Hindawi
Scientific Programming
Volume 2021, Article ID 6397786, 10 pages
https://doi.org/10.1155/2021/6397786

Research Article

A Fine-Grained Horizontal Scaling Method for Container-Based Cloud

Chunmao Jiang and Peng Wu

School of Computer Science and Information Engineer, Harbin Normal University, Harbin, Heilongjiang 150025, China

Correspondence should be addressed to Peng Wu; 864782389@qq.com

Received 26 September 2021; Revised 5 November 2021; Accepted 9 November 2021; Published 27 November 2021

Academic Editor: Punit Gupta

The container scaling mechanism, or elastic scaling, means the cluster can be dynamically adjusted based on the workload. As a typical container orchestration tool in cloud computing, Horizontal Pod Autoscaler (HPA) automatically adjusts the number of pods in a replication controller, deployment, replication set, or stateful set based on observed CPU utilization. There are several concerns with the current HPA technology. The first concern is that it can easily lead to untimely scaling and insufficient scaling for burst traffic. The second is that the antijitter mechanism of HPA may cause an inadequate number of onetime scale-outs and, thus, the inability to satisfy subsequent service requests. The third concern is that the fixed data sampling time means that the time interval for data reporting is the same for average and high loads, leading to untimely and insufficient scaling at high load times. In this study, we propose a Double Threshold Horizontal Pod Autoscaler (DHPA) algorithm, which fine-grained divides the scale of events into three categories: scale-out, no scale, and scale-in. And then, on the scaling strength, we also employ two thresholds that are further subdivided into no scaling (antijitter), regular scaling, and fast scaling for each of the three cases. The DHPA algorithm determines the scaling strategy using the average of the growth rates of CPU utilization, and thus, different scheduling policies are adopted. We compare the DHPA with the HPA algorithm under different loads, including low, medium, and high. The experiments show that the DHPA algorithm has better antijitter and antiload characteristics in container increase and reduction while ensuring service and cluster security.

1. Introduction

The rapid growth of container technology requires effective deployment and management strategies for containerized applications while addressing their runtime adaptability. In addition, the ability of cloud computing to provide resources on demand encourages the development of elastic applications that can accommodate changes in working conditions (e.g., variable workloads). Horizontal elasticity allows increasing (scaling-out) and decreasing (scaling-in) the number of application instances (e.g., containers) [1]. Most of the existing horizontal scaling methods explore resilience, which respond quickly to small load changes [2–4]. In this study, we build fine-grained horizontal scaling to cope with sudden load peaks.

As two crucial quantitative metrics, response time and resource utilization are essential measurements for various load variations under dynamic environmental conditions [2]. Container-based virtualization technology can improve application performance and resource utilization more efficiently than virtual machines (VM). Many existing scaling mechanisms employ fixed thresholds, which are based on cloud platform metrics, in general, such as CPU utilization. In contrast, such an approach is widely used, including Amazon's EC2, a virtual machine-based cloud platform. However, for applications that are constantly changing their requirements for CPU, memory, and other resources, their performance and resource utilization decrease significantly [5–7].

The adaptation of advanced metrics and dynamic thresholds may respond more finely to fluctuations in the workload, so it can improve application performance and get higher resource utilization. Therefore, we hope to develop a

new dynamic autoscaling approach that automatically adjusts the thresholds based on the state of the execution environment observed by the monitoring system. In this way, the monitoring information, including infrastructure and application-specific metrics, will help the service provider to accomplish a satisfactory adaptation mechanism to various operational states. Furthermore, fine-grained scalability thresholds and degrees of scalability can better improve resource utilization and better cope with dynamic workload variations.

Therefore, this study aims to develop a new fine-grained dynamic scaling method based on the thought of granular computation. The major contributions of this study are as follows: first, we classify the container scaling events into three categories by establishing two thresholds, i.e., scale-out, neither scale-out nor scale-in, and scale-in. Second, we further subdivide the scaling strength into three levels for the scaling events, i.e., no scaling (to prevent jitter), regular scaling, and fast scaling. Third, the scalability metric applied in this study considers not only CPU utilization but also the growth rate of CPU utilization. We validate the algorithm's effectiveness by simulation under low load, medium load, and high load scenarios, respectively. The results show that the proposed algorithm in this study can resist high load and jitter well and effectively guarantee the cluster's quality of service (QoS).

The remainder of this study is organized as follows. Section 2 reviews the horizontal scaling mechanism of container clouds and presents the limitations that currently exist in Kubernetes. In Section 3, we present the DHPA algorithm, which is a dual-threshold horizontal scaling algorithm. And then, a specific example is given to illustrate the idea and process of the DHPA algorithm. Section 4 gives the experiment result and analysis. Finally, we conclude this study and prospective future studies in Section 5.

2. Related Work

Kubernetes [8–10] offers Horizontal Pod Autoscaler (HPA) [11–13], a built-in horizontal scaling controller, which automatically scales the ReplicaSet controller, deployment controller, or pod quantity based on statistical CPU utilization (or other custom metrics). This section presents the Kubernetes' horizontal scaling technique, including the acquisition of HPA metrics, how it works, and its limitations.

2.1. Horizontal Pod Autoscaler. HPA is a cyclic control process. The controller manager queries resource utilization during each cycle based on the metrics specified in each Horizontal Pod Autoscaler definition.

