文章编号：2095-2163(2023)02-0001-06 中图分类号：TP311.5 文献标志码：A

基于 Kubernetes 的资源调度策略研究与改进

于泽川，张 娜，包梓群，苏鸿斌
（浙江理工大学 信息学院，杭州 310018）

摘 要：针对 Kubernetes 默认调度策略在多 Pod 调度时无法考虑多任务调度过程的全局特征，导致无法保证集群整体负载均衡的问题，本文设计了一种优化的静态资源调度策略 Ku-PSO，通过改进粒子群算法（PSO），提升集群负载均衡效率。首先，通过建立多 Pod 调度模型，以集群负载均衡度作为适应函数，并设置约束条件保证集群正常运行；其次，通过改进粒子群算法的惯性因子、个体学习因子和社会学习因子实现权值优化，使得粒子群在前期寻优过程中具有优秀的全局搜索能力，后期寻优过程中能够迅速收敛，应用于集群资源调度能够快速找出资源的最优分配方案。实验表明，使用 Ku-PSO 算法进行 Kubernetes 资源调度较默认调度策略集群均衡度显著提升，较 PSO 算法可以有效减少部署时间，实现更优的均衡调度。

关键词：Kubernetes；资源调度；粒子群算法；权值优化；均衡调度

Research and improvement of resource scheduling strategy based on Kubernetes

YU Zechuan, ZHANG Na, BAO Ziqun, SU Hongbin
(School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China)

【Abstract】 Aiming at the problem that the Kubernetes scheduling strategy cannot consider the global characteristics of the multi-task scheduling process when scheduling multiple Pods, resulting in the inability to ensure the overall load balance of the cluster, this paper designs an optimized static resource scheduling strategy Ku-PSO. It incorporates the particle swarm algorithm (PSO) to improve cluster load balancing efficiency. First, by establishing a multi-Pod scheduling model, the cluster load balance degree is used as the adaptation function, and constraints are set to ensure the normal operation of the cluster. Then, by improving the inertia factor, individual learning factor and social learning factor of the particle swarm algorithm with the weight optimization, the particle swarm has excellent global search ability the in the early optimization process. In later optimization process, the local optimization ability is improved, and it can converge quickly. When applied to cluster resource scheduling, it can quickly find the optimal allocation scheme of resources. Experiments show that the use of the Ku-PSO algorithm for Kubernetes resource scheduling significantly improves the cluster balance of the default scheduling strategy, and can effectively reduce the deployment time compared with the PSO algorithm and achieve better balanced scheduling.

【Key words】 Kubernetes; resource scheduling; particle swarm optimization; weight optimization; balanced scheduling

0 引 言

底层虚拟化技术的快速成熟，各大云计算厂商也开始积极建立自己的云部署平台，与此同时以 Kubernetes（以下简称 K8s）为代表的容器编排技术成为云原生的标准。K8s 集群中主要包括 api-server、kube-scheduler、controller-manager 3 大组件。其中，kube-scheduler 负责执行调度指令，将需要运行的 Pod（基于 Docker 实现的 K8s 的基本调度单位，提供一组服务的集合）调度到符合要求的 Node（负责负载 Pod 的基本单位）节点上去运行；kube-scheduler 的默认资源调度策略相对简单，但考虑不够全面，默认调度采用先到先服务的原则，选择负载服务的节点时采用贪心思想，保证当前最优，在多 Pod 调度的任务场景中无法保证全局最优[1-3]。由于容器云平台的资源调度存在很大的优化空间，由此引起了广大学者的研究改进。

张玉芳等[4]在考虑节点本身性能的情况下，为

基金项目：国家级大学生创新创业训练计划项目（202010338024）；浙江省教育厅一般科研项目（Y202147659）；浙江省重点研发计划项目（2020C03094）；国家自然科学基金（6207050141）。

作者简介：于泽川（1995-），男，硕士研究生，主要研究方向：服务器资源调度；张 娜（1977-），女，硕士，副教授，主要研究方向：软件工程、数据挖掘、智能信息处理。
通讯作者：张 娜 Email：zhangna@zstu.edu.cn

