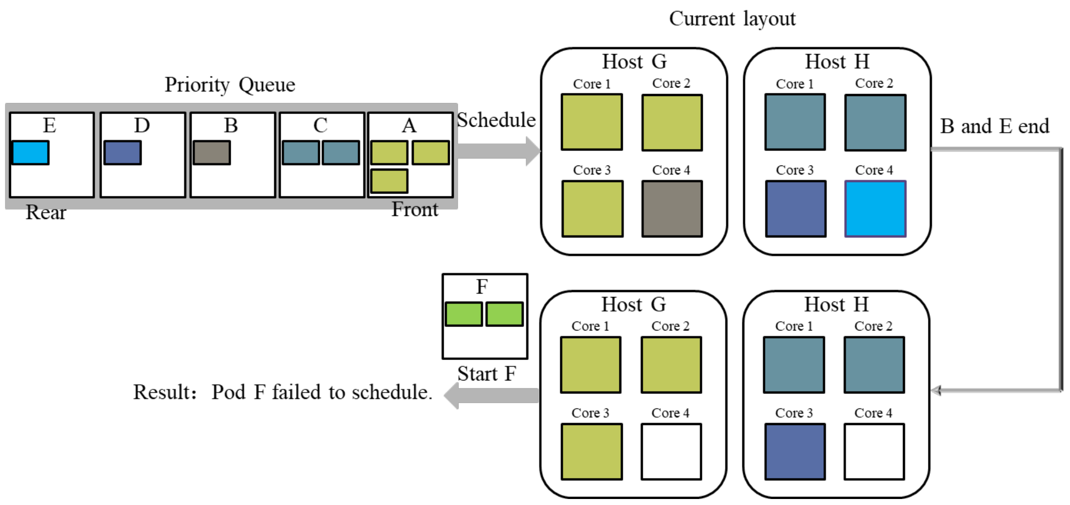

Figure 3. The layout changes, and the platform needs to be dynamically adjusted.

5. The Implementation of Our Strategy

There are three scheduling strategies exploited in different situation. The first one is Default Scheduler provided by Kubernetes, the second is Batch Scheduler used in busy hour and the third one is Dynamic Scheduler for GPU tasks. The last two scheduler are proposed and implemented by our team. First of all, if there were enough resources, the default Scheduler will be employed and it will work very well. So, Batch Scheduler and Dynamic Scheduler will only be brought online by hand or a system monitor when fierce competition for resources was detected.

When Batch Scheduler is employed, the submitted task in a given period will gather in a Candidate Queue firstly. The Pods in this queue will be sorted according to different indicators. Then two optimizing steps called initialSchedule and Swap will be made to get a better deploy info. Based on these data, the batch tasks are scheduled. So, there will be more delay but more throughputs comparing with Default Scheduler.

GPU resources are always scarce in our private cloud, and some tasks which are occupying GPU will last a long time. It is necessary to support a preemptive scheduler in this situation. In our cloud, every GPU task should support pause and resume action. Based on these prerequisites, a Dynamic Scheduler is implemented to manager GPU related tasks. The scheduling process is depicted in Figure 4.