The controller manager can retrieve data from the following sources: (1) gather CPU utilization and memory usage data from Heapster, (2) use the Resource Metrics API to collect data from the Metrics Server that contains resource metrics for each pod in the cluster, and (3) the Custom Metrics Adapter provides the data collected by third-party plug-ins such as Prometheus to the Custom Metrics API, which the cluster then uses to fetch the data. In the latest version of Kubernetes, the cluster introduces a new data reporting channel—aggregation layer, an abstract data reporting interface that third-party plug-ins or administrators can use to implement this interface themselves. The approach of HPA to acquire data is shown in Figure 1.

2.2. How HPA Works. The principle of HPA is to poll resources of each pod every 30 seconds to determine whether the number of copies of the target pod needs to be adjusted by statistically analyzing the changes in the load of the target pod. There are two approaches to HPA to calculate the number of targets that the pod needs to scale-out or scale-in.

2.2.1. CPU Utilization Percentage. CPU utilization percentage represents the average CPU utilization of all copies of the current pod. A Pod's CPU utilization is the Pod's current CPU usage divided by the Pod Request value [14]. The calculation of the target number of pods for a scaling capacity is given by

$$\mathrm{ER} = \mathrm{ceil}\left[\mathrm{cR} * \left(\frac{\mathrm{cV}}{\mathrm{dV}}\right)\right], \tag{1}$$

where ER (expect replicas) represents the expected number of pods needed for expansion. The cR (current replicas) represents the number of pods in the current state. The cV (current value) represents the metrics that are currently be detected, such as memory usage, CPU utilization, and HTTP request traffic. The dV (desired value) represents the threshold for scaling up or scaling down, and Ceil represents the value, which is the nearest integer that is greater than or equal to the dV. Suppose the value of CPU utilization percentage exceeds 80% at a given moment. In that case, it means that the current number of pod copies is likely insufficient to support more subsequent requests, and dynamic scaling is required. When the request peak passes, the CPU utilization of Pod drops again, and HPA will reduce the number of pod copies to a reasonable level.

2.2.2. Application-Based Defined Metrics. CPU utilization percentage is implemented by the Heapster plug-in when calculating the CPU usage of the Pod, but adding a plug-in increases the complexity of the system while decreasing the efficiency of HPA's scaling. Kubernetes supports using custom metrics as metrics starting with version 1.2, which requires the given properties such as the metric units and how the metrics data are obtained. This mechanism is not widely used yet. The HPA control is illustrated in Figure 2.

The workflow of HPA can be summarized as follows. HPA will fetch the metrics data in the cluster every 30 seconds. Suppose the fetched metrics exceed the initial threshold. In that case, the HPA starts counting the number of target pods, and the HPA controller sends a command to the corresponding controller of the pods (ReplicaSet and deployment). The controller recycles or scales out the number of pods according to the number of target pods. After the operation of the pod is completed, the service layer

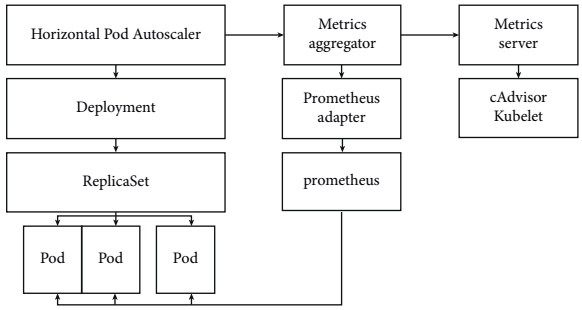

Figure 1: Flowchart of HPA collective data.

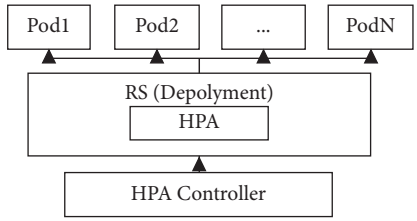

Figure 2: HPA system.

inside Kubernetes will automatically perform load balancing operations for the scaled-up or scaled-down pods. At this point, HPA has completed the entire horizontal scaling operation, and the scaling flowchart is shown in Figure 3.

2.3. Limitations Analysis of HPA. By analyzing the Kubernetes source code, we found that the HPA system implementation is relatively simple and has some limitations.

(1) The algorithm used by HPA for expansion and contraction is based on equation (1), which is simple to implement and inflexible. For example, suppose there are many network requests instantaneously. In that case, HPA will scale out, but it needs time and resources to start a pod service. Suppose the scale-out is not timely, or the number of scale-out is insufficient. It may seriously crash service and even threaten the cluster's security.

(2) Due to HPA's antijitter mechanism, the cluster will not be rescaled within 3 minutes after an expansion, which may result in an inadequate expansion. The number of containers cannot meet the subsequent service requests. The quality of service will be severely degraded or even collapse, which significantly affects the user experience and even cluster security. Simultaneously, there will not be any scaled operations within 5 minutes. If the scale occurs when traffic peaks to arrive again, the pod copy is not enough, which will eventually lead to a decline in the quality of service, cluster crash, and other issues.

(3) HPA fixes the time of data sampling to save resource consumption. The data reporting interval is the same during regular and high load periods, seriously affecting the cluster's access to information about the entire load during high load. The mechanism makes the cluster unable to correctly estimate the current pod load, prone to untimely and inadequate capacity expansion.

