开源云上的 Kubernetes 弹性调度

张可颖[1],彭丽苹[2],吕晓丹[2],吕尚青[3]
(1.贵州大学 大数据与信息工程学院,贵州 贵阳 550025;
2.贵州大学 计算机科学与技术学院,贵州 贵阳 550025;
3.北京邮电大学 信息与通信工程学院,北京 100000)

摘 要: 针对私有云资源弹性调度问题,将 Kubernetes 结合已有 Openstack 云平台,提出一种基于容器的弹性调度策略。一方面,因为 Openstack 虚拟机启动时间较长,给调度带来额外时间开销,所以利用容器拉起时耗远小于虚拟机的特性,用 Docker 容器取代了 Openstack 默认的虚拟机;一方面优化了 Kubernetes 调度算法,建立了一个提高集群资源利用率的优化模型,通过对云平台各个服务器节点四种类型资源的监控和应用队列预设模板匹配,选择调度资源利用率最高的服务器。整个调度过程包括容器应用的初次调度和在线迁移算法。实验结果表明,相比原有 Kubernetes 调度算法和一些其他的调度策略,该调度策略对数据中心资源进行了更细粒度的划分,在保证服务器性能的同时,实现了云平台资源弹性调度,集群资源利用率也得到了提高,同时降低了数据中心能耗。

关键词: 私有云; 弹性调度; Kubernetes; 容器; Openstack
中图分类号: TP319 **文献标识码:** A **文章编号:** 1673-629X(2019)02-0109-06
doi: 10.3969/j.issn.1673-629X.2019.02.023

Elastic Scheduling Strategy for Private Cloud Resource Based on Kubernetes and Openstack

ZHANG Ke-ying[1],PENG Li-ping[2],LYU Xiao-dan[2],LYU Shang-qing[3]
(1.School of Big Data and Information Engineering,Guizhou University,Guiyang 550025,China;
2.School of Computer Science and Technology,Guizhou University,Guiyang 550025,China;
3.School of Information and Communication Engineering,Beijing University of Posts and Telecommunications,
Beijing 100000,China)

Abstract: Aiming at the problem of flexible scheduling of private cloud resources,Kubernetes combined with the existing Openstack cloud platform,we propose a flexible scheduling strategy based on container. On the one hand,because Openstack virtual machine takes a long time to start up and brings extra time cost to scheduling,Docker container is used to replace the default virtual machine of Openstack by taking advantage of the feature that container takes much less time to pull up than virtual machine. On the one hand,the Kubernetes scheduling algorithm is optimized,and an optimization model to improve the utilization rate of cluster resources is established. By monitoring four types of resources of each server node of the cloud platform and applying queue preset template matching,the server with the highest utilization rate of scheduling resources is selected. The whole scheduling process includes the initial scheduling of container applications and the online migration algorithm. Experiment shows that compared with the original Kubernetes scheduling algorithm and some other scheduling strategies,this scheduling strategy divides the data center resources into finer granularity. While ensuring the server performance,it realizes the flexible scheduling of cloud platform resources,improves the utilization rate of cluster resources and reduces the energy consumption of the data center.

Key words: private cloud; elastic scheduling; Kubernetes; container; Openstack

收稿日期: 2018-04-02 **修回日期:** 2018-08-09 **网络出版时间:** 2018-11-15
基金项目: 贵州省科技计划联合基金项目(黔科合 LH 字 [2014] 7636)
作者简介: 张可颖(1997-),女,CCF 会员(82186G),研究方向为云计算;吕晓丹,硕士,副教授,通信作者,研究方向为云计算、算法设计与数据分析。
网络出版地址: http://kns.cnki.net/kcms/detail/61.1450.TP.20181115.1051.090.html

