计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(4):187−196 [doi: 10.15888/j.cnki.csa.009034]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于改进秃鹰搜索算法的 Kubernetes 资源调度应用[①]

耿棒棒, 王　勇

(桂林电子科技大学 计算机与信息安全学院, 桂林 541004)
通信作者: 耿棒棒, E-mail: 942933128@qq.com

摘　要: Kubernetes 是一个管理容器化应用的开源平台, 其默认的调度算法在优选阶段仅把 CPU 和内存两种资源来作为计算节点的评分指标, 同时还忽略了不同类型的 Pod 对节点资源的占用比例是不同的, 容易导致某一资源达到性能瓶颈, 从而造成节点对资源使用失衡. 针对上述问题, 本文在 Kubernetes 原有的资源指标基础上增加了带宽和磁盘容量, 考虑到 CPU、内存、带宽和磁盘容量这 4 类资源在节点上的占用比例对节点的性能的影响, 可能造成 Pod 中应用的非正常运行, 甚至杀死 Pod, 从而影响集群整体的高可靠性. 本文将等待创建的 Pod 区分为可压缩消耗型、不可压缩消耗型以及均衡型, 并为每种类型的 Pod 设置相应的权重, 最后通过改进的秃鹰搜索算法 (TBESK) 来寻找出最优节点进行调度. 实验结果表明, 随着集群中 Pod 的数量在不断增加, 在集群负载较大的情况下, TBESK 算法的综合负载标准差和默认的调度算法相比提升了 24%.
关键词: Kubernetes 平台; 权重; 资源调度; 高可靠性; TBESK 算法; 容器云

引用格式: 耿棒棒,王勇.基于改进秃鹰搜索算法的 Kubernetes 资源调度应用.计算机系统应用,2023,32(4):187–196. http://www.c-s-a.org.cn/1003-3254/9034.html

Improved Bald Eagle Search Algorithm for Kubernetes Resource Scheduling Application

GENG Bang-Bang, WANG Yong

(School of Computer Science and Information Security, Guilin University of Electronic Technology, Guilin 541004, China)

Abstract: Kubernetes is an open-source platform for managing containerized applications. Nevertheless, its default scheduling algorithm only uses two resources, the central processing unit (CPU) and memory, as scoring metrics for computing nodes at the preference stage. Moreover, it also neglects the point that different types of Pods occupy different ratios of node resources. Consequently, a certain resource is highly likely to reach a performance bottleneck, ultimately causing an imbalance in the use of resources by nodes. To address the above problem, this study adds bandwidth and disk capacity to the original resource metrics of Kubernetes. The impact of the occupancy ratios of the four types of resources, i.e., CPU, memory, bandwidth, and disk capacity, at the nodes on the performance of the nodes may cause abnormal operation of the applications in Pods and even kill the Pods, ultimately affecting the overall high reliability of the cluster. For this reason, the study classifies the Pods waiting to be created into the compressible consumption type, the incompressible consumption type, and the balanced type and sets corresponding weights for each type. Finally, the optimal nodes for scheduling are obtained by an improved bald eagle search algorithm (TBESK). The experimental results show that as the number of Pods in the cluster increases, the standard deviation of the synthetic load of the TBESK algorithm is 24% lower than that of the default scheduling algorithm in the case of a large cluster load.
Key words: Kubernetes platform; weight; resource scheduling; high reliability; TBESK algorithm; container cloud

---

① 基金项目: 国家自然科学基金 (61662018, 61661015, 61831013); 广西创新驱动发展专项 (科技重大专项桂科 AA18118031)
收稿时间: 2022-08-24; 修改时间: 2022-09-27; 采用时间: 2022-10-21; csa 在线出版时间: 2023-02-10
CNKI 网络首发时间: 2023-02-13

© 中国科学院软件研究所 http://www.c-s-a.org.cn

1 概述

Kubernetes 是 Google 公司开源的一个容器编排与调度管理框架, 该项目最初是 Google 内部面向容器的集群管理系统, 现在由云原生计算基金会托管开源平台, 由 Google、AWS 等主要参与者支持, 其目标是通过创建一组新的通用容器技术来推进云原生技术和服务的开发[1]. 在 Kubernetes 平台中调度是进行容器编排工作重要的一环, 其中 Scheduler 又是这一环中的核心组件, 其功能是根据调度策略将待调度的 Pod 匹配到最合适的节点上[2]. Pod 在 Kubernetes 集群中进行调度时需要经历 Predicates 和 Priorities 两个阶段: 在预选阶段, Kubernetes 会遍历所有的计算节点, 排除那些完全不能符合对应 Pod 的基本运行节点; 而在优选阶段由 Scheduler 组件按照相应的优选算法对通过第一阶段筛选合格的节点进行打分, 最后决定出分数最高的节点进行 Pod 调度.

Kubernetes 中 Scheduler 模块在整个平台中有着举足轻重的地位, 不合适的预选和优选策略会对集群的资源利用率产生不利的影响, 会对用户部署请求缺乏快速响应, 服务质量下降, 增加服务商开销成本.

现阶段, 国内外学者针对 Kubernetes 平台中 Pod 调度做了大量的研究工作, 并且已经产生较多的研究成果. 胡程鹏等[3] 在原有的资源指标上增加了带宽和磁盘 IO, 并将改进的遗传算法运用在 Kubernetes 调度策略中, 该算法优化了集群的负载均衡, 但是没有考虑到不同类型的应用对 Pod 中资源的消耗是不同的, 容易导致节点单个资源的瓶颈; Zhang 等[4] 将蚁群算法和粒子群优化算法相结合, 提出基于启发式算法的 Kubernetes 调度算法, 实验表明该算法相较于原有调度算法, 降低了节点总成本和最大负载, 使负载更加均衡, 但是只考虑了 CPU 和内存两种资源, 忽略了其他资源对节点的影响, 易导致其他类型的资源过载; 孔德瑾等[5] 面向 5G 边缘计算的资源调度场景, 提出一种基于 CPU、内存、带宽和磁盘 4 种指标的自适应权重调度策略 WSLB, 避免了节点中带宽和磁盘资源的过载, 提高了资源利用率; Li 等[6] 提出了 BDI (balanced-disk-IO-priority) 和 BCDI (balanced-CPU-disk-IO-priority) 动态算法, 两种算法分别改善了节点之间磁盘 I/O 平衡和解决单个节点的资源不平衡问题, 但是没有考虑到 Pod 中的应用在不同时刻对节点资源消耗是不同的, 可能导致节点资源耗尽; 常旭征等[7] 针对 Kubernetes 默认调度算法没有考虑到节点本身的资源利用率, 且没有考虑到网络和 IO 指标, 提出了相应的改进算法, 实验证明改进的算法可以提高进群的负载; Dua 等[8] 实现可以将任务替换的算法, 通过对每个作业打上标签, 将其分到不同特定类型任务集群; 何龙等[9] 实现了基于应用历史记录的 Kubernetes 调度方法, 该方法在提升节点资源均衡程度和减少资源浪费相比原有算法均有一定提升, 但是该算法仅考虑了 CPU 和内存指标, 对于磁盘容量以及网络带宽等指标等方面研究不足; 张可颖等[10] 通过将 OpenStack 和 Kubernetes 相结合提出一种基于容器的弹性调度策略, 实验表明在数据中心能耗和资源利用率方面得到了提高.