收稿日期：2022-04-18

不同资源给出不同计算权重,率先提出利用负载权值计算的方法;李华东等[5]提出基于合作博弈论的多资源负载均衡(Multi-resource load Balancing algorithm based on Cooperative Game Theory, MBCGT)算法,解决面对不同资源请求时的算法下界和集群资源碎片化现象;聂清彬等[6]提出基于改进蚁群算法的自适应云资源调度模型。目前,关于智能优化算法在资源调度方面的研究成为主流。Kaewkasi 等[7]将蚁群算法应用于 SwarmKit 集群进行容器调度,较贪心算法的性能有大幅提升;Mathiyalagan 等[8]借助网格计算编程的粒子群优化算法发展系统能力;王悦悦等[9]使用自适应神经网络构建云资源预测模型,解决资源供求不均衡导致的资源利用率差等问题。随着大量研究的进行,智能优化算法被广泛应用于优化资源调度,并取得不错的成果。在智能优化算法中,粒子群算法因其简单易行、收敛速度快被广泛关注,其借助多个粒子在解空间中的迭代寻优寻找最优解[10]。本文针对智能优化算法中的粒子群算法作出改进,通过动态改变惯性因子、个体学习因子以及社会学习因子的计算权重,改善其前期的全局寻优能力和后期的局部寻优能力,加快其后期收敛速度,将之运用在 K8s 的资源调度方面,建立多 Pod 调度模型,并将改进的粒子群算法和传统粒子群算法以及 K8s 默认的调度策略进行比较,证明改进后的粒子群算法对集群的负载均衡度有所改善。

1 K8s 默认调度策略分析与粒子群算法

1.1 K8s 默认资源调度策略分析

K8s 的默认资源调度过程分为预选和优选两个环节。在预选过程中,根据用户提交的 yaml 配置文件遍历所有 Node,过滤掉不符合要求的 Node,该过程保证了通过预选的节点都是足以运行当前 Pod 的节点。优选过程是为通过筛选的节点打分,整个选择过程通过预选阶段选择出可支持运行当前 Pod 的 Node 集合,作为下一阶段的输入,得到节点集合输入的优选过程以为各个节点打分为目标,得到分数最高的节点作为 Pod 落户的目标节点。打分范围是[0,10]之间的整数,通过分数表明该节点的优先级,10 代表最优,0 代表最差。当最高分节点不唯一,系统会从最高分节点中随机选择一个作为目标节点,存在一定随机性。

K8s 的默认调度策略保持着启发式调度策略中的先到先服务准则(FCFS),会将调度 Pod 按照先后顺序依次执行上述预选和优选的过程,在为 Pod 选择负载节点时则使用贪心思想,永远将当前 Pod 调度到得分最高的节点上去,追求局部忽略了整个多任务调度过程的全局性;其次,K8s 的调度策略只将 CPU 和内存作为资源调度的主要考量,忽略了网络带宽和磁盘 IO 的影响,Kubernetes 集群的综合性能离不开这两者的的支持,因此网络带宽和磁盘 IO 需要加入运算,完成对节点更完整的画像;最后,在待调度队列中 Pod 被逐个调度时,同一副本下的调度任务可能创建多个副本,相同副本拥有相同的资源需求量,在进行相同副本的首次调度时,对符合要求的 Node 节点已经进行过打分,在调度下一个相同副本下的 Pod 时,大部分的节点打分仍然具有意义,但在 K8s 的调度策略中无法复用。基于上述问题,本文将对粒子群算法进行优化,并应用于 K8s 的资源调度,解决上述问题。

1.2 粒子群算法

粒子群算法(PSO)是一种智能优化算法,由 Kennedy 和 Eberhart 于 1995 年提出,作为一种基于群体智能(Swarm Intelligence)的优化方法,具有简单易行,收敛速度快和设置参数少的特点[11]。PSO 通过在解空间中部署大量粒子来寻优。粒子仅具有两个重要属性,速度和位置。速度代表移动的快慢,从某种程度上反映了单次移动的步长;位置代表其作为一个解自身所对应的属性。一个粒子通过速度计算出在某维度上的速度,即会发生在该维度上的移动距离,当所有维度的数据特征聚集在一起就完成了一个粒子在解空间中的位置改变[12]。