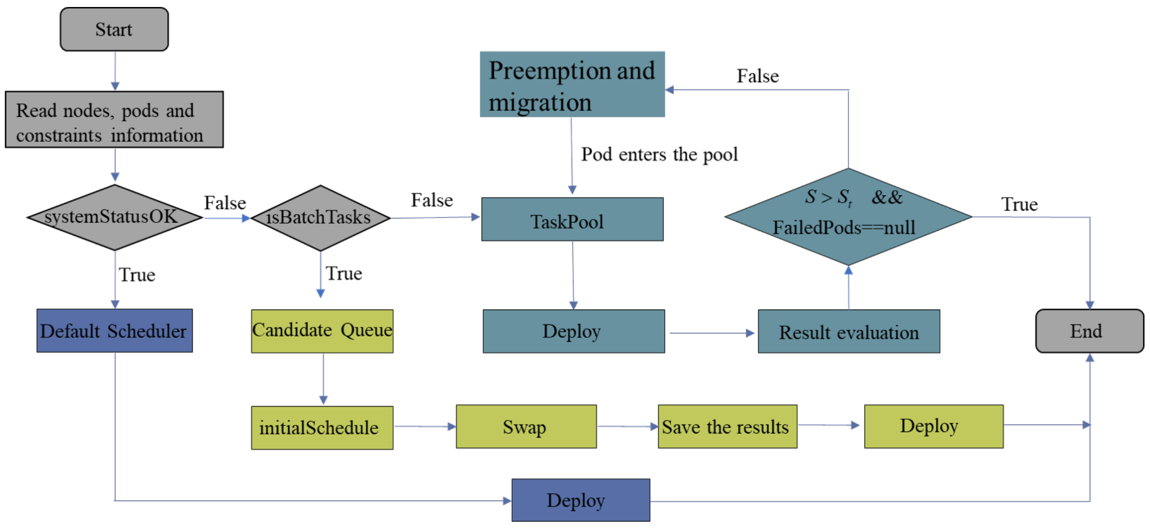

Figure 4. Flow chart of the strategy.

5.1. Batch Scheduler

In general, course design tasks submitted in one minute are treated as a set of batch tasks. These batch tasks are gathered in a Candidate Queue. In order to tackle issues listed in Figures 2 and 3, two improved algorithms are proposed. Firstly, these tasks in queue will be sorted by Algorithm 1. The sorted queue is depicted as $podsSorted$. A predicate function $PREDICATE1$ is used as a parameter of Golang function $SliceStable()$ to sort Pods. In the algorithm, I/O is the most important resource for comparator and the function $PREDICATE1$ implements the comparison (Lines 6–16). Then, the tasks are drained out of the queue one by one to an initial deploy step in which greedy algorithm is exploited. The greedy algorithm is composed of double loops. In inner loop, every node is predicated whether the node can place the Pod selected in outer loop according to the constraints. If a node was selected, the inner loop will be broken. If there were not nodes available, the Pod will be deployed on a special dummy node.

Algorithm 1 Sorting in Candidate Queue Algorithm

```
Input: $pods$
Output: $podsSorted$
 1: function SortCandidateQueue($pods$)
 2:     $Predicate\ Sort\ such\ as\ sort.SliceStable(pods,$ Predicate1$())$
 3:     return $pods$
 4: end function
 5:
 6: function Predicate1($pods[i], pods[j]$)
 7:     if $pods[i].io == pods[j].io$ then
 8:         if $pods[i].cpu == pods[j].cpu$ then
 9:             return $pods[i].mem >= pods[j].mem;$
10:         else
11:             return $pods[i].cpu >= pods[j].cpu;$
12:         end if
13:     else
14:         return $pods[i].io >= pods[j].io;$
15:     end if
16: end function
```

The initial deployment can be optimized in most cases. So, the Swap step is employed which is described in Algorithm 2. It takes $nodeAndPods$ and $rule$ as parameters. The $nodeAndPods$ is deployment relationship from node to Pods in initial deployment. The $rule$ is constraints of the deployment. Firstly, the nodes are sorted from small to large according to their occupied resources and the result is stored in $nodeAndPodsSorted$ (Line 2). $nodePodsMap$ represents the mapping of node and Pods on it. Then, a $ReDeploy$ function where uses $flag$ and $FailedPods$ as signals is employed to decide for each node whether all Pods on it can be migrated to other nodes (Lines 17–33). If $reFlag$ is true, that means all pods on this node can be migrated (Lines 10–11). In order to reduce resource fragmentation, Pods on the node should be sorted according to resources in advance (Line 8). After these steps, we finally get the deployment info for input tasks. If there still were Pods in the dummy node, the scheduling algorithm will just leave these Pods for next scheduling process.