虽然国内外学者针对 Kubernetes 资源调度取得了一定的成果, 但是忽略了不同时刻节点中的应用对资源的消耗是不同的, 如果节点中的应用处于低消耗状态, 此时节点满足 Pod 调度条件并调入, 当节点中的应用处于高消耗的状态, 可能导致节点出现资源耗尽杀死 Pod, 同时也没有考虑到应用的资源类型对 Pod 的影响是各异的[11]. 针对上述问题, 本文在 Kubernetes 原有指标基础上, 增加带宽和磁盘容量两种指标, 根据 Pod 中应用的资源需求将待调度的 Pod 分类, 并为不同种类的 Pod 设置相应的权重, 并且选择节点某个时刻的最大的资源利用率作为计算指标, 最后利用改进的秃鹰算法寻找出最优节点进行调度.

2 默认调度算法分析

Kubernetes Scheduler 是 Kubernetes 关键模块, 其职责通过 Kubernetes watch 机制从集群中获取待调度 Pod 的相关信息, 通过 Pod 和节点的相关信息计算出集群中最合适的工作节点信息进行捆绑, 并将相关的绑定内容写入 ETCD 中[12]. 接着 Kubelet 会接收到 Pod 的调度消息, 从 ETCD 中获取 Pod 配置文件并完成相关资源和容器的创建.

Kubernetes 中 Pod 调度主要经过两个流程, Predicates 和 Priorities. Predicates 算法通过遍历集群中全部节点, 将满足要求的节点记录并作为 Priorities 的输入, 如果全部节点都不满足, 则 Pod 一直处于悬挂阶段. 而优选阶段是对预选阶段的筛选出的节点按照优选策略进行节点打分, 通过分数进行优先级排序, 选出优先级最高的节点进行 Pod 部署. 在 Kubernetes 的优选阶段中, 主要提供 BalancedResourcesAllocation、

© 中国科学院软件研究所 http://www.c-s-a.org.cn

LeastRequestedPriority 等策略对节点进行优先级排序.

1) 默认的 LeastRequestedPriority 算法, 该算法从 Predicates 筛选出来的节点中, 选择 CPU 和内存剩余最多的节点及资源消耗最少的节点进行 Pod 调度, 如式 (1) 所示:

$$score = \left[\frac{S_{\mathrm{CPU}} - N_{\mathrm{CPU}}}{S_{\mathrm{CPU}}} \times 10 + \frac{S_{\mathrm{mem}} - N_{\mathrm{mem}}}{S_{\mathrm{mem}}} \times 10 \right] \Big/ 2 \quad (1)$$

2) 默认的 BalancedResourceAllocation 算法, 该算法从 Predicates 筛选出来的节点中, 选择 CPU 和内存利用率相差最小的节点进行 Pod 调度, 如式 (2) 所示:

$$score = 10 - abs\left(\frac{N_{\mathrm{CPU}}}{S_{\mathrm{CPU}}} - \frac{N_{\mathrm{mem}}}{S_{\mathrm{mem}}} \right) \times 10 \quad (2)$$

其中, S_{CPU}和S_{mem}分别代表节点所拥有的 CPU 和内存的总资源, N_{CPU}和N_{mem}则代表节点已用的资源 (Pod 需要的资源和节点已被使用的资源之和). 通过分析 Kubernetes 调度器的默认算法可以发现其默认算法对节点的评价指标中只考虑了 CPU 和内存的使用率, 没有考虑磁盘容量和网络带宽的使用. 并且 Kubernetes 调度器是一种静态调度策略, 该调度机制虽然简单, 但是缺乏灵活性, 且只能在 Pod 初次部署时进行资源配置. 还忽略了应用对资源的消耗是不均匀的, 容易导致节点对不同资源的使用不均衡, 从而造成节点对某一资源的性能瓶颈. 同时默认调度策略调度的优先级是根据应用的请求量来衡量的, 不能很好地反映节点的实际资源情况, 容易使节点的资源失衡.

3 秃鹰搜索算法改进与实验验证

在 Pod 调度过程中, 若将 m 个 Pod 分配到 n 个节点中, 根据排列组合可得出共有 n^m 种情况, 这是一个 NP Hard 问题, 在解决这类问题上, 如果问题的固有知识不能被用来减少搜索空间, 很容易产生搜索的组合爆炸.

针对 NP Hard 问题, 常用启发式算法来解决, 秃鹰搜索算法 (bald eagle search, BES)[13] 是 2020 年马来西亚学者 Alsattar 等提出的一种模仿秃鹰捕食过程的新型元启发式算法. 该算法相较于传统的启发式算法具有较强的全局搜索能力, 在应对各类复杂数值优化问题都能够有效的解决. 该算法可分为 3 个阶段, 即选择搜索猎物的区域、在选择的搜索区域内搜索猎物和俯冲捕猎.

3.1 秃鹰搜索算法介绍

3.1.1 选择搜索空间

在选择阶段, 秃鹰寻找并挑选所选搜索区域内它们可以捕食最理想的空间 (就食物数量而言), 式 (3) 在数学上描述了这种行为.

$$P_{i,\mathrm{new}} = P_{\mathrm{best}} + \alpha \times \lambda (P_{\mathrm{mean}} - P_i) \quad (3)$$

其中, α是用于控制位置更改的参数, 该参数的值介于 1.5 到 2 之间; λ是一个随机数, 取值在 0 到 1 之间; P_{best}表示秃鹰表示当前选择的搜索空间, 该搜索空间是由秃鹰根据它们之前搜索时确定的最佳位置选择的; P_{mean}表示这些秃鹰已经用完了之前各点的所有信息; P_i为第i只秃鹰所处位置.