0 引 言

云平台通过虚拟化技术将计算机资源整合成资源池,以按需付费的方式实现了用户对计算资源的弹性需求[1]。云计算发展至今,虚拟化技术一直是云平台中的关键技术[2]。Openstack是完全开源的云操作系统,在近几年已经占有私有云市场,是基于传统虚拟化技术的私有云。传统虚拟化技术启动虚拟机时间过长,在弹性扩容方面存在不足。

容器技术是一种新兴的虚拟化技术[2],它的出现给传统虚拟机虚拟化技术带来了挑战,为构建高效的云平台提供了新思路[3-6]。容器与Openstack云平台的结合受到国内外企业的普遍关注[7],如华为、Easystack、Redhat、Vmware等。Kubernetes是容器编排技术的代表,市场占有率越来越多,数据显示2017年77%企业使用Kubernetes作为容器编排[8]。容器技术的高可扩展性得到了行业内的普遍认可[4-7],根据IBM测试报告显示容器启动时间平均是虚拟机的1/21[8],这给Openstack高效弹性调度提供了新思路。IBM的研究人员比较了虚拟机与Docker容器的性能。他们使用一系列工具模拟CPU,内存,存储和网络资源的工作负载进行测试,结果表明几乎在所有情况下,容器的性能都等于或优于虚拟机。

Kubernetes是Google的Borg开源版本,一个通用的容器调度编排器,是典型的Master-Slave模型,使用一个Master管理多个Node(物理机或者虚拟机)上的容器。通常应用程序被分在一个或者多个容器中执行。Kubernetes Scheduler是运行在Master节点上的调度器,通过监听Apiserver将Pod调度到合适的Node上。调度过程简述如下:

第一步:Predicate,过滤掉不满足资源条件的节点。

第二步:Priority,计算各个节点的CPU和内存使用率权重(目前Kubernetes最多支持CPU、内存和GPU)。使用率越低,权重越高。计算镜像权重,镜像越大,权重越高,倾向于调度到已经有需要用到的镜像的节点。由此来对各个节点打分,以确定它们的优先级顺序,选择打分最高的节点作为Pod运行的Node。

由此可见,Kubernetes调度算法存在很大的局限性。第一,Kubernetes提供了一个只考虑CPU和内存的动态资源配置机制[9]。这是不现实的,因为影响应用程序性能的因素有很多,如网络、I/O和存储。第二,Kubernetes的权重打分机制倾向于将Workload平均分布在各个节点,一方面在资源高效利用方面存在不足,除了应用高峰期,其他时间整个集群都处于低负载状态,同时也增加了数据中心的能耗;另一方面,资源均分一定程度造成资源碎片化,降低了集群资源利用率,也可能造成新进大资源无法部署,永远处于Pending状态[10]。另外,Kubernetes社区尚不成熟,本身在存储、网络和多租户管理方面不完善[6],因而与Openstack结合是对其良好的补充。

针对上述问题,首先将Openstack虚拟机容器化,作为Kubernetes集群中的Docker容器,以获得容器的弹性扩展,高效在线迁移的特性。用Kubernetes自带的挂在卷Volume集群作为后端跨主机的块存储,一定程度保证冷数据的安全性。然后建立一个基于Openstack的Kubernetes集群资源调度优化模型,在综合考虑资源负载和应用服务性能的前提下,对集群资源进行了细粒度划分,实现了Openstack集群的容器化的虚拟机的调度和应用容器的在线迁移。

1 相关工作

对Openstack和Kubernetes结合进行数据中心弹性调度的研究目前较少。但在相关领域,学者和企业进行了大量研究。

Chia Chen等[9]提出了一种基于资源利用率和应用QoS度量指标的Kubernetes资源调配算法,在原本Kubernetes考虑CPU的基础上,又加入了系统其他的资源利用率(如内存和磁盘访问)和QoS指标(如响应时间),在一定程度上完善了Kubernetes的调度机制。唐瑞[11]改进了一种抢占式的Pod调度策略,通过将Pod划分为三个优先级,在资源不足时有效提高了高优先级Pod的运行比例。杨鹏飞[12]提出一种基于训练融合ARIMA模型和神经网络模型的动态资源调度算法,有效提高了Kubernetes调度资源利用率和应用服务质量。彭丽萍等[13]在Ceph集群研究中指出集群除了在应用高峰期外,集群中大多数点都处于低负载状态,这就造成资源浪费并增加了系统能耗,并基于Docker和Ceph加权平均的调度策略,在保证数据安全性的前提下,提出一种尽量使少量节点处于高负载,休眠低负载节点的数据中心节能的弹性调度算法[10]。