首先,在解空间中给出大量粒子,每个粒子需要在解空间具有一个初始位置和速度,这些粒子要求均匀分布于解空间中,以防止结果陷入局部最优,无法收敛于全局最优。

粒子的速度更新公式(1)如下:

$$v_{i+1} = \omega v_i + c_1 \times \text{random1}() \times (Pbest_i - x_i) + c_2 \times random2() \times (Gbest_i - x_i) \quad (1)$$

其中,ω 为惯性因子,取值一般为 0.9,为保持上一步移动的惯性,来对下一步移动产生影响,决定了粒子继承先前移动轨迹的权重;c_1 是个体学习因子,表明自身历史最优值对当前移动的影响;c_2 是社会学习因子,表明群体最优粒子对当前粒子的吸引力,c_1 和 c_2 的取值一般为 2;$Pbest$ 是当前自身历史最优位置;$Gbest$ 是当前群体最优位置;x_i 就是当前位置;i 为迭代次数;$random1()$ 和 $random2()$ 为两个随机函数,随机产生范围在[0,1]的值,用于

增强粒子寻优的随机性，防止粒子过早收敛，全局寻优能力降低，陷于局部最优。

单维度上的粒子的位置更新公式(2)为

$$x_{i+1} = x_i + v_{i+1} \tag{2}$$

其中，x 代表相对位置，i 为迭代次数。

由此可见，速度直接决定了位置变化。

ω 越大，当前粒子群的全局搜索能力越强，ω 越小，当前粒子群的局部搜索能力越强。所以在实际应用中，为保证粒子群在迭代前期拥有较好的全局搜索能力，在后期可以有较强的局部寻优能力从而迅速收敛，ω 通常是动态的，常用的动态变换公式(3)为

$$\omega_{r+1} = \omega_{\max} - (\omega_{\max} - \omega_{\min}) \times \frac{r}{r_{\max}} \tag{3}$$

其中，$\omega_{\max}$ 和 $\omega_{\min}$ 分别代表惯性因子的上下界，选值通常为 0.9 和 0.4；r 是迭代次数；$r_{\max}$ 为预设的最大迭代次数，本次取值 200。

随着迭代次数的不断增加，惯性因子 ω 完成了从最大值到最小值的线性变化，某种程度上摆脱了固定惯性因子大小带来的容易陷入局部最优的缺陷。

设置迭代次数后，对初始化粒子的位置还有个体历史最优位置和全局最优位置进行迭代更新，而确认个体历史最优位置和全局最优位置的方法就是通过适应函数计算粒子的适应值，结束条件通常为结果收敛或已经达到预设迭代次数。

2　粒子群算法改进与应用研究

2.1　改进粒子群算法 Ku-PSO

PSO 算法通过迭代寻优，具有易于实现，易于理解，设置参数少等特点，但其仍存在易陷入局部最优解，难以得到精确的全局最优解等缺陷。本文对传统 PSO 算法作出改进，改进后的算法称为 Ku-PSO 算法，前期迭代更为依赖个体学习，拥有更好的全局寻优能力，后期迭代更为依赖社会学习，拥有更好的局部搜索能力，利于收敛。在整个搜索过程中，当前位置适应度较低时给出更大速度，扩展更多可能性，在当前位置适应度较高时给出相对较高的惯性因子，提升收敛速度。

公式(1)中 ω，c_1 和 c_2，分别充当了前一次迭代速度，前一次迭代全局最优位置和自身粒子历史最优位置参与计算当前迭代速度的权重。

首先，对 ω 作出改进。ω 较大时，粒子会拥有更好的全局寻优能力；ω 较小时，粒子会拥有更好的局部寻优能力，故提出 ω 的动态变化公式(3)，使得 ω 完成了由 $\omega_{\max}$ 到 $\omega_{\min}$ 的线性变化。