3.1.2 搜索空间猎物 (探索)

在搜索阶段, 秃鹰在挑选好的搜寻区域内搜索猎物, 并在螺旋形区域内不断向不同地方移动, 以加快搜索速度. 俯冲的最佳位置如式 (10) 所示:

$$\theta(i) = \alpha \times \pi \times rand \quad (4)$$

$$r(i) = \theta(i) + R \times rand \quad (5)$$

$$xr(i) = r(i) \times \sin(\theta(i)) \quad (6)$$

$$yr(i) = r(i) \times \cos(\theta(i)) \quad (7)$$

$$x(i) = xr(i) / \max(|xr|) \quad (8)$$

$$y(i) = yr(i) / \max(|yr|) \quad (9)$$

$$P_{i,\mathrm{new}} = P_i + x(i) \times (P_i - P_{\mathrm{mean}}) + y(i) \times (P_i - P_{i+1}) \quad (10)$$

其中, $\theta(i)$与$r(i)$分别表示螺旋方程的极角与极径; α与R是控制螺旋轨迹的参数, 取值范围分别为 (5, 10)、(0.5, 2); $x(i)$与$y(i)$表示在极坐标下秃鹰的位置, 取值均为 (−1, 1); P_{i+1}表示第i只秃鹰下一次更新的位置.

3.1.3 俯冲捕获猎物 (利用)

在俯冲阶段, 秃鹰从最佳位置向目标猎物所摇摆, 其他秃鹰也都朝着最优点移动. 利用极坐标方式来描述运动状态, 如第 2.1.2 节式 (4)–式 (9) 所示.

秃鹰的位置更新方式如式 (11) 和式 (12) 所示:

$$\begin{cases} \delta_x = x(i) \times (P_i - C_1 \times P_{\mathrm{mean}}) \\ \delta_y = y(i) \times (P_i - C_2 \times P_{\mathrm{mean}}) \end{cases} \quad (11)$$

$$P_{i,\mathrm{new}} = rand \times P_{\mathrm{best}} + \delta_x + \delta_y \quad (12)$$

3.2 秃鹰搜索算法的改进

与其他智能优化算法一样, 秃鹰搜索算法也存在容易陷入局部最优和收敛精度低的问题[14]. 因此本文

© 中国科学院软件研究所 http://www.c-s-a.org.cn

提出了一种基于混沌映射和 t 分布改进的秃鹰搜索算法 (TBESK). 首先, 为了增强种群的多样性, 本文在初始化种群时, 使用 Tent 混沌映射来提高秃鹰的全局搜索能力; 其次在搜寻阶段, 算法陷入局部最优, 使用 t 分布扰动跳出局部最优, 增强整体寻优能力.

3.2.1　Tent 混沌映射

秃鹰搜索算法 (BES) 在进行种群初始化时, 通常将随机产生的数据用作初始种群信息, 遗失了种群多样性, 使得算法寻优能力降低. 由于混沌变量包含随机性、遍历性和规律性等特性[15], 很多学者将其应用于初始化种群, 增强了种群的多样性, 改善算法的全局搜索能力. 目前常用于群体智能领域的混沌序列有 Tent 映射、Logistic 映射、Singer 映射等, 但是, 不同的混沌序列有着不同的特点, 对算法的优化效果也不一样. 单梁等[16] 研究验证了 Tent 映射在均匀分布和收敛速度方面均优于 Logistic 映射, 通过严格的数学推理, 在 [0, 1] 区间内 Tent 映射可以产生优化算法的混沌序列.

本文使用 Tent 映射来初始化种群, Tent 映射的表达式如下所示:

$$x_{i+1}=\begin{cases}2x_i,\ 0\leqslant x\leqslant \dfrac{1}{2}\\ 2(1-x_i),\ \dfrac{1}{2}<x\leqslant 1\end{cases} \tag{13}$$

即经过贝努利移位变换后表达式如下:

$$x_{i+1}=(2x_i)\ \mathrm{mod}\ 1 \tag{14}$$

利用 Tent 混沌映射对种群进行初始化的步骤如下.

Step 1. 在 (0, 1) 之间生成随机数 x_0, 记 $j=0$.

Step 2. 根据式 (13) 计算得到新的 x 序列, $j=j+1$.

Step 3. 当 j 达到设置最大迭代次数, 则停止迭代, 保存好 x 序列.

3.2.2　自适应 t 分布

学生 t 分布简称 t 分布, 是以 0 为中心, 左右对称的单峰分布. t 分布是一簇曲线, 其形态变化与自由度 n 有关, n 的值越小, 其曲线越低平; n 值越大, 其曲线越接近标准正态分布曲线, $t(n\to\infty)\to N(0,1)$, $t(n=1)=c(0,1)$, 其中 $N(0, 1)$ 为高斯分布, $c(0, 1)$ 为柯西分布, 由此可以看出 t 分布的两个边界特例分布为标准高斯分布和柯西分布[17]. 对秃鹰选择搜索空间的位置 $P^t_{i,\mathrm{new}}$ 定义见式 (15):

$$P^t_{i,\mathrm{new}}=P_{i,\mathrm{new}}+P_{i,\mathrm{new}}\times t(n) \tag{15}$$

其中, $P^t_{i,\mathrm{new}}$ 为变异后的搜索空间, $P_{i,\mathrm{new}}$ 为第 i 个秃鹰的搜索空间, $t(n)$ 表示自由度为 n 的 t 分布, 其中 n 为迭代次数. 该定义式在秃鹰进行搜索空间选择时增加了 t 分布扰动 $P_{i,\mathrm{new}}\times t(n)$, 使搜索空间陷入局部最优时, 能够跳脱出来, 提升了搜索精度, 同时加快了收敛速度.

自适应 t 分布变异使用算法的迭代次数作为 t 分布的自由度参数, 在算法运行初期, 迭代次数的值较小, t 分布变异近似于柯西分布变异, 算法具有良好的全局搜索能力; 在算法运行后期, t 分布近似于高斯分布变异, 算法具有良好的局部开发能力; 在算法运行中期, t 分布变异介于柯西分布变异和高斯分布变异之间. t 分布的变异算子结合高斯算子和柯西算子的优势, 同时提高了算法的全局探索性和局部开发性.

3.2.3　TBESK 算法描述

基于 Tent 混沌映射和 t 分布对 BES 算法的改进, TBESK 算法具体实现流程如下所示.

Step 1. 设置种群的规模 N, 迭代次数 $t_{\max}$, 搜索空间维度以及初始边界条件.

Step 2. 使用第 3.2.1 节中 Tent 混沌映射对种群进行初始化, 根据适应度函数求出秃鹰种群个体的适应度值, 对秃鹰种群进行排序, 找出全局最优的个体.

Step 3. 在秃鹰选择搜索阶段, 使用 t 分布对式 (15) 进行位置更新.

Step 4. 在秃鹰搜索空间猎物阶段, 按照式 (10) 进行位置更新.

Step 5. 在俯冲阶段, 按照式 (12) 进行秃鹰的位置更新.