在企业界,相关的云平台调度策略有Borg、Yarn、Meros等等。Borg把应用程序分成两类—批处理作业和长服务。批处理作业是类似于MapReduce和Spark的作业,在一定时间内会运行结束,长服务则类似于Web Service、HDFS Service等,可能会永久运行下去,永不停止[14]。Borg对长服务的支持细节未知,因为是闭源的,但是Meros和Yarn对长服务存在以下问题:由于Yarn和Meros和Kubernetes具有相似的打分机制,倾向于将Workload平均分配在集群,会造成长服务永远占着资源,预留资源可能永远不足于分配给新服务的情况。

长服务运行一段时间以后,可能需要的资源会有

动态变化。资源伸缩有两个维度: 一个是横向的,即增加实例数目; 另一方面是纵向的,即原地增加正在运行实例的资源量[15]。

以上的弹性调度策略完善了云平台资源监控的度量,优化了数据中心资源调度算法,在一定程度上提高了数据中心资源利用率。但并没有细分考虑服务类型,仅仅考虑初始资源分配,没有考虑服务运行的时间对集群资源调度的影响,以及应用长时间运行而导致资源碎片化问题。因此,针对长服务,实现了 Openstack 使用 Kubernetes 弹性调度容器化的虚拟机云平台调度的弹性策略。

2　私有云调度模型的建立与求解

模型假设: 假设已知各类应用 CPU、内存、网络和 I/O 的最低需求; 假设容器迁移时存储可靠且启动时间和初次容器拉起时间一样,运行一段时间后不造成额外重启开销。

2.1　Kubernetes 集群系统建模

假设集群内共有 K 台服务器 $\text{server}_i(0 < i < K)$, $i \in Z$,分别用 W_{cpu}、W_{m}、W_{n}、$W_{\text{i/o}}$ 表示 CPU 利用率、内存利用率、网络利用率和 I/O(iops) 利用率。

基于以上,建立一个资源调度优化模型:

$$W_i = \frac{1}{4}(W_{\text{cpu}} + W_{\text{m}} + W_{\text{n}} + W_{\text{i/o}}) \tag{1}$$

$$\delta = \sqrt{\frac{(W_{\text{cpu}} - W_i)^2 + (W_{\text{m}} - W_i)^2 + (W_{\text{n}} - W_i)^2 + (W_{\text{i/o}} - W_i)^2}{4}} \tag{2}$$

$$P_{\text{alloc}} = P_{\text{node}} - (P_{\text{sys}} + P_{\text{kube}} + P_{\text{take}}) \tag{3}$$

$$W_{\text{CPU}} = W_{\text{cpu}} + \lambda t_{\text{wait}} \tag{4}$$

$$W_{\text{M}} = W_{\text{m}} + \lambda t_{\text{wait}} \tag{5}$$

$$W_{\text{N}} = W_{\text{n}} + \lambda t_{\text{wait}} \tag{6}$$

$$W_{\text{I/O}} = W_{\text{i/o}} + \lambda t_{\text{wait}} \tag{7}$$

$$W_{\text{ALL}} = W_{\text{CPU}} + W_{\text{M}} + W_{\text{N}} + W_{\text{I/O}} \tag{8}$$