Algorithm 2 Swap Algorithm

```
Input: $nodeAndPods, rule$
Output: $nodeAndPods$
1: function ALGORITHM2($nodeAndPods, rule$)
2:     $nodeAndPodsSorted \leftarrow$ SORTNODESBYRESOURCE($nodeWithPods$)
3:     $nodePodsMap \leftarrow ToNodePods(nodeWithPodsSorted)$
4:     for $node \in nodeList$ do
5:         if $nodePodsMap[node] == null$ then
6:             $Continue$
7:         end if
8:         $pods \leftarrow sortPodsByResource(nodePodsMap[node])$
9:         $reFlag \leftarrow$ REDEPLOY($nodeAndPodsSorted, pods, rule$)
10:        if $reFlag == true$, that is all pods on this node can be migrated then
11:            Migrate pods, update nodeAndPods and nodePodsMap
12:        end if
13:    end for
14:    return $nodeAndPods$
15: end function
16:
17: function REDEPLOY($nodeAndPodsSorted, pods, rule$)
18:    for $pod \in pods$ do $flag \leftarrow true$
19:        for $node \in nodeList$ do
20:            if the pod can be put on the node then
21:                $flag \leftarrow false$
22:                $Break$
23:            end if
24:        end for
25:        if $flag == true$ then
26:            this pod needs to be put in FailedPods
27:        end if
28:    end for
29:    if FailedPods.empty, that is all pods on the node can be migrated then
30:        return $true$
31:    end if
32:    return $false$
33: end function
```

5.2. Dynamic Scheduler

In our cloud, Dynamic Scheduler is a preemptive scheduler for some tasks which are occupying GPU. The TaskPool is its core data structure. Tasks in the TaskPool have priorities which will change dynamically. Formulas (3)–(5) describe how to calculate the priority. There are estimated runtime Q_{time} and required GPU resources Q_{res} submitted by users. Dishonest users who submit cheat info will be punished in post-audit. In this way, every Pod gets a default priority called $P_{default}$ following the Formulas (3) and (4). In Formula (4), w_i is the resource weight, and C_i is the number of this resource that users request. In runtime, tasks priorities P_{task} are dynamically adjusted every 10 min following Formula (5) in which W_t is the waiting time, and R_t is the running time. The pause and resume costs for GPU tasks are expensive, so a task will keep running at least an hour before its pause even if its priority was lower than others. The formula shows that when a task waits longer, its priority is higher. The longer the running time, the lower the priority.

$$P_{default} = \frac{1}{Q_{time} * Q_{res}} \tag{3}$$

$$Q_{res} = \sum w_i * C_i, i \in resourcelist \tag{4}$$

$$P_{task} = P_{default} * (1 + \frac{W_t}{max(R_t, 1)}). \tag{5}$$

All active tasks are stored in the TaskPool and these tasks are sorted by priority. Dynamic Scheduler will try to pause a set of tasks whose priorities are lower than paused tasks from high priority to low priority. In the ideal situation, the tasks can be deployed directly to those released resources. But in a very few cases, resource constraints of some tasks cannot be satisfied and that will lead to a migration of some tasks. When a task is going to pause, it has 10 min to store its states in a distributed file (Swift + NFS) after receiving a pause signal.

6. Experiments and Results

Experiments of Batch Scheduler are made in Alibaba scheduling dataset [27]. The dataset is composed of three files, which are used to describe the information of apps, nodes, and constraint rules. The results are listed below.

Table 1 shows the comparison between our algorithm and the default Scheduler for batch tasks. Resource utilization and layout score are important indicators. Resource utilization is formulated as $\frac{Cost_{used}}{Cost_{total}}$. During this experiment, the weights of I/O, CPU, and MEM resources are set as 10, 4, and 2, respectively. Based on these data, the value of $Cost_{total}$ can be calculated. Besides, there is a soft constraint to make tasks more robust, that the pods in a task were supposed to deploy on different machines, so the layout score formula is $\sum a \sum A(c_{Aa} - 1) * cost_a(c_{Aa} > 1)$. c_{Aa} is the total number of containers of application a on machine A and $cost_a$ is the cost of the container of application a on the same machine. The $cost_a$ is a const with value 1 in Alibaba dataset which will be improved in the future research to fulfill different application requirements. Accordingly, the smaller the value of the layout score, the better the layout.

Table 1. Batch tasks.