Step 6. 不断更新种群最优位置以及适应度值, 更新迭代次数. 判断算法是否达到设置的迭代次数或者精度要求, 若满足, 则终止算法, 输出适应度最优位置及最优解; 若没有, 则返回 Step 3 进行循环.

3.3　实验验证

本文选取秃鹰搜索算法作为 Kubernetes 资源调度算法, 主要是因其具有较高的收敛精度. 而收敛精度的优劣则可以通过一些基本的测试函数来测试对比. 并且本文主要工作在于将改进的秃鹰算法用于 Kubernetes 资源调度, 因此后文主要将原秃鹰搜索算法与 TBESK 算法在 Kubernetes 环境中对比验证. 本文选取了常见的 4 个测试函数进行实验对比, 详细的测试函数信息见表 1.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

表 1 常见的 4 个测试函数

函数名	表达式
Quartic	$f_1(x)=\sum_{i=1}^{n} ix_i^4+random(0,1)$
Griewank	$f_2(x)=\frac{1}{4000}\sum_{i=1}^{n} x_i^2-\prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right)+1$
Ackley	$f_3(x)=-20\exp\left(\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)-\exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2x_i)\right)+20+\mathrm{e}$
Sphere	$f_4(x)=\sum_{i=1}^{n} x_i^2$

为了验证本文改进的秃鹰搜索算法的收敛性和稳定性, 选取了秃鹰搜索算法 (BES)、使用 t 分布改进后的麻雀算法 (tSSA)[18]、使用 t 分布改进的缎蓝园丁鸟算法 (tSBO)[19] 和 t 分布改进的粒子群算法 (tPSO)[20] 进行对比实验. 为了降低随机性, 本文选取 5 种算法分别在 4 个测试函数上独立运行 10 次. 表 2 分别列出了 TBESK、BES、tSSA、tSBO、tPSO 独立运行 10 次所得的最优值、最差值以及平均迭代次数.

表 2 不同的算法对比

函数	维度	算法	最优	最差	平均迭代次数
Quartic	10	TBESK	1.481178E–06	7.8456054E–05	162
	10	BES	1.7194089E–05	1.2249E–04	208
	10	tSSA	3.223E–04	2.09131E–03	264
	10	tSBO	1.2509E–04	4.0208E–04	280
	10	tPSO	1.7314E–04	1.123E–03	320
Griwank	10	TBESK	0	0	3
	10	BES	0	0	5
	10	tSSA	0	0	173
	10	tSBO	0	0	63
	10	tPSO	9.6501E–12	3.7925E–09	660
Ackley	10	TBESK	4.4408921E–16	4.4408921E–16	8
	10	BES	4.4408921E–16	4.4408921E–16	10
	10	tSSA	4.4408921E–16	4.4408921E–16	338
	10	tBSO	4.4408921E–16	4.4408921E–16	170
	10	tPSO	2.2160E–07	0.0001	608
Sphere	10	TBESK	0	0	88
	10	BES	0	0	102
	10	tSSA	2.8928E–83	1.3272E–12	561
	10	tBSO	5.8689E–292	1.0367E–181	243
	10	tPSO	9.4051E–15	1.4024E–07	453

根据实验结果可知, 在所有的测试函数中, 本文提出的融合 Tent 混沌映射和 t 分布的秃鹰搜索算法全部优于 BES、tSSA、tSBO、tPSO.

4 TBESK 调度策略

4.1 划分 Pod 资源类型

在集群中, Pod 中的应用对资源的不同需求影响了节点的性能, 比如有的租户对 CPU 需求比较大, 会加大节点的 CPU 负载, 有的租户则对内存需求比较大, 会加大内存的负载, 因此需要对 Pod 类型进行划分. 由于 CPU 和带宽超出配额后可以通过软性限制避免超额使用, 而对于内存和磁盘容量超出配额后只能通过内核抛出 out-of-memory 错误地将 Pod 杀死, 并将 Pod 重新调度[21], 本文按照 Pod 中应用对 CPU、内存、带宽、磁盘容量资源的请求, 将 Pod 分为可压缩资源消耗型、不可压缩资源消耗型和均衡型.

在 Kubernetes 集群中, 租户把对 CPU、内存、带宽、磁盘容量的资源需求写在 Pod 的 yaml 文件中. 本文根据 yaml 文件中资源配置, 计算出 Pod 部署后对相应节点上的资源消耗程度 C_r, 定义 R_r 为 yaml 文件中应用对资源 r 的需求量, T_r 为某一节点上资源 r 的总量, 该 Pod 对节点资源的消耗大小如式 (16) 所示:

$$C_r = R_r / T_r \tag{16}$$

其中, 资源 r 代表着节点上 CPU、内存、带宽和磁盘容量. 根据式 (16) 可计算出 $C_{\rm CPU}$、$C_{\rm mem}$、$C_{\rm net}$ 和 $C_{\rm disk}$. 如果 $C_{\rm CPU}/C_{\rm mem}>1$ 和 $C_{\rm net}/C_{\rm disk}>1$, 规定该 Pod 资源相对于该节点为可压缩资源消耗型; 如果 $C_{\rm CPU}/C_{\rm mem}<1$ 和 $C_{\rm net}/C_{\rm disk}<1$, 规定该 Pod 资源相对于该节点为不可

© 中国科学院软件研究所 http://www.c-s-a.org.cn

压缩资源消耗型; 如果$C_{CPU}/C_{mem}<1$、$C_{net}/C_{disk}>1$或$C_{CPU}/C_{mem}>1$、$C_{net}/C_{disk}<1$, 规定该 Pod 资源相对于该节点属于均衡型.

4.2 适应度函数的选择

启发式算法最重要的是适应度函数选择, 本文选取节点的负载均衡度和节点的可用资源剩余率作为打分标准.

1) 可用资源剩余率

当 Pod 在集群中任意节点上部署时, 对应节点会获取 Pod 中应用的需求, 为 Pod 分配相应的资源. 随着 Pod 部署的逐渐增多, 节点当前的可用资源在逐渐减少. 定义 Pod 的各个资源需求量与节点上可用资源的比为可用资源剩余率, 公式如下所示:

$$R_{CPU}=1-\frac{P_{CPU}}{N_{CPU}-S_{CPU}} \tag{17}$$

$$R_{mem}=1-\frac{P_{mem}}{N_{mem}-S_{mem}} \tag{18}$$

$$R_{net}=1-\frac{P_{net}}{N_{net}-S_{net}} \tag{19}$$

$$R_{disk}=1-\frac{P_{disk}}{N_{disk}-S_{disk}} \tag{20}$$

$$R_N=\mu_1 R_{CPU}+\mu_2 R_{mem}+\mu_3 R_{net}+\mu_4 R_{disk} \tag{21}$$

其中, P_{CPU}、P_{mem}、P_{net}和P_{disk}分别表示 Pod 中应用对 CPU、内存、带宽和磁盘容量的需求; S_{CPU}、S_{mem}、S_{net}和S_{disk}表示节点上已经使用的 CPU、内存、带宽和磁盘容量的资源量; N_{CPU}、N_{mem}、N_{net}、N_{disk}则表示节点所拥有的总资源量; R_{CPU}、R_{mem}、R_{net}和R_{disk}反映了 Pod 所需资源部署在节点上的可用 CPU 剩余率可用内存剩余率、可用带宽剩余率和可用磁盘容量剩余率. R_N为调度器将 Pod 部署在某个节点后, 该节点的可用资源剩余率, 其值越大, 该节点分数越高; μ_1、μ_2、μ_3和μ_4分别表示R_{CPU}、R_{mem}、R_{net}和R_{disk}的权重, $\mu_1+\mu_2+\mu_3+\mu_4=1$. 权重根据 Pod 的资源类型划分相应的大小.