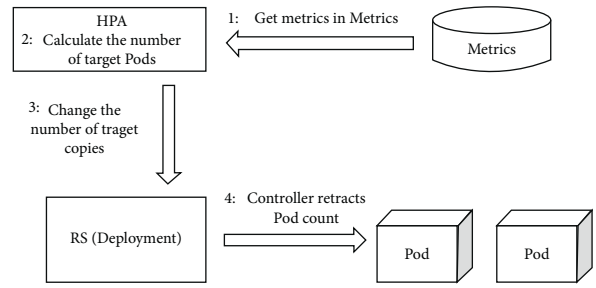

Figure 3: HPA flow chart.

A summary of the related work is shown in Table 1.

3. Dual-Threshold Horizontal Scaling Algorithm

In this section, we present a dual-threshold-based scaling algorithm (DHPA) and analyze the algorithm through an example.

3.1. The Basic Idea of DHPA. The basic idea of the DHPA algorithm is to divide the container scaling into finer granularity by introducing the idea of granular computation. First, a threshold is set for scale-out and scale-in, respectively, and the two thresholds divide a scaling event into three parts: scale-out, no scale-in, and scale-in. The scaling strength is also subdivided as follows: no scale-out, normal scale-out and scale-in, and fast scale-out and scale-in. This fine-grained division of the container scale-out and scale-in capacity problems can be an excellent solution to the problems mentioned above, and the algorithm implementation steps are as follows:

(1) In the DHPA algorithm, there will be no longer mechanisms such as no more expansion within 3 minutes and no more expansion within 5 minutes of shrinkage. DHPA will use dynamic antijitter measures in place with the original static antijitter mechanism.

(2) The DHPA algorithm dynamically adjusts the reporting time of cluster monitoring pod data, which is subdivided into three granularities, i.e., at low load, the reporting time is 30 seconds. For medium load, it is 10 seconds. For high load, the data uptime is once every 1 second. This mechanism improves the mastery of the pod load situation of this algorithm under different load cases, allowing for better control of the system's scaling operations.

(3) The DHPA algorithm dynamically adjusts the pod's expansion by triangulating the pod expansion situation. It performs no expansion operation when the fluctuation of the pod load changes little. When the fluctuation variation is moderate, it performs the regular expansion operation. If the pod's load

Table 1: Overview of various HPA for container.

Virtualization	Basis	Metrics	Method	Ability
Container	CPU and memory	Time and throughput	Control theory	Dynamic
Container	CPU	Nothing	Rule-based	Static
VM and container	CPU and bandwidth	Application throughput	Rule-based	Static
VM and container	CPU	Nothing	Rule-based	Static
Container	CPU, memory, and bandwidth	Time and throughput	Rule-based	Dynamic

fluctuation varies sharply, the algorithm will perform a robust expansion operation to meet the pod's load demand. This case will reduce the number of expansion resources wasted because of jitter and fully consider the expansion under different load conditions.

(4) During capacity reduction, the DHPA algorithm also dynamically adjusts the capacity reduction range of pods. It can effectively reduce the frequent expansion and reduction problems caused by the sudden increase in the load after the load drop and reduce the business crash caused by the antijitter problem.

3.2. Scheduling Algorithm

Definition 1. (base threshold). Let α and β represent two thresholds, which are used to adjust the capacity of provided pods.

When the DHPA monitors the current pod's CPU utilization U over α, it changes the monitoring time from 30 seconds to 10 seconds and starts the capacity expansion judgment. If the CPU utilization U of the monitored pod exceeds β, we change the refresh rate to 1 second.

Definition 2. A pod's CPU utilization queue is set, where n is customizable and in this study is provisionally defined as 3. The larger the value, the better the antijitter effect, but the more stringent the scaling conditions will be.

Definition 3. Two thresholds δ and β are defined, satisfying $0.1 < \delta < \beta$, and c and d are the two critical granularity thresholds used by the DHPA algorithm to determine the strength of the expansion and contraction. We suggest that a and b take 40% of their range of values, while d and e are suggested to be 70% of their range of values. The developer can determine the most appropriate threshold value by conducting experiments in their cluster.

Let $\Delta_n = (x_n - x_{n-1})/x_{n-1}$ be the growth rate between two neighboring CPU utilization rates in the CPU utilization queue. $\varphi_n = (\Delta_2 + \Delta_3 + \cdots + \Delta_{n-1} + \Delta_n)/n$ is the average of the growth rate of CPU utilization. The DHPA algorithm as follows addresses the above scaling problem and formulates scheduling algorithms for each of the three granularities in the scaling case.