$$S_i = \left(\frac{P^{\text{cpu}}_{\text{alloc}} - W^{\text{cpu}}_{\text{need}}}{P^{\text{cpu}}_{\text{alloc}}}\right) + \left(\frac{P^{\text{m}}_{\text{alloc}} - W^{\text{m}}_{\text{need}}}{P^{\text{m}}_{\text{alloc}}}\right) + \left(\frac{P^{\text{n}}_{\text{alloc}} - W^{\text{n}}_{\text{need}}}{P^{\text{n}}_{\text{alloc}}}\right) + \left(\frac{P^{\text{i/o}}_{\text{alloc}} - W^{\text{i/o}}_{\text{need}}}{P^{\text{i/o}}_{\text{alloc}}}\right) \tag{9}$$

W_i 表示服务器的平均使用率,由于应用往往对资源要求不是均匀的,一个服务器节点往往在内存占用密集时,CPU、网络或 I/O 处于空闲状态。针对这种情况,对应用类型进行分类排序使服务器负载尽可能均匀,将应用占用最多的资源进行分类,在各个分类内从大到小排列分类资源,形成四个应用链表。δ 描述负载是否均匀。

P_{alloc} 表示系统可分配, P_{node} 表示服务器节点总的可用资源, P_{sys} 表示操作系统保留资源, P_{kube} 表示 Kubernetes 系统保留资源, P_{take} 表示 Kubernetes 已分配资源。当 $P_{\text{take}} = 0$ 表示该服务器上没有任何应用在运行。

模型中 λ 是可调等待时间权重, t_{wait} 是任务的等待时间, W_{ALL} 用于排序等待的待分配任务队列。W_{CPU}、W_{M}、W_{N}、$W_{\text{I/O}}$ 综合考虑了应用负载和等待时间,用于优先级调度通过分类排序每一项资源等待的待分配任务队列。S_i 是改进的打分算法,尽可能将应用部署在得分小,即在保证有预留资源的情况下,提高服务器资源负载,最终达到提高集群资源利用率的目的。

2.2　调度策略

2.2.1　初始调度算法

根据以上系统模型,以 Δt 为时间周期,采集 server_i 上各个已经部署应用及其总占用的真实 CPU、内存、网络和 I/O 的数据。通过对采集的数据进行加工整理,可知服务器适合部署哪种应用。再根据应用对 CPU、内存、网络和 I/O 的最低资源需求可初步判断应用是什么类型。如果服务器任一资源占用率大于 80%,表示该服务器节点处于高负载,则用 Kubernetes 命令 cordon 将此节点设置为不可调度,一方面是为了保证服务器性能,一方面是预留部分资源给已经运行在该节点的应用。也将 $P_{\text{take}} = 0$ 的服务器用 cordon 命令设置为不可调度(未分配容器应用的服务器),随后将该台服务器休眠,以提高集群资源利用率,同时节省数据中心资源。当集群中无法满足新应用最低资源要求,再重新唤醒添加休眠中的一台服务器进入集群,随后使用 uncordon 命令恢复节点为可调度节点。具体调度流程如图 1 所示。

2.2.2　服务器上应用迁移算法

对于服务器实际任意资源占用率大于 90%在 ΔT 时间内(ΔT 由云平台管理员设置),或 $\delta > k$ (k 是衡量服务器各项资源是否负载均匀的可调参数),表示该节点单种资源负载过高,或者四种资源严重不均匀,需要迁移应用。前者为了保证服务器性能和预留安全资源,后者是为提高集群资源利用率。详细的应用迁移过程如下:

$$\delta' = \sqrt{\frac{a(W_{\text{cpu}} - W_i)^2 + b(W_{\text{m}} - W_i)^2 + c(W_{\text{n}} - W_i)^2 + d(W_{\text{i/o}} - W_i)^2}{4}} \tag{9}$$

$$a + b + c + d = 1 \tag{10}$$

假设 $\text{server}_m(0 < m < K)$, $m \in Z$ 上应用容器需要迁移。server_m 上运行容器总数目是 C_{m},参数 a、b、c、d (云平台管理员自行设置根据每种触发迁移的条件)