2) 负载均衡度

当集群中的应用处于低峰期时, 此时应用对节点的资源使用比较低, 此时去选择节点部署 Pod, 容易在应用处于高峰期时, 对节点资源的使用增大导致节点的资源过载, 影响集群的负载均衡. 本文选择当集群处于高峰期时的节点资源指标, 用来衡量集群负载大小. 为了预防节点因资源过载而影响节点中的 Pod 正常运行, 本文为节点上的资源设置最大负载率, 即由 (最大负载率–(Pod 资源量+高峰期已用资源量)/总资源量) 确定, 计算公式如下:

$$L_{CPU}=\frac{P_{CPU}+S_{CPU}}{N_{CPU}} \tag{22}$$

$$L_{mem}=\frac{P_{mem}+S_{mem}}{N_{mem}} \tag{23}$$

$$L_{net}=\frac{P_{net}+S_{net}}{N_{net}} \tag{24}$$

$$L_{disk}=\frac{P_{disk}+S_{disk}}{N_{disk}} \tag{25}$$

$$\begin{aligned}L_N=&v_1(T_{CPU}-L_{CPU})+v_2(T_{mem}-L_{mem})\\&+v_3(T_{net}-L_{net})+v_4(T_{disk}-L_{disk})\end{aligned} \tag{26}$$

其中, L_{CPU}、L_{mem}、L_{net}和L_{disk}分别代表 Pod 部署在节点上, 该节点的 CPU、内存、带宽和磁盘容量负载率; T_{CPU}、T_{mem}、T_{net}和T_{disk}表示资源的临界负载, 其取值为 (0, 1); 式 (26) 中, L_N代表着节点的负载均衡度; v_1、v_2、v_3和v_4为自定义均衡度权重, 其权重大小是根据资源对节点的均衡度的影响程度设置, 其中$v_1+v_2+v_3+v_4=1$.

节点 N 的优选总得分计算公式如下:

$$S_N=(R_N+L_N) \tag{27}$$

其中, S_N表示节点的总得分, 当节点的剩余资源越多R_N值越大, 资源综合负载均衡度越高L_N值越大, 从而S_N值越大, 节点部署 Pod 的优先越高.

4.3 TBESK 调度模型

假设 Kubernetes 集群由 m 个配置不尽相同的节点服务器, 有 n 个待调度的 Pod 需要调度至 m 个节点上. 初始种群有 x 个秃鹰, 秃鹰在 m 维空间搜索最优解, 最优解即为 Pod 寻找最优的部署节点. 第 k 个秃鹰的可行解表示为$Pos_k=(Pos_{k1},Pos_{k2},\cdots,Pos_{ki},\cdots,Pos_{kn})$, $k\in[1,x]$, Pos_{ki}表示第 i 个 Pod 被调度部署的所在节点, 其取值区间为 $[1, m]$; TBESK 策略先根据 Pod 中资源的需求将 Pod 分类, 然后则通过 3 个筛选阶段来确定 Pod 最终分配节点.

TBESK 在选择阶段, 从 m 个节点中选择最佳搜索区域进行 Pod 调度, 定义的公式为:

$$P_{ki,new}=P_{best}+\alpha\times\lambda(P_{mean}-Pos_{ki}) \tag{28}$$

$$P^t_{ki,new}=P_{ki,new}+P_{ki,new}\times t(n) \tag{29}$$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

其中, $P_{ki,\text{new}}$为第 k 个秃鹰中第 i 个待调度 Pod 所在的新节点, P_{best}为通过自定义的适应度函数计算出上次迭代 Pod 部署在节点上的最优位置, α是用于控制位置更改的参数, 该参数的值介于 1.5 到 2 之间; λ是一个随机数, 取值在 0 到 1 之间; P_{mean}表示这些秃鹰已经用完了之前各点的所有信息; $P^{t}_{ki,\text{new}}$为经过 t 分布变异后待部署 Pod 所在新的节点.

TBESK 策略在搜索阶段, 秃鹰在选择好的节点区域内选择合适的节点部署 Pod, 并在螺旋形区域内不断向不同地方移动, 定义的公式如下:

$$P_{ki,\text{new}} = P_{ki} + x(i)\times(P_{ki} - P_{\text{mean}}) + y(i)\times(P_{ki} - P_{ki+1}) \tag{30}$$

其中, $x(i)$与$y(i)$表示在极坐标下秃鹰的位置, 取值均为(−1, 1); P_{ki+1}表示第 k 只秃鹰下一次 Pod 更新的位置.

TBESK 策略在俯冲阶段, 秃鹰从搜索空间的最佳位置摇摆到它们的目标猎物. 所有的秃鹰也都朝着 Pod 被调度到最合适的节点移动, 定义公式如下:

$$P_{ki,\text{new}} = rand\times P_{\text{best}} + \delta_x + \delta_y \tag{31}$$

通过 3 个阶段筛选迭代, TBESK 算法选择出最合适 Pod 分配方案, 调度器将 Pod 调度到相应的节点上. TBESK 调度模型图如图 1 所示.