此处将 ω 的线性变化公式由公式(3)替换公式(4)：

$$\omega_{r+1} = \omega_{\max} - (\omega_{\max} - \omega_{\min}) \times \frac{r^2}{r_{\max}^2} \tag{4}$$

借助抛物线，获得更为合理的由 $\omega_{\max}$ 到 $\omega_{\min}$ 的非线性变换。由于粒子群算法易陷入局部最优解，通过非线性变换使得 ω 在前期迭代中缓慢减小，后期迭代中快速减小，从而增强前期全局寻优能力和后期的局部收敛能力。

在此基础上，为增强粒子的全局寻优能力，设计在粒子适应度较小时，继续维持较大的速度，从而获得更大的步长，得到更广阔的解空间搜索范围。因此对 c_1 和 c_2 给出式(5)变化，假设公式(1)中 $c = c_1 = c_2$。

$$c = \begin{cases} c_{\min} + (c_{\max} - c_{\min}) \times \tanh\left(\dfrac{F - F_{avg}}{F_{\max} - F_{avg}} \times \theta\right), & F \geqslant F_{avg} \\ c_{\max}, & F < F_{avg} \end{cases} \tag{5}$$

其中，$c_{\min}$ 即为 c 的最小值，取值为 1；$c_{\max}$ 即为 c 的最大值，取值为 3；F 为当前粒子通过适应函数计算出的适应度；$F_{\max}$ 为前一次迭代全局最佳适应度；F_{avg} 为前一次迭代的粒子群平均适应度；θ 为曲线平滑度控制因素，此处取值为 3。

$\tanh(x)$ 为双曲正切函数，其特征 x 在 $(0,+\infty)$ 的范围内，其值在 $(0,1)$ 上非线性递增，斜率逐渐降低。设置 θ 为 3，边界更靠近 1，使得 c 完成了从 $c_{\min}$ 到 $c_{\max}$ 的非线性变换。随着迭代次数增加，从一定程度上降低了惯性因子的影响，使得自身收敛特征更为明显。通过分段函数使得粒子在自身适应度较低时，可以通过一个更大的速度移动，从而探索更多的解空间，而在自身适应度高于均值时，进行正常的收敛。这样做防止了粒子过早地陷入局部最优，保存了种群多样性。

为再次强化前期的全局搜索能力和后期的局部搜索能力，有必要在前期给个体学习因子 c_1 更高的权重，在后期将权重逐渐偏向社会学习因子 c_2。

于是给出公式(6)和公式(7)，使得在前期迭代中，个体学习因子 c_1 占较大比重，可以增强前期的全局寻优能力，探索更大的解空间；后期迭代中，社会学习因子 c_2 比重逐渐升高，可以在基本确定全局最优范围后迅速收敛。

$$c_1 = c \times (1 - \frac{r^2}{r_{\max}^2}) \tag{6}$$

$$c_2 = c \times \frac{r^2}{r_{\max}^2} \tag{7}$$

Ku-PSO 针对惯性因子,个体学习因子,社会学习因子均给出了优化方案,使得前期的寻优能力大幅提升的同时不影响后期粒子群的收敛速度,基本解决了粒子群算法容易陷入局部最优的问题,借此完成均衡调度。

2.2 Ku-PSO 在 K8s 资源调度的应用

通过 Ku-PSO 算法针对 K8s 集群多 Pod 应用部署创建模型。

将 n 个 *Pod* 应用部署到 m 个负载应用的工作节点上,设置对应配置矩阵。部署 *Pod* 集合为 $Pods = \{p_1, p_2, p_3, \cdots, p_n\}$,负载节点集合 $Nodes = \{n_1, n_2, n_3, \cdots, n_m\}$,为 *Pods* 中每个 P_i 在 *Nodes* 中选择一个负载节点 n_j 进行部署。