	Default Scheduler				**Our Algorithm**			
Nodes	776	815	847	945	776	815	847	945
Apps	382	402	420	413	382	402	420	413
Pods	7423	8021	8210	9241	7423	8021	8210	9241
Resource utilization	69.20%	57.30%	67.40%	59.70%	91.90%	84.20%	89.60%	87.40%
Layout score	1158	1317	1295	1272	1065	1174	1168	1253

The Dynamic Scheduler is tested in a dataset generated by our team. This dataset is based on the deep learning tasks running in our laboratory. To compute the value of $P_{default}$, users should submit approximate estimated running time. In general, the running time is estimated by tasks FLOPS. These tasks shall be tested in submitters' local GPU with a small dataset to estimate whole FLOPS. For instance, the Network Architecture Search (NAS) task trained by small dataset which consumed 600T FLOPS is running 20 min in a local GTX960 card produced by Colorful, ShenZhen, China, so the task's running time for whole dataset which will consume 3E FLOPS is estimated as 40 h in a cloud GTX2080 card by submitter. When we calculate Q_{res}, GPU is the most important resource, so the weights of GPU, CPU, MEM resources are set as 10, 2, 1, respectively.

The results are listed in Table 2. For GPU tasks, we compare the algorithms from the resource utilization and the average task waiting time, as shown in Table 2. The average waiting time is formulated as $\frac{\sum_{i \in apps} t_i}{c(apps)}$, which means the ratio of the sum of the waiting time of all tasks to the number of tasks. Because of the priority and preemption mechanism, we can schedule more tasks per unit time. So, the average waiting time of our algorithm is better than the default Scheduler.

Experiments show that these two scheduling strategies proposed in this article improve resource utilization and increase task throughput.

Table 2. GPU tasks.

	Default Scheduler				**Our Algorithm**			
Nodes	46	49	53	60	46	49	53	60
Apps	102	106	122	152	102	106	122	152
Pods	311	305	364	339	311	305	364	339
Resource utilization	60.70%	57.50%	65.30%	61.20%	85.40%	79.30%	81.20%	83.60%
Waiting time	7.5 h	6.1 h	6.3 h	7.1 h	2.3 h	1.6 h	1.7 h	2.2 h

7. Conclusions

With the rapid development of the containerization technique, CaaS is becoming more and more popular in cloud computing services. For computer science laboratories in universities, the resource is relatively limited and the types of services are diverse. To fulfill the actual needs of our labs, we have proposed scheduling algorithms which are exploited in appropriate situation for different tasks and achieved good results in terms of resource utilization and waiting time. Despite all this, there is additional work to improve our algorithms. For example, we currently need users to provide an approximate running time, but we hope that the system can be independent on this variable. Further, our system should better detect different scenarios to switch the scheduling algorithm instead of relying heavily on manual operations. In the future, we will improve and extend our work in several directions. Firstly, we will make the scheduling algorithms more automated and stable by repairing the above defects. In addition, we will continue to research on scheduling strategies and keep track on new technologies, such as applying deep learning with scheduling.

Author Contributions: All authors contributed to the development and completion of this paper. Conceptualization, Z.W. and K.W.; methodology, H.L. and L.H. (Laipeng Han); software, L.H. (Lan Huang) and Z.W.; validation, H.L. and Z.W.; resources, Z.W. and K.W.; writing–original draft preparation, H.L.; writing–review and editing, Z.W., L.H. (Lan Huang) and K.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Jilin Province Science and Technology Development Plan Project grant number 20190201273JC.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to copyright considerations.

Acknowledgments: The authors would like to thank the Jilin Provincial Key Laboratory of Big Data Intelligent Computing for their technical and resource support.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CaaS	Software-as-a-Service
BCDI	Balanced-CPU-Disk-IO-Priority
DRF	Dominant Resources Fairness
LDAP	Lightweight Directory Access Protocol
NFS	Network File System
FLOPS	Floating-point Operations Per Second
NAS	Network Architecture Search