The process of scaling up a container can be outlined as follows. For a given CPU utilization history queue $P = \{x_1, x_2, \ldots, x_n\}$, we first compute each item Δ_i in queue P, if not all of Δ_i are greater than δ, or one x_i is not greater than α, i.e., $\exists\Delta_i < \delta$ or $\exists x_i < \alpha$; then, the cluster will not be scaled up because the algorithm will determine it to be a normal jitter for pod services. If each Δ_i is greater than δ, but there is one Δ_i is not greater than α, or each utilization x_i in the queue is greater than α, i.e., $\forall\Delta_i < \delta$ and $\exists\Delta_i < \varepsilon$ or $\forall x_i > \alpha$, then the DHPA algorithm determines it as a normal cluster load rise and performs the normal scaling up, and the number of scaled-up pod copies is computed according to the following equation:

$$\mathrm{ER} = \mathrm{ceil}\left[\mathrm{cR} * \left(\frac{\mathrm{cV}}{\alpha}\right)\right]. \tag{2}$$

If the growth rate of each is greater than ε, and each xi in the queue is greater than α, that is, $\varepsilon < \Delta_2 < \Delta_3 < \ldots < \Delta_{n-1} < \Delta_n$, $\forall\Delta_i < \varepsilon$, and $\forall\Delta_i < \alpha$, then the algorithm determines that the traffic peak is about to come; therefore, this strategy adopts emergency expansion. The number of needed to expansion copies of the pod according to equation is as follows:.

$$\mathrm{ER} = \mathrm{ceil}\left[\mathrm{cR} * \left(\frac{\mathrm{cV}}{\alpha}\right) * \left|\varphi_n\right| * 10\right]. \tag{3}$$

The scaling-up strategy of the DHPA algorithm is summarized in the following equation:

$$\begin{cases} \exists\Delta_i < \delta, \quad \text{or, } \exists x_i < \alpha & \text{no expansion} \\ \left|\forall\Delta_i > \delta, \quad \text{and, } \exists\Delta_i < \varepsilon, \quad \text{or, } \forall x_i > \alpha\right. & \text{normal expansion} \\ \varepsilon < \Delta_2 < \Delta_3 < \ldots < \Delta_n, \quad \text{and, } \forall\Delta_i < \varepsilon, \quad \text{and, } \forall x_i > \alpha & \text{rapid expansion} \end{cases} \tag{4}$$

Similarly, we give the following procedure for container scaling down. If there is a Δ_i that is greater than 0, or there is a Δ_i greater than $-\delta$, i.e., $\exists\Delta_i > -\delta$ or $\exists\Delta_i > 0$, the algorithm determines that this is a normal cluster load fluctuation and does not perform a scale-down operation. If each Δ_i is less than $-\delta$, there is one Δ_i that is greater than $-\varepsilon$, or each utilization x_i in the queue is less than α, i.e., $\forall\Delta_n < -\delta$ and $\exists\Delta_n > -\varepsilon$ or $\forall x_i < \alpha$. The algorithm determines that this is a normal cluster

load drop and performs a normal scaling-down operation, and the number of shrunken pods is calculated according to the following equation:

$$\mathrm{ER} = \mathrm{ceil}\left[\mathrm{cR} * \left(\frac{\mathrm{cV}}{\alpha}\right)\right]. \quad (5)$$

If each of Δ_i is less than $-\varepsilon$, and at the same time each x_i in the queue is less than α, that is, at this point, the cluster load drops faster, this time you can do a quick scaling down, in order to save resources, scaling down the number of copies of the pod according to the following equation:

$$\mathrm{ER} = \mathrm{ceil}\left[\mathrm{cR} * \left(\frac{\mathrm{cV}}{\alpha}\right) * \left|\varphi_n\right| * 10\right]. \quad (6)$$

The time complexity of the DHPA algorithm is mainly focused on the polling step of the cluster load. Suppose the n represents the number of copies of each pod in the container cluster and u represents the cluster load at each moment. In the scaling process, each time needs to traverse the n copies of the cluster, so the time complexity of the DHPA algorithm is $O(N)$. The DHPA algorithm has a CPU utilization list and a pod list, each with a finite number of internal objects, so the space complexity of the DHPA algorithm is $O(N)$.

3.3. An Illustrative Example. This section gives an example of the DHPA algorithm. In the example, we use sin function to simulate the CPU utilization of a set of pods per second as shown in equation

$$U_{\mathrm{t}} = 200 * \sin(t), \quad (7)$$

where t represents the times (second), the entire experiment lasts 180 seconds $t = \{1, 2,\ldots,180\}$, and then, the utilization for each second is $U_t = \{0, 3, \ldots, 200\} \cup \{200, \ldots, 3, 0\}$, thus simulating the trend of the pod's CPU utilization. Suppose $\alpha = 50$ and $\beta = 70$; these two basic thresholds are used to dynamically adjust the data reporting time of CPU utilization. Suppose $\delta = 0.1$ and $\varepsilon = 0.3$; these two granularity thresholds are used to determine the increase or decrease in the CPU utilization queue to determine the scaling effort. The RT can be used to represent the cluster data reporting interval. $p = \{x_1, x_2, \ldots, x_n\}$ represents the queue that holds the CPU utilization history. Then, $n = 3$ in this case.

(1) The experiment starts from 1 second, and $U_t = 3.49$ according to equation (7). According to the algorithm, we derive the current CPU utilization data reporting time $RT = 30$, which U_t is not reached α at this time, and the historical rate of change in the utilization has not reached δ or ε, therefore, not scaling up and scaling down.

(2) After an interval of 30 seconds, $U_t = 99$, and $P = \{3.49, 99.9\}$, the CPU utilization exceeds α, but the rate of change of the historical CPU utilization has not yet reached δ or ε, so do not perform a capacity scaling up. The RT is modified by 1 because$U_t = 99 > \beta$.