是为了调整迁移容器的负载，尽可能次数少的迁移应用。计算 $server_m$ 上每个容器的 δ'，按从大到小排列容器。C_i^m 表示 $server_m$ 上排序后的第 i 个容器。$W_m = (W_{cpu}, W_m, W_n, W_{i/o})$ 表示 $server_m$ 上服务器各种资源的负载。$e = (\Delta e_1, \Delta e_2, \Delta e_3, \Delta e_4)$ 表示容器迁移时 CPU、内存、网络和 I/O 的改变量。$price_i^m = t_{first}$ 表示 $server_m$ 中容器 i 的迁移代价，其中 t_{first} 是容器第一次启动时间。如果 $price_i^m > q(\min)(q \in Z)$，则认为容器迁移代价过高，此容器暂时不进行迁移，标注为集群负载波谷迁移。

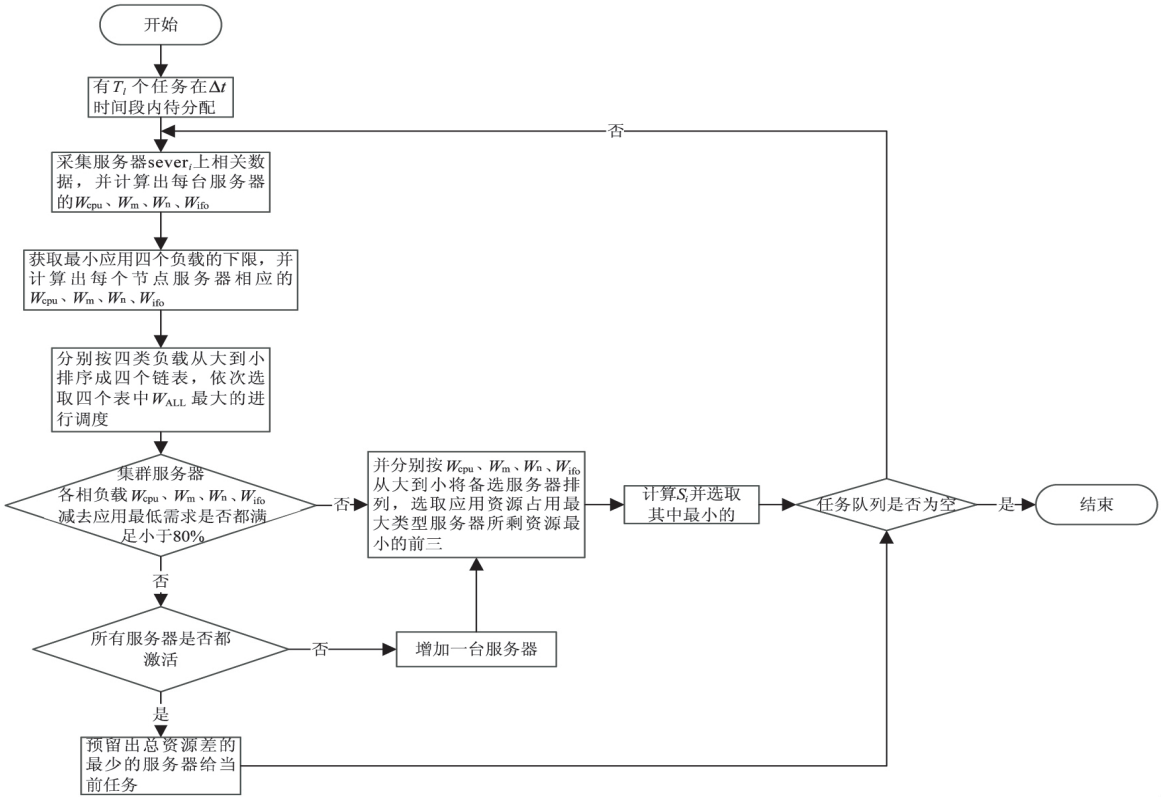

图 1 容器部署流程

选择 $server_m$ 需要迁移容器的具体算法如下：

输入： C_i^m、e_i、W_{cpu}、W_m、W_n、$W_{i/o}$、t_{first}、a、b、c、d

输出： C_i^m

step1：按 $e = (\Delta e_1, \Delta e_2, \Delta e_3, \Delta e_4)$ 的 δ' 将容器从大到小排成队列，依次选取队列容器进行下一步计算；