配置矩阵如下:

$$\boldsymbol{D} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & \cdots & d_{1m} \\ d_{21} & d_{22} & d_{23} & \cdots & d_{2m} \\ d_{31} & d_{32} & d_{33} & \cdots & d_{3m} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ d_{n1} & d_{n2} & d_{n3} & \cdots & d_{nm} \end{bmatrix}$$

d_{ij} 取值为 0 或 1,为 1 代表将第 i 个 *Pod* 部署到了第 j 个 Node 上运行,为 0 则相反。一个矩阵 $\boldsymbol{D}$ 就是一种调度方案。在 Ku-PSO 算法中,一个粒子代表一个配置矩阵,即代表一种调度方案。

K8s 默认调度算法只将 CPU 和内存性能考虑其中,忽略了网络带宽和磁盘 IO 对 Pod 部署的影响。在网络应用和涉及存储的 Pod 部署时,网络带宽和磁盘 IO 也同样重要,此次在考虑 CPU 和内存的基础上将网络带宽和磁盘 IO 也加入运算,通过时序数据库 InfluxDB 来获取各个 Node 节点的资源总量 {Ccpu,Cmem,Cnet,Cdisk},和各个 Pod 的资源请求数值 {Rcpu,Rmem,Rnet,Rdisk}。

接下来为粒子群算法设置目标函数,即为适应函数,式(8),对所有的方案进行筛选和优化比较。

$$F = m/[\sum_{j=1}^{m}(|U_j^{cpu} - U_{avg}^{cpu}| + |U_j^{mem} - U_{avg}^{mem}| + |U_j^{net} - U_{avg}^{net}| + |U_j^{disk} - U_{avg}^{disk}|)] \tag{8}$$

其中,U 代表资源的利用率;j 为 Node 的编号;U_{avg} 代表所有工作节点上的平均利用率。

借此式来表示一种部署方案下的集群负载均衡度,式(9):

$$U_{avg} = \frac{1}{m}\sum_{j=1}^{m} U_j \tag{9}$$

设置好部署矩阵和评判粒子好坏的适应函数,接下来需要为配置矩阵设置约束条件,避免 Pod 应用部署时产生矛盾。

首先,每个 Pod 最多只能部署到一个 Node 上运行,故存在式(10):

$$\sum_{j=1}^{m} d_{ij} = 1 \tag{10}$$

其次,各 Node 节点上部署到 Pod 请求资源总量不得超过当前节点上拥有的资源总量。故存在式(11):

$$\sum d_{ij} \times R < C \tag{11}$$

其中,i 为部署到第 j 个 Node 上的 Pod 编号。

上述过程按照标准的粒子群算法改进,但 K8s 多 Pod 任务部署过程是一个离散型问题,需要在 Ku-PSO算法的基础上,进行二进制离散粒子群的映射。标准粒子群算法使用 sigmoid 函数进行映射,通过映射的方式得到映射概率,可以计算出当前粒子在该维度突变为 1 的概率,式(12):

$$s(v_{ij}) = \frac{1}{1 + \exp(-v_{ij})} \tag{12}$$

其中,v_{ij} 为标准粒子群计算所得速度。

位置更新公式(13):

$$d_{ij} = \begin{cases} 0, random() \leqslant s(v_{ij}) \\ 1, random() > s(v_{ij}) \end{cases} \tag{13}$$

其中,$random()$ 为随机函数,计算得值范围为 [0,1]。

然而这种计算方式已被刘建华[13]等学者证明:随着迭代次数增加,其突变的随机性会逐渐升高,在最终迭代次数时达到最大,使得后期需要收敛的结果产生了更多不确定性,不易收敛到全局最优值。

在此使用刘建华等学者改进的方法,将式(12)改为式(14):

$$s(v_{ij}) = \begin{cases} 1 - \dfrac{2}{1 + \exp(-v_{ij})}, & v_{ij} \leqslant 0 \\ \dfrac{2}{1 + \exp(-v_{ij})} - 1, & v_{ij} > 0 \end{cases} \tag{14}$$

速度在原点左右对称,为偶函数。速度为正时,概率函数递增;速度为负时,概率函数递减。

对应的位置变化公式由(13)改为式(15)~式(16):