(3) At 31 seconds, $U_t = 103$, and $P = \{3.49, 99.9, 103\}$, the CPU utilization exceeds α, but the rate of change of the historical CPU utilization in the middle has not reached δ or ε, so do not perform a capacity scaling up.

(4) At 32 seconds and $P = \{99.9, 103, 105\}$, the CPU utilization has exceeded α, but the rate of change in the historical CPU utilization has not reached ε, so normal expansion. According to equation (2), the approach to calculate the number of copies of the pod should be expanded to 3, and then expansion starts.

(5) Since it takes 5 seconds to expand a container, the container is expanded to 3 copies at 42 seconds, so the expansion operation is completed.

(6) At 150 seconds, $U_t = 99$, and $P = \{105, 103, 99\}$, the CPU utilization is over α, but the rate of change in CPU utilization is less than 0, so the normal shrink operation is performed at this time according to equation (5). The number of copies of the shrink pod should be 2.

(7) Since it takes 5 seconds to shrink one container, at 160 seconds, the container will be shrunk to two, at which point the shrink operation is completed.

4. Experiments and Data Analysis

This section conducts comparative experiments on the DHPA algorithm's effectiveness in low, medium, and high load cases. The number of containers produced by the DHPA algorithm is compared with the number of containers produced by the HPA algorithm and the number of containers theoretically required to analyze the actual performance of the DHPA algorithm in the three load cases.

The experiment was conducted based on a simulator program written in Java. The specific environment was as follows: operating system Windows 10 1909 version, JDK version 1.8, data analysis program using Python language for writing, the data analysis tool Matplotlib version 3.1.1, and NumPy version 1.16.5. In the simulation experiments, the CPU utilization of a single pod was simulated using the sin function as the base data and multiplied by the corresponding multiplier to simulate the CPU utilization under different pressures. Ten experiments were performed for each of three cases, and the average of the experimental data was taken as a sample value.

4.1. Analysis of Experimental Data under Low Load Conditions. This experiment carries out a comparison by simulating the DHPA algorithm and Kubernetes' own HPA algorithm under low load, simulated node 4, node CPU cores for 4 cores, single-core processing power of 2,252 MIPS, node RAM of 16 GB, hard disk capacity 1 T, bandwidth 1,000 MB/s. In this experiment, CPU utilization ranges between 0% and 200%. We set that every second the CPU utilization of the pod is

$$U_t = 200 * \sin(t), \quad (8)$$

where t is the number of seconds, the whole experiment lasts 180 seconds, and the initial number of pods is set to 1. Part of the experimental data is shown in Table 2, where field time represents the time, BeforeUtil represents the real-time CPU utilization, CalPod represents the theoretical calculation of the number of pods, RealPod represents the actual number of pods after the expansion of the algorithm, AfterUtil represents the expansion of the calculation, and IsBreak represents whether the cluster crashes or not (a single pod crashes if its CPU utilization exceeds 100%).

It can be seen from Table 2 that the DHPA algorithm can perform expansion and contraction operations efficiently at all time points under low load.

The simulated experimental data for HPA are shown in Table 3. The HPA algorithm has a gap between the number of pods and the number of computed pods in most of the time points under low load and cannot promptly perform the scaling operation.

As shown in Figure 4, most of the time, the actual number of pods for the HPA algorithm is lower than the number of pods required, and this problem is largely due to the inadequate prediction of the HPA algorithm at the time of capacity expansion and the cooldown time after the expansion and contraction operation. The DHPA algorithm can efficiently expand and contract capacity after a short delay, which is very close to the theoretical number of pods needed by the cluster, which shows that the DHPA algorithm has a great advantage over the HPA native algorithm low load situations.

4.2. Analysis of Experimental Data under Low Load Conditions. In the medium load experiment, we assume that the number of nodes is 20, the number of CPU cores per node is 4, the single-core processing power of 2,252 MIPS, node RAM is 16 GB, hard disk capacity 1 T, and bandwidth 1,000 MB/s. We expand the multiples of the sin function to simulate the CPU utilization of the pod. The experimental CPU utilization ranges between 0% and 1,000%. We set the CPU utilization per second as follows:

$$U_t = 200 * \sin(t), \tag{9}$$

where t is the number of seconds, the entire experiment lasts 360 seconds, the initial pod number is set to 10, and some of the experimental data are shown in Table 4.

There is a small difference between the number of pods scaled by the DHPA algorithm and the theoretical number of pods under medium load, proving that the DHPA algorithm also has good performance under medium load. As shown in Table 5, the HPA algorithm scales out the number of pods that are needed under medium load and the total number of pods that are needed.

From Figure 5, it can be seen that HPA algorithm has a large gap between the number of pods and the actual number of pods needed, so there were several cluster crashes, which show that the HPA algorithm has a large defect in scaling up and scaling down under medium load.

Table 2: Experimental data of the DHPA algorithm under low load.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
30	100	2	2	50	False
60	173	4	4	43	False
90	200	4	4	50	False
120	173	4	4	43	False
150	100	2	3	33	False

Table 3: Experimental data of the HPA algorithm under low load.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
30	100	2	1	100	False
60	173	4	3	58	False
90	200	4	3	67	False
120	173	4	3	58	False
150	100	2	3	33	False

The number of pods produced by the DHPA algorithm is very similar to the actual number of pods needed, so it can be seen that DHPA algorithm performs relatively well in scaling under medium load.