step2：计算当前容器以及队列前面容器累计迁移应用后 $server_m$ 的各项资源利用率并判断 W_{cpu}、W_m、W_n、$W_{i/o}$ 是否均小于 90%，或者满足 $\delta \leqslant k$。若成立，进行下一步；不成立选择队列中下一个应用，重新计算 step2；

step3：判断需要迁移的每一个容器是否都满足 $price_i^m > q$，若成立，将所选容器都加入到 2.1 中待调度链表中排队。若不满足，去掉不满足的容器资源占用，选择队列中下一个容器，重新计算 step2。

3 实验及对比分析

3.1 实验环境

为了验证调度策略的可行性和有效性，搭建了一个基于 Openstack 的 Kubernetes 集群。在实验中使用 9 台戴尔 R710 服务器（其中 Kubernetes 集群 7 台，一台作为 Master，七台作为 Slave）。K8s 使用 1.9.2 版本，Docker 使用 1.17.03 版本，操作系统为 Centos7.4。

为了模拟云平台负载，用以下工具模拟云平台各项资源负载。

（1）CPU 和内存：对于 CPU 负载使用 Linux 下标准负载测试工具包 Stress-ng 模拟 CPU 和内存负载。

（2）网络：在容器里使用 tomcat 搭建简单网站，使用专业网站测压工具 Apache Bench 模拟网站用户并发访问负载。

（3）I/O：用专业磁盘测试工具 fio 模拟和监测磁盘负载。

3.2 实验过程及对照分析

3.2.1 实验一：容器和虚拟机的开机时间对比

对 Openstack 启动虚拟机和 Kubernetes 启动 centos7.4 容器所需的时间分别进行测试，结果如表 1 所示。

表 1 Openstack 容器和虚拟机开机时间对比 s

组数	Openstack	Kubernetes
第一组实验	10	6
第二组实验	13	4

续表 1

组数	Openstack	Kubernetes
第三组实验	9	4
第四组实验	17	2
第五组实验	11	2
第六组实验	10	2
第七组实验	13	2
第八组实验	14	3
第九组实验	13	3
第十组实验	12	2
平均数	12.2	3.4

实验证明了在此云平台上，容器从创建命令开始到成功创建启动时间是虚拟机创建到启动时间的约四分之一。

3.2.2　实验二: 首次部署调度和迁移对照实验

实验组 1: 容器初次调度算法对照实验。

假设在 Δt 时间内有以下应用，分别按照 Kubernetes 默认算法和改进后的调度算法调度以下应用。用 Centos 命令 sar 获取 CPU 服务器系统占用率; vmstat 获得内存占用数据; 用 iftop 获取网络带宽数据; 用 iostat 获得磁盘 iops 数据，采集时间间隔为 10 s，共采集 10 次，将命令执行结果写成脚本重定向到指定文件，然后整理数据，计算出各项负载平均值作为 server_i 的 W_{cpu}、W_{m}、W_{n}、$W_{\text{i/o}}$。应用情况和部署每一个应用集群内每一台服务器状态转移情况如图 2 所示。