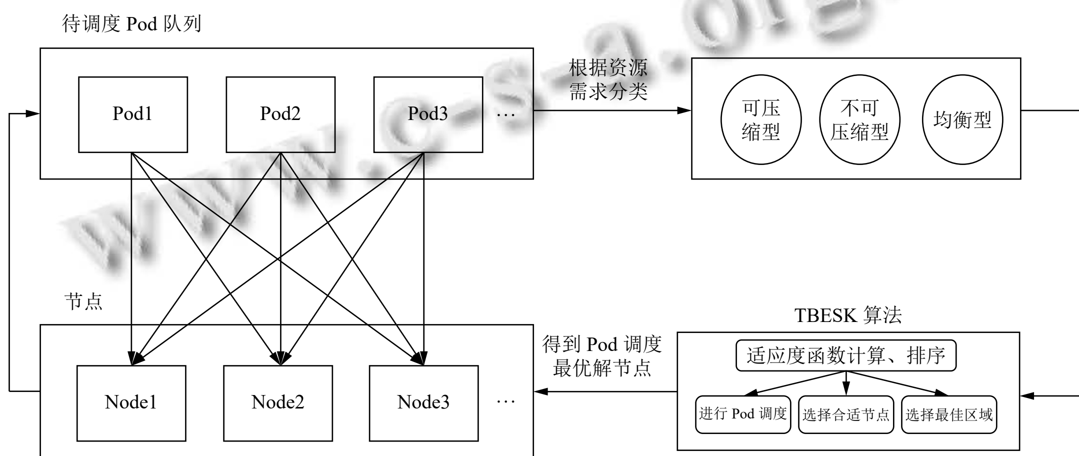

图 1 TBESK 调度模型图

5 实验验证与分析

为测试本文提出的 TBESK 算法在 Kubernetes 集群的性能, 进行实验验证. 所有实验均由 PyCharm 2020.2 编程实现, 并基于平台: Windows 10, AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx 2.10 GHz, 16 GB 内存.

5.1 实验环境

本文数据集采用文献 [22] 中的数据集. 在仿真环境下, 模拟一个包含 31 个节点的 Kubernetes 集群, 节点具体信息如表 3.

同时, 考虑到节点对不同资源的敏感程度, 将应用需求分为 3 类: 可压缩消耗型、不可压缩消耗型以及均衡型. 表 4 显示了部分应用程序请求信息.

表 3 节点资源信息

Pod (id)	CPU (core)	Memory (MB)	Bandwidth (Gb/s)	Disk (GB)
1	16	24574	30	800
2	32	32768	30	1000
3	64	49512	40	1200

表 4 部分 Pod 资源需求表

Pod (id)	CPU (core)	Memory (MB)	Bandwidth (Gb/s)	Disk (GB)
1	0.58	212	0.68	1.47
2	0.17	247	0.38	1.93
3	0.07	225	0.05	1.71
4	0.5	383	0.7	14.87

5.2 实验评价指标

假定 Kubernetes 集群是由 n 台服务器搭建, 其中每台服务器有 m 种资源. 定义服务器上单个资源的资源利用率为$A(i, j)$, 其中i表示不同的服务器, j 表示服务器上的资源; 定义服务器上所有资源利用率总和的平均值为A_{avg}; $Z(i)$代表着服务器i资源失衡度, 即服务器i上的资源标准差之和; Z_{avg}为集群的平均资源失衡度, 其值越小, 表明集群中资源的分布越均匀, 服务器上的资源不易出现倾斜, 可以为集群提供更多资源去部署 Pod:

$$A_{\text{avg}} = \sum_{j=1}^{m}\frac{A(i,j)}{m} \tag{32}$$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

$$Z(i) = \sqrt{\sum_{j=1}^{m}\left(A(i,j) - A_{\rm avg}(i)\right)^2} \tag{33}$$

$$Z_{\rm avg} = \sum_{i=1}^{n}\frac{Z(i)}{n} \tag{34}$$

5.3 实验结果和分析

本文使用 Kubernetes 平台自带的两种调度的策略: LeastRequestedPriority (LRA) 算法、BalancedResourceAllocation (BRA) 算法和本文中提出的改进的秃鹰搜索算法 (TBESK), 从资源失衡度、CPU、内存、带宽、磁盘容量利用率角度来对比 3 种算法的表现; 本文还使用了组合权重 TOPSIS 调度算法 (CWT)[22] 以及原始的 BES 算法与 TBESK 算法在集群资源失衡度上对比 3 种算法的表现.

5.3.1 集群资源失衡度

图 2 为集群资源失衡度曲线, 在 Pod 应用部署初期阶段, 可以发现 TBESK 集群失衡度在下降, 这是由于 TBESK 算法在对 Pod 进行调度时, 相对于自带的调度算法, 不仅考虑了带宽和磁盘容量两种资源指标、还考虑了 Pod 资源类型并设置相应的权重, 使节点资源更加均衡, 降低了集群失衡度. 整体上来看, TBESK 算法的资源失衡度与 LRA 算法、BRA 算法相比, 分别下降 24% 和 21%, 这表示 TBESK 算法可以有效地调节集群负载, 有效降低了节点出现资源使用出现失衡的情况.

图 3 为 CWT 算法、BES 算法以及 TBESK 算法的资源失衡度曲线, 可以发现在 Pod 数目较少的情况下, 资源失衡度相差不大, 但是随着 Pod 数目的增多, TBESK 相较于其他两种算法在资源失衡度上有着明显的优势

5.3.2 集群资源失衡度

图 4 和图 5 体现了当 Kubernetes 集群中的 Pod 数量为 1 800 时, 各个节点分别在 TBESK 算法、LRA 算法和 BRA 算法作用下, CPU 和内存资源利用率. 3 种算法都考虑到 CPU 和内存, 因此在 CPU 和内存的资源利用率上没有很大的差异.