4.3. Analysis of Experimental Data under High Load Conditions. In the high load experiment, we assume that the number of nodes is 40, the number of CPU cores per node is 4, the single-core processing power of 2,252 MIPS, node RAM is 16 GB, hard disk capacity 1 T, and bandwidth 1,000 MB/s. We expand the multiples of the sin function to simulate the CPU utilization of the pod. The experimental CPU utilization ranges between 0% and 2,000%. We set the CPU utilization per second as follows.

$$U_t = 1000 * \sin(t), \tag{10}$$

where t is the number of seconds, the entire experiment lasts 360 seconds, the initial pod number is set to 15, and some of the experimental data are shown in Table 6.

As seen in Table 6, under high load, the DHPA algorithm falls short of the actual number of pods needed because of the container expansion time limit. However, there is a high overlap with the actual number of pods in the overall expansion and contraction trend.

Table 7 shows that the antijitter delay mechanism still constrains the HPA algorithm, and the number of pods scaled out differs significantly from the theoretical number of pods, thus leading to multiple cluster crashes.

As shown in Figure 6, on the one hand, the DHPA algorithm has a lag in the trend of scaling-down capacity compared to the theoretical pod curve, but the overall trend remains consistent. The HPA algorithm, on the other hand, always maintains a lower number of pods, much lower than the actual number of pods needed. Hence, the DHPA algorithm still has a more significant scheduling advantage over HPA in high load situations and can properly schedule the number of containers to ensure the regular operation of the cluster.

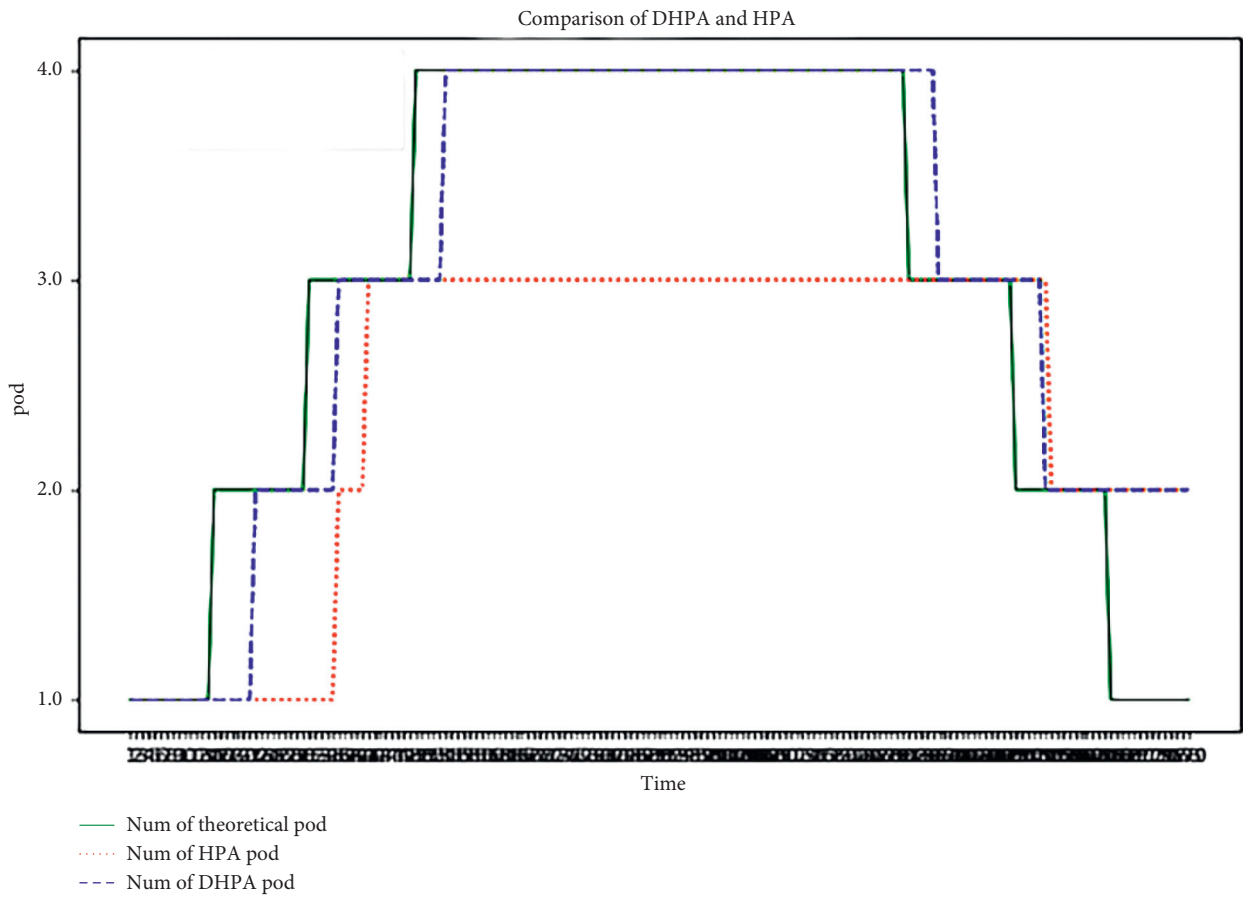

Figure 4: Comparison of the DHPA and HPA algorithms.