当 $v_{ij} < 0$ 时，

$$d_{ij} = \begin{cases} 0, random() \leqslant s(v_{ij}) \\ d_{ij}, random() > s(v_{ij}) \end{cases} \quad (15)$$

当 $v_{ij} > 0$ 时，

$$d_{ij} = \begin{cases} 1, random() \leqslant s(v_{ij}) \\ d_{ij}, random() > s(v_{ij}) \end{cases} \quad (16)$$

为避免 $s(v_{ij})$ 过于接近极值 1，可通过 $v_{ij} \in [-v_{max}, v_{max}]$ 进行限制。通过二进制映射关系，使得粒子群算法应用到二进制离散的 K8s 多 pod 部署问题。

对粒子进行初始化，其位置初始化过程，式(17)：

$$d_{ij} = \begin{cases} 0, random() \leqslant 0.5 \\ 1, random() > 0.5 \end{cases} \quad (17)$$

速度初始化公式(18)为

$$v_{ij} = v_{min} + random() \times (v_{max} - v_{min}) \quad (18)$$

其中，$v_{min} = -2, v_{max} = 2$。

3　实验设计

3.1　实验过程

实验环境采用 Kubernetes1.18，通过虚拟机的形式部署在 PC 上，集群中包括 1 个 Master 节点和 3 个用于负载 Pod 的 Node 节点。

构建 4 台虚拟机作为 Node 节点，具体参数见表 1。

表 1　集群节点配置

Tab. 1　Configuration of cluster nodes

Node 名称	操作系统	CPU 核数	内存/GB	网络带宽/Mbps	磁盘 IO/Mbps
Master	CentOS7	2	2	100	100
Node1	CentOS7	2	4	50	100
Node2	CentOS7	4	6	100	50
Node3	CentOS7	2	2	100	100

部署 Pod 描述：本次部署选取应用场景最为丰富的网络服务 web 应用，该应用使用 SpringBoot 开发，接受无参请求，通过循环从 100 到 1 的阶乘计算，并在循环中进行当前结果的字符串拼接，并在计算完成后，存储进入文件中，其结果会返回给前端。web 应用打包为 Docker 镜像，该 Pod 名称成为 test-forpso。

测试流程：通过 yaml 文件进行文件部署，镜像在私有镜像仓库拉取，每个 yaml 文件中定义的 Pod 使用 CPU、内存、磁盘的资源请求量见表 2。

表 2　Pod 资源请求量

Tab. 2　Resource requests for Pods

资源类型	请求量
CPU(核数)	0.2
内存/MB	100~200
网络/Mbps	2
磁盘/Mbps	2~4

test-forpso 部署数量为 5 个，依次递增至 50 个，每次调度在提升数量的同时，删除已部署的 Pod，借此计算部署时间。

实验中使用默认调度算法，PSO 和 Ku-PSO 分别进行实验，并对结果进行比较。

3.2　实验结果分析

通过实验，分别统计 test-forpso 不同部署个数的负载均衡度。结果如图 1 所示。

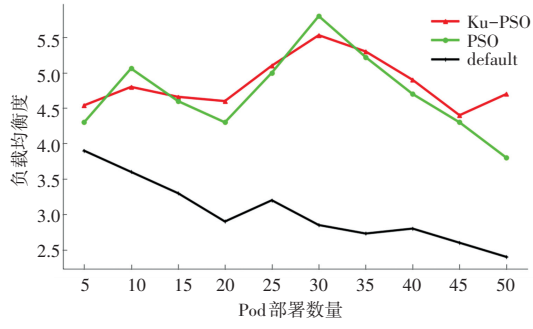

图 1　负载均衡度对比

Fig. 1　Comparison of changes in load evaluation

对比 3 种调度方法的负载均衡度，Ku-PSO 明显更为稳定，且随着调度的 Pod 数目不断增多的过程中，其全局寻优能力更为明显，反观 PSO 算法更容易陷入局部最优导致负载均衡度降低，而默认的调度方法在多 Pod 调度时无法观察整个调度过程全局的特征，因而导致负载均衡度下降严重。

分别统计不同调度个数的时间消耗，结果如图 2所示。

(下转第 14 页)