References

1. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and Linux containers. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015.
2. Jennings, B.; Stadler, R. Resource Management in Clouds: Survey and Research Challenges. *J. Netw. Syst. Manag.* **2015**, *23*, 567–619. [CrossRef]
3. Liu, P.; Hu, L.; Xu, H.; Shi, Z.; Tang, Y. A Toolset for Detecting Containerized Application's Dependencies in CaaS Clouds. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 2–7 July 2018.
4. Dragoni, N.; Giallorenzo, S.; Lluch-Lafuente, A.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*; Springer: Cham, Switzerland, 2017.
5. Singh, V.; Peddoju, S.K. Container-based microservice architecture for cloud applications. In Proceedings of the 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, 5–6 May 2017; pp. 847–852. [CrossRef]
6. Balalaie, A.; Heydarnoori, A.; Jamshidi, P. Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture. *IEEE Softw.* **2016**, 33, 42–52. [CrossRef]
7. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [CrossRef]
8. Netto, H.V.; Lung, L.C.; Correia, M.; Luiz, A.F.; Moreira, S.; De Souza, L. State machine replication in containers managed by Kubernetes. *J. Syst. Archit.* **2017**, *73*, 53–59. [CrossRef]
9. Medel, V.; Tolosana-Calasanz, R.; Bañares, J.; Arronategui, U.; Rana, O.F. Characterising resource management performance in Kubernetes. *Comput. Electr. Eng.* **2018**, *68*, 286–297. [CrossRef]
10. The Default Scheduler in Kubernetes. 2020. Available online: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (accessed on 9 November 2020).
11. Ghodsi, A.; Zaharia, M.; Hindman, B.; Konwinski, A.; Shenker, S.; Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In Proceedings of the NSDI 2011, Boston, MA, USA, 30 March–1 April 2011.
12. Mesos: Dominant Resources Fairness. 2020. Available online: http://mesos.apache.org/documentation/latest/app-framework-development-guide/ (accessed on 9 November 2020).
13. Grandl, R.; Ananthanarayanan, G.; Kandula, S.; Rao, S.; Akella, A. Multi-resource Packing for Cluster Schedulers. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 455–466. [CrossRef]
14. Menouer, T. KCSS: Kubernetes container scheduling strategy. *J. Supercomput.* **2020**. [CrossRef]
15. Li, D.; Wei, Y.; Zeng, B. A Dynamic I/O Sensing Scheduling Scheme in Kubernetes. In Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications, Guangzhou, China, 27–29 June 2020; pp. 14–19. [CrossRef]
16. Zhang, W.; Ma, X.; Zhang, J. Research on Kubernetes' Resource Scheduling Scheme. In Proceedings of the 8th International Conference on Communication and Network Security, Qingdao, China, 2–4 November 2018.
17. Orhean, A.I.; Pop, F.; Raicu, I. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *J. Parallel Distrib. Comput.* **2017**, *117*, 292–302. [CrossRef]
18. Huang, J.; Xiao, C.; Wu, W. RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning. In Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E), Sydney, Australia, 21–24 April 2020.
19. Yu, G.; Christina, D. The Architectural Implications of Cloud Microservices. *IEEE Comput. Archit. Lett.* **2018**, *17*, 155–158.
20. Carullo, G.; Mauro, M.D.; Galderisi, M.; Longo, M.; Postiglione, F.; Tambasco, M. Object Storage in Cloud Computing Environments: An Availability Analysis. In Proceedings of the 12th International Conference, GPC 2017, Cetara, Italy, 11–14 May 2017.
21. Jaramillo, D.; Nguyen, D.; Smart, R. Leveraging microservices architecture by using Docker technology. In Proceedings of the SoutheastCon 2016, Norfolk, VA, USA, 30 March–3 April 2016; pp. 1–5. [CrossRef]
22. Guan, X.; Wan, X.; Choi, B.Y.; Song, S.; Zhu, J. Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers. *IEEE Commun. Lett.* **2017**, *21*, 504–507. [CrossRef]
23. Torroglosa-Garcia, E.; Perez-Morales, A.D.; Martinez-Julia, P.; Lopez, D.R. Integration of the OAuth and Web Service family security standards. *Comput. Netw.* **2013**, *57*, 2233–2249. [CrossRef]
24. Vasanthi, N.A. LDAP: A lightweight deduplication and auditing protocol for secure data storage in cloud environment. *Clust. Comput.* **2017**, *22*, 1247–1258.
25. Chae, C.J.; Kim, K.B.; Cho, H.J. A study on secure user authentication and authorization in OAuth protocol. *Clust. Comput.* **2017**, *22*, 1991–1999. [CrossRef]
26. Rattihalli, G.; Govindaraju, M.; Lu, H.; Tiwari, D. Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019.
27. The Alibaba Dataset. 2020. Available online: https://code.aliyun.com/middleware-contest-2020/django (accessed on 9 November 2020).