Table 4: Experimental data of the DHPA algorithm under medium load.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
60	866	18	16	54	False
100	984	20	21	21	False
160	342	7	11	31	False
230	766	16	11	69	False
280	984	20	21	46	False

Table 5: Experimental data of HPA algorithm under medium load case.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
60	866	18	9	96	False
100	984	20	9	109	Ture
160	342	7	9	38	False
230	766	16	4	191	Ture
280	984	20	4	246	Ture

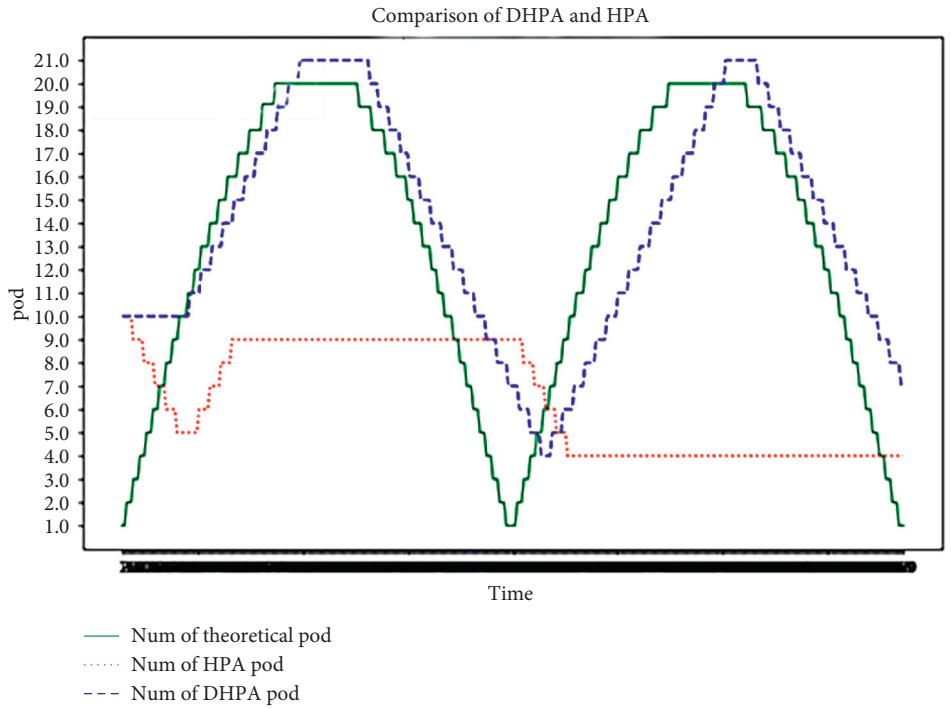

Figure 5: Comparison of the DHPA and HPA algorithms.

Table 6: Experimental data of the DHPA algorithm under high load.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
60	1732	35	22	78	False
100	1969	40	30	65	False
160	684	14	30	22	False
230	1509	31	22	68	False
280	1969	40	32	61	False

Table 7: Experimental data of HPA algorithm under high load conditions.

Time (s)	BeforeUtil	CalPod	RealPod	AfterUtil	IsBreak
60	1732	35	10	173	Ture
100	1969	40	15	131	Ture
160	684	14	15	45	False
230	1509	31	7	215	Ture
280	1969	40	32	281	Ture