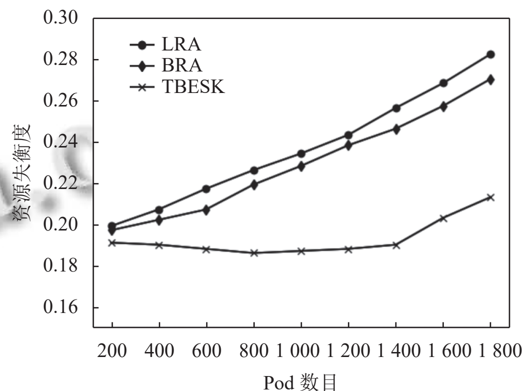

图 2 集群失衡度 I

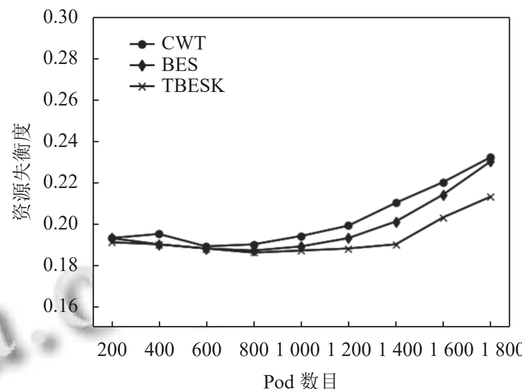

图 3 集群失衡度 II

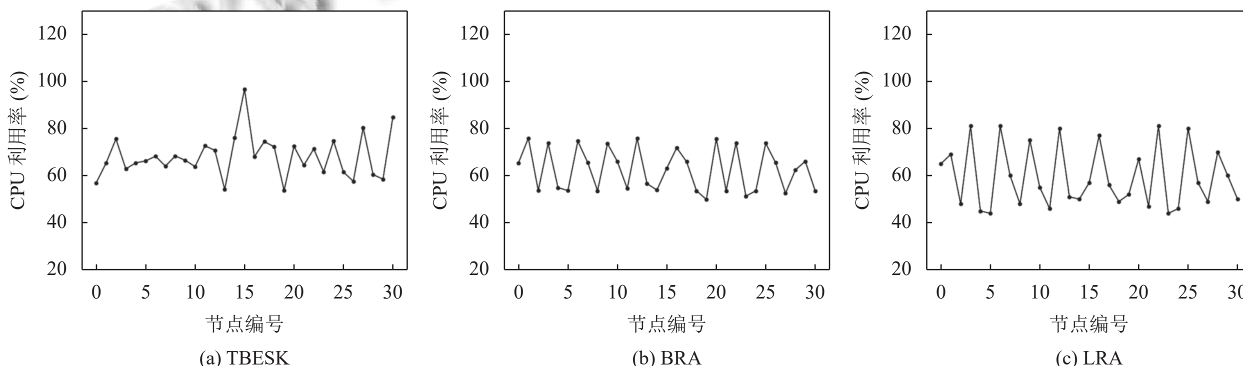

图 4 不同策略下的 CPU 利用率

© 中国科学院软件研究所 http://www.c-s-a.org.cn

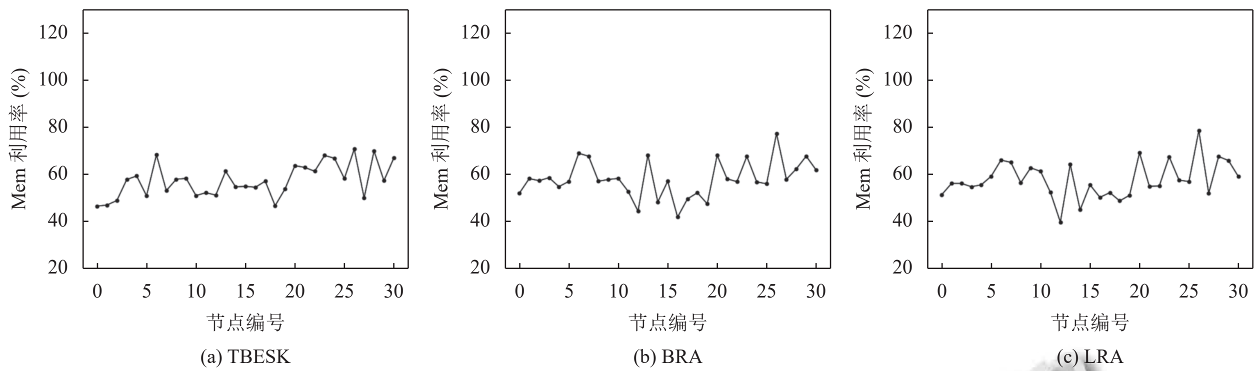

图 5　不同策略下的内存利用率

5.3.3　带宽、磁盘容量资源利用率

图 6 和图 7 展示了在 Pod 数量为 1 800 时, 集群分别在 TBESK 算法、LRA 算法和 BRA 算法作用下节点的带宽利用率和磁盘容量的资源利用率. 在 LRA 算法和 BRA 算法下集群中部分节点网络带宽和磁盘容量出现倾斜, TBESK 算法相对于其他两种算法, 集群中节点的带宽利用率和磁盘容量利用率的波动相对较小.

在 LRA 算法下, 集群中不同节点的带宽利用率最大相差 54%, 不同节点的磁盘容量利用率最大相差 65%, 而在 BRA 算法下, 集群中不同节点的带宽利用率最大相差 60%, 磁盘容量利用率相差 80%. 同时可以明显看到集群中部分节点的带宽和磁盘容量的资源利用率已超过 100%, 如果继续将相同需求 Pod 调度这些节点上, 可能导致应用出现一些问题.

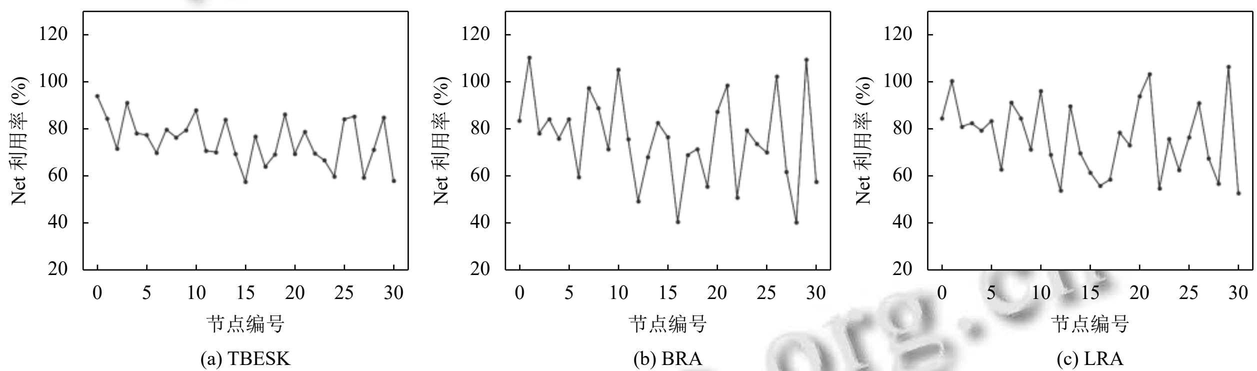

图 6　不同策略下的带宽利用率

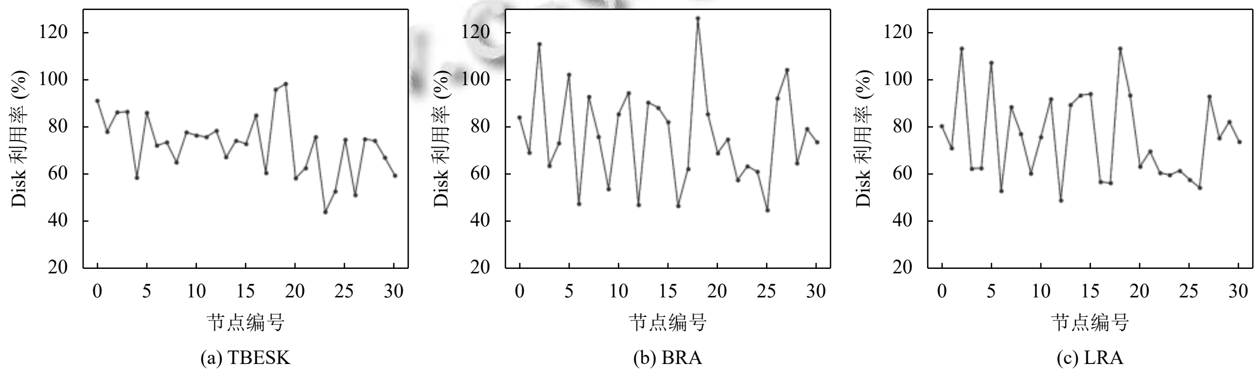

图 7　不同策略下的磁盘容量利用率

在 TBESK 算法下, 集群中各个节点的带宽资源利用率都在 55% 到 95% 的区间内波动, 节点中带宽没有发生超分的情况; 磁盘容量利用率都处在 43% 到 98% 的区间内波动, 没有出现磁盘容量超过 100% 情况, 和 LRA 算法、BRA 算法对比, 集群带宽和磁盘容量在 TBESK 算法下更加稳定.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