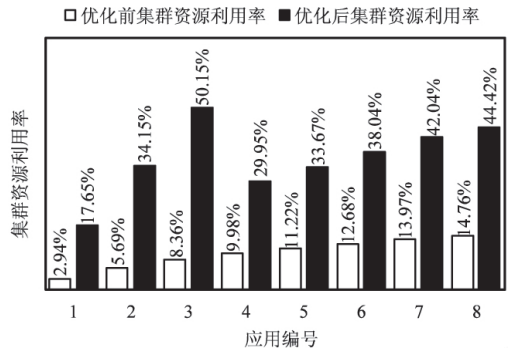

图 2　部署应用过程优化前后集群资源占用率对比

最开始，优化后的算法集群内只有一个激活节点，当部署第三个应用时再激活第二个节点; 而 Kubernetes 自带算法最开始集群内所有节点都被激活，导致整体集群负载率很低，造成服务器资源的大量浪费。

实验组 2: 容器迁移算法。

由于队列中应用任务大多数时都是随机的，且应用真实负载是未知的，server_i 节点在经过一段时间的运行，资源有可能存在不足即某一个或多个资源占用率超过节点资源的 90%，即 $P_{\text{alloc}} < 10\%P_{\text{node}}$，或者节点资源严重分配不均匀，$\delta > k$，此次实验设置 k 为 16。

在时间内随机顺序对 Kubernetes 部署应用，由于应用的不可预知性，所以此时节点真实负载不均匀，需要进行二次调度即迁移调整。

应用运行一段时间后每个节点详细资源使用率如图 3 所示，节点的平均资源使用率和方差如表 2 所示。

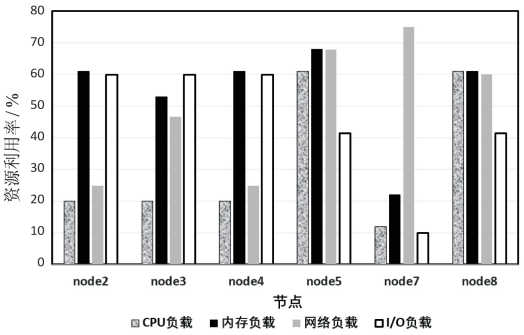

图 3　应用运行一段时间后每个节点详细资源使用率

表 2　应用运行一段时间后每个节点的平均资源使用率和方差

调度节点	W_i /%	δ
node2	41.45	19.08
node3	44.88	15.11
node4	41.45	19.08
node5	59.6	10.83
node7	29.73	26.54
node8	55.85	8.3

将不满足 $\delta > k$ 的 node7 上容器根据 δ' 从大到小排列，如表 3 所示。因为主要是网络导致的资源分布不均，尽可能调走网络资源而保留其他资源，所以设 a、b、c、d 为 0.15，0.15，0.55，0.15。这里将所有容器符 q 设置为 5 min，假设所有容器迁移代价符合 $\text{price}_i^{\text{m}} < 5$ min。

表 3　node7 上应用按负载最高资源从小到大排列

应用	W_{CPU} /%	W_{M} /%	W_{N} /%	$W_{\text{I/O}}$ /%	δ'
1	8	15	50	6.6	11.72
2	4	7	25	3.3	5.90

由表 3 确定迁移应用 1，迁移后 node7 的 $\delta = 8.87$ 满足 $\delta \leqslant 15$，将应用一加入到 2.1 中调度模型重新调度。