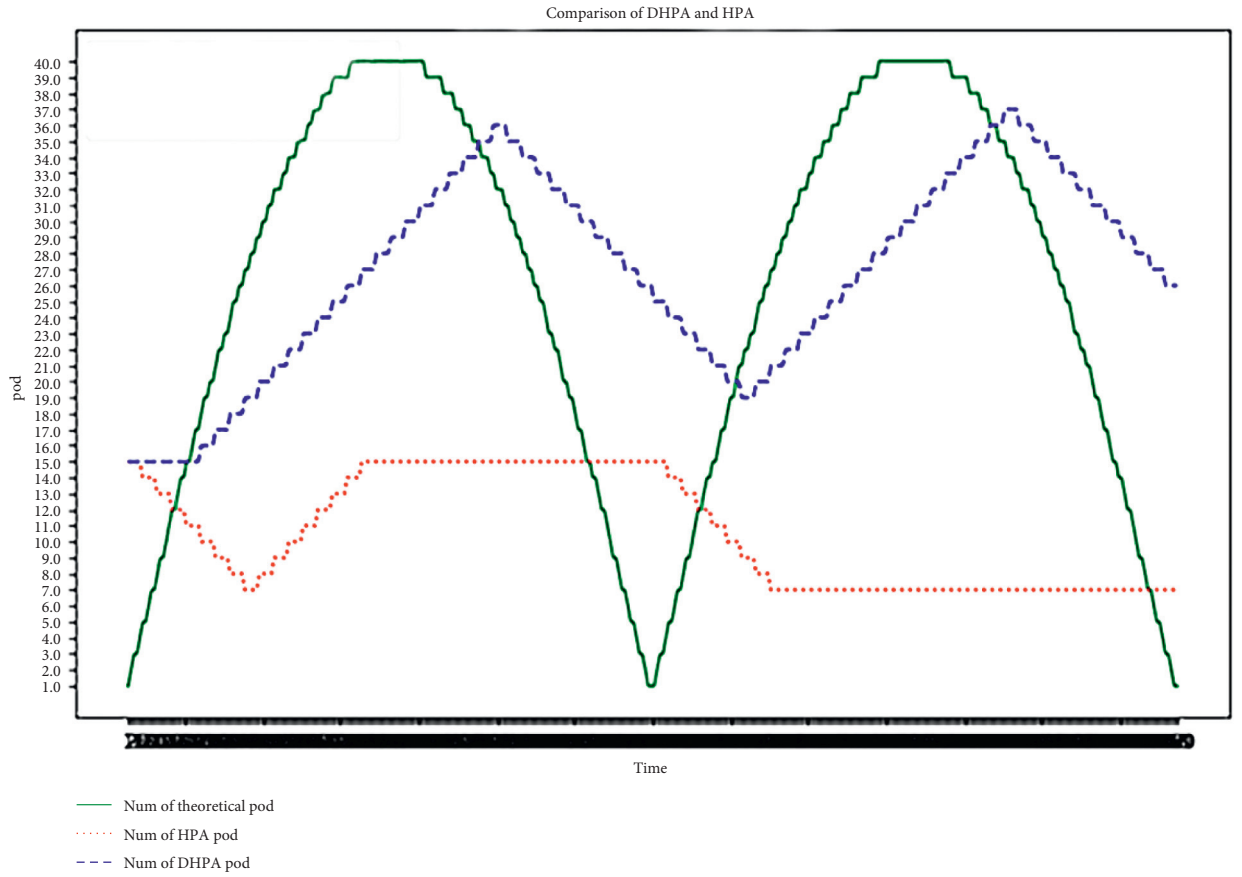

Figure 6: Comparison of the DHPA and HPA algorithms.

5. Conclusions

For highly dynamic workloads in cloud environments, this study proposes a fine-grained horizontal scaling mechanism that can apply dynamic rules to automatically increase or decrease the total number of compute instances to adapt to different workloads. The expansion and contraction operations of the DHPA algorithm are in a dynamic equilibrium state. Because of the pod expansion and contraction time lag, the queue cannot be updated in real time. Each time it scales, it is placed inside the message queue as a single task, so the number of pods dispatched by the algorithm deviates somewhat from the theoretical calculation, but the overall balance is dynamic.

The original HPA algorithm counts how many pods the entire cluster has each time and determines whether to expand or shrink based on the calculated expected pod value. This approach consumes many system resources. In this study, the proposed DHPA algorithm's expansion or contraction operation is based on calculating the growth rate of CPU utilization and on whether the CPU utilization exceeds the threshold to decide by introducing the idea of granularity calculation. Therefore, the DHPA algorithm is to traverse all pods each time in the cluster after calculating whether expansion is needed or not. If there is no expansion or contraction at this point, then there is no need for further operations, which nicely reduces the cluster's performance pressure with each poll. Simultaneous use of two metrics to comprehensively control the expansion and contraction trigger has better stability. The experiments also show that the DHPA algorithm has better antijitter performance in container spreading and shrinking capacity, ensuring the cluster's quality of service and security. In the future, we will try to extend the proposed approach to multi-instance architectures and high-level service customization.

Data Availability

All data used during the study are available in a repository or online in accordance with funder data retention policies (https://archive.ics.uci.edu/ml/datasets.php and http://cs.uef.fi/sipu/datasets/).

Conflicts of Interest

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

Acknowledgments

This work was supported in part by the Natural Science Foundation of Heilongjiang Province (LH2020F031).

References

[1] A. J. Younge, G. Von Laszewski, L. Wang, S. Lopez-Alarcon, and W. Carithers, "Efficient resource management for cloud computing environments," in *Proceedings of the International Conference on Green Computing*, pp. 357–364, IEEE, Chicago, IL, USA, August 2010.

[2] F. Al-Haidari, M. Sqalli, and K. Salah, "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources," in *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science*, pp. 256–261, Bristol, UK, December 2013.

[3] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[4] D. Bernstein, "Containers and cloud: from LXC to docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[5] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *Proceedings of the SoutheastCon 2016*, pp. 1–5, IEEE, Norfolk, VA, USA, April 2016.

[6] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, IEEE, Milan, Italy, July 2019.

[7] K. M. Patel, R. Kandula, B. R. Vempati, H. M. Negalaguli, and P. Chandana, "System and method for elastic scaling using a container-based platform," US Patent 9,462,427, 2016.

[8] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: the impact of relative and absolute metrics," in *Proceedings of the 2017 IEEE 2nd International Workshops on Foundations and Applications of Self ∗ Systems (FAS ∗ W)*, pp. 207–214, IEEE, Tucson, AZ, USA, September 2017.

[9] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, p. 167, Kohala, HI, USA, August 2015.

[10] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology*, vol. 20, no. 2, pp. 1–24, 2020.

[11] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, pp. 1–27, 2020.

[12] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020.

[13] S. K Lin, U. Altaf, G. Jayaputera et al., "Auto-scaling a defence application across the cloud using docker and kubernetes," in *Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 327–334, IEEE, Zurich, Switzerland, December 2018.

[14] F. Rossi, V. Cardellini, and F. L. Presti, "Hierarchical scaling of microservices in Kubernetes," in *Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 28–37, IEEE, Washington, DC, USA, August 2020.