6 结论与展望

在 Kubernetes 集群中, 默认调度算法忽略了应用对不同的资源消耗是不同的, 易导致某一资源到达瓶颈, 同时还忽略了带宽和磁盘容量对节点的影响. 针对上述问题, 本文根据 Pod 的资源需求, 将 Pod 分为可压缩消耗型、不可压缩消耗型以及均衡型, 为每种类型 Pod 的资源自定义相应的权重, 并在 CPU、内存资源指标的基础上考虑了带宽、磁盘容量对节点的影响. 考虑秃鹰搜索算法易陷入局部最优和收敛精度低问题, 本文通过 Tent 映射和自适应 t 分布来改进秃鹰搜索算法. 最后结合改进的秃鹰搜索算法 (TBESK) 将待调度的 Pod 分配在最优节点上, 加强了集群的负载能力, 有效提升了节点的资源利用率. 实验证明改进的秃鹰搜索调度算法的有效性合理性. 下一阶段将考虑 Pod 的部署速度、成本, 提高集群中 Pod 在节点的部署速度, 降低集群中 Pod 的部署成本.

参考文献

1 郑东旭. Kubernetes 源码剖析. 北京: 电子工业出版社, 2020.

2 唐瑞. 基于 Kubernetes 的容器云平台资源调度策略研究 [硕士学位论文]. 成都: 电子科技大学, 2017.

3 胡程鹏, 薛涛. 基于遗传算法的 Kubernetes 资源调度算法. 计算机系统应用, 2021, 30(9): 152–160. [doi: 10.15888/j.cnki.csa.008062]

4 Zhang WG, Ma XL, Zhang JZ. Research on Kubernetes resource scheduling scheme. Proceedings of the 8th International Conference on Communication and Network Security. Qingdao: ACM, 2018. 144–148.

5 孔德瑾, 姚晓玲. 面向 5G 边缘计算的 Kubernetes 资源调度策略. 计算机工程, 2021, 47(2): 32–38. [doi: 10.19678/j.issn.1000-3428.0058047]

6 Li D, Wei Y, Zeng B. A dynamic I/O sensing scheduling scheme in Kubernetes. Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications. Guangzhou: ACM, 2020. 14–19.

7 常旭征, 焦文彬. Kubernetes 资源调度算法的改进与实现. 计算机系统应用, 2020, 29(7): 256–259. [doi: 10.15888/j.cnki.csa.007545]

8 Dua A, Randive S, Agarwal A, *et al*. Efficient load balancing to serve heterogeneous requests in clustered systems using Kubernetes. Proceedings of the 2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC). Las Vegas: IEEE, 2020. 1–2.

9 何龙, 刘晓洁. 一种基于应用历史记录的 Kubernetes 调度算法. 数据通信, 2019, (3): 33–36. [doi: 10.3969/j.issn.1002-5057.2019.03.009]

10 张可颖, 彭丽苹, 吕晓丹, 等. 开源云上的 Kubernetes 弹性调度. 计算机技术与发展, 2019, 29(2): 109–114. [doi: 10.3969/j.issn.1673-629X.2019.02.023]

11 Li X, Qian ZZ, Lu SL, *et al*. Energy efficient virtual machine placement algorithm with balanced and improved resource utilization in a data center. Mathematical and Computer Modelling, 2013, 58(5–6): 1222–1235.

12 Liu D, Sui X, Li L. An energy-efficient virtual machine placement algorithm in cloud data center. Proceedings of the 2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD). Changsha: IEEE, 2016. 719–723.

13 Alsattar HA, Zaidan AA, Zaidan BB. Novel meta-heuristic bald eagle search optimisation algorithm. Artificial Intelligence Review, 2020, 53(3): 2237–2264. [doi: 10.1007/s10462-019-09732-5]

14 丁容, 高建瓴, 张倩. 融合自适应惯性权重和柯西变异的秃鹰搜索算法. 小型微型计算机系统, 2022: 1–9. http://kns.cnki.net/kcms/detail/21.1106.TP.20220418.1308.018.html. (2022-07-16).

15 Liu LF, Sun SZ, Yu HY, *et al*. A modified fuzzy C-means (FCM) clustering algorithm and its application on carbonate fluid identification. Journal of Applied Geophysics, 2016, 129: 28–35. [doi: 10.1016/j.jappgeo.2016.03.027]

16 单梁, 强浩, 李军, 等. 基于 Tent 映射的混沌优化算法. 控制与决策, 2005, 20(2): 179–182. [doi: 10.3321/j.issn:1001-0920.2005.02.013]

17 王梓坤. 概率论基础及其应用. 北京: 科学出版社, 1979.

18 黄敬宇. 融合 t 分布和 Tent 混沌映射的麻雀搜索算法研究 [硕士学位论文]. 兰州: 兰州大学, 2021.

19 韩斐斐, 刘升. 基于自适应 t 分布变异的缎蓝园丁鸟优化算法. 微电子学与计算机, 2018, 35(8): 117–121. [doi: 10.19304/j.cnki.issn1000-7180.2018.08.025]

20 柳子来, 王健敏. 基于自适应 t 分布的改进粒子群实时任务调度算法. 化工自动化及仪表, 2020, 47(5): 393–397, 424. [doi: 10.3969/j.issn.1000-3932.2020.05.005]

21 徐正伦, 杨鹤标. 基于 Kubernetes 调度器的服务质量优化调度算法研究. 软件导刊, 2018, 17(11): 73–76.

22 张文辉, 王子辰. 基于组合权重 TOPSIS 的 Kubernetes 调度算法. 计算机系统应用, 2022, 31(1): 195–203. [doi: 10.15888/j.cnki.csa.008251]

(校对责编: 孙君艳)

© 中国科学院软件研究所 http://www.c-s-a.org.cn