同理，筛选出 node2、node4 和 node3 上需要迁移的应用，加入到 2.1 中调度模型链表中，重新分配，最后运行状态如图 4 所示。

在实现迁移算法后，运行同样数目的容器应用，能减少运行应用的服务器节点，从而提高集群资源利用率，节约数据中心能耗。

优化后的调度策略更细粒度地划分了数据中心资源和增加了调整云平台的参数，在提高集群资源利用

率的条件下综合考虑了各种资源的负载均匀以及迁移时间代价，有效优化了基于 Openstack 的 Kubernetes 私有云调度模型。

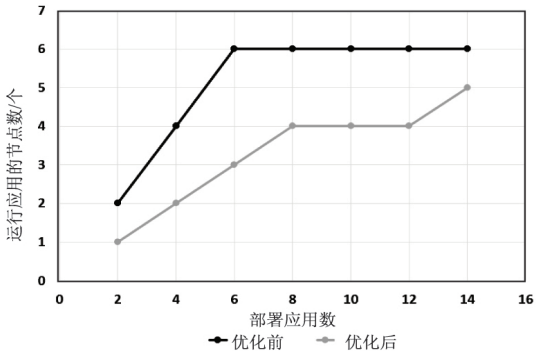

图 4 集群中运行应用的节点数对比

4 结束语

改进了 Kubernetes 原有的调度策略，结合 Openstack 云平台，提出一种基于 Openstack 的 Kubernetes 私有云弹性调度策略，细粒度划分了调度资源。通过对每种资源需求从多到少的排序轮转调度和运行一段时间后监测负载过高以及资源严重分配不均的服务器节点应用二次调整迁移算法，实现了对私有云的弹性调度，有效提高了云平台资源利用率，达到合理使用数据中心硬件资源和降低数据中心运营成本的效果。下一步将研究复杂的调度、混合云调度，通过更加细粒度地划分服务类型如有无持久存储需求，服务时长以及在可知和不可知应用最大资源上限的调度等，结合现有开源云架构如 Hadoop、Kubernetes、Openstack 和更高级算法解决实际问题。

参考文献:

[1] 刘 鹏.云计算 [M].第 3 版.北京: 电子工业出版社,2011.
[2] TURNBULL J.The Docker book: containerization is the new virtualization [M]. [s.l.]: [s.n.],2014: 4-7.
[3] CASALICCHIO E. Autonomic orchestration of containers: problem definition and research challenges [C] //EAI international conference on performance evaluation methodologies and tools. [s.l.]: [s.n.],2017.
[4] 陈 耿.开源容器云 openshift 构建基于 kubernetes 的企业应用云平台 [M].北京: 机械工业出版社,2017.
[5] NETTO H V,LUNG L C,CORREIA M,et al.State machine replication in containers managed by Kubernetes [J].Journal of Systems Architecture,2017,73: 53-59.
[6] 龚 正.Kubernetes 权威指南: 从 Docker 到 Kubernetes 实践全接触 [M].北京: 电子工业出版社,2015.
[7] KOMINOS C G,SEYVET N,VANDIKAS K.Bare-metal, virtual machines and containers in OpenStack [C] //20th conference on innovations in clouds, internet and networks. Paris,France: IEEE,2017: 36-43.
[8] FELTER W,FERREIRA A,RAJAMONY R,et al.An updated performance comparison of virtual machines and Linux containers [C] //IEEE international symposium on performance analysis of systems and software. Philadelphia, PA, USA: IEEE,2014: 438-453.
[9] CHEN C,REN S,HAU E.A Kubernetes-based monitoring platform for dynamic cloud resource provisioning [C] //IEEE global communications conference. [s.l.]: IEEE,2017: 1-6.
[10] 董西成.Hadoop 技术内幕: 深入解析 YARN 架构设计与实现原理 [M].北京: 机械工业出版社,2014.
[11] 唐 瑞.基于 Kubernetes 的容器云平台资源调度策略研究 [D].成都: 电子科技大学,2014.
[12] 杨鹏飞.基于 Kubernetes 的资源动态调度的研究与实现 [D].杭州: 浙江大学,2017.
[13] 彭丽苹,吕晓丹,蒋朝惠,等.基于 Docker 的云资源弹性调度策略 [J].计算机应用,2018,38(2): 557-562.
[14] 董西成.Hadoop 技术内幕: 深入解析 MapReduce 架构设计与实现原理 [M].北京: 机械工业出版社,2013.
[15] 董春涛,李文婷,沈晴霓,等.Hadoop YARN 大数据计算框架及其资源调度机制研究 [J].信息通信技术,2015,9(1): 77-84.