软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):1005−1017 [doi: 10.13328/j.cnki.jos.006451]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

新型分布式计算系统中的异构任务调度框架*

刘瑞奇[1], 李博扬[1], 高玉金[1], 李长升[1], 赵恒泰[2], 金福生[1], 李荣华[1], 王国仁[1]

[1](北京理工大学 计算机学院, 北京 100081)

[2](东北大学 计算机科学与工程学院 辽宁 沈阳 110819)

通信作者: 李博扬, E-mail: liboyang@bit.edu.cn

摘 要: 随着大数据和机器学习的火热发展, 面向机器学习的分布式大数据计算引擎随之兴起. 这些系统既可以支持批量的分布式学习, 也可以支持流式的增量学习和验证, 具有低延迟、高性能的特点. 然而, 当前的一些主流系统采用了随机的任务调度策略, 忽略了节点的性能差异, 因此容易导致负载不均和性能下降. 同时, 对于某些任务, 如果资源要求不满足, 则会导致调度失败. 针对这些问题, 提出了一种异构任务调度框架, 能够保证任务的高效执行和被执行. 具体来讲, 该框架针对任务调度模块, 围绕节点的异构计算资源, 提出了概率随机的调度策略 resource-Pick_kx 和确定的平滑加权轮询算法. Resource-Pick_kx 算法根据节点性能计算概率, 进行概率随机调度, 性能高的节点概率越大, 任务调度到此节点的可能性就越高. 平滑加权轮询算法在初始时根据节点性能设置权重, 调度过程中平滑加权, 使任务调度到当下性能最高的节点上. 此外, 对于资源不满足要求的任务场景, 提出了基于容器的纵向扩容机制, 自定义任务资源, 创建节点加入集群, 重新完成任务的调度. 通过实验在 benchmark 和公开数据集上测试了框架的性能, 相比于原有策略, 该框架性能提升了 10%–20%.

关键词: 任务调度; 负载均衡; 自动扩容; 分布式计算; 异构任务

中图法分类号: TP311

中文引用格式: 刘瑞奇, 李博扬, 高玉金, 李长升, 赵恒泰, 金福生, 李荣华, 王国仁. 新型分布式计算系统中的异构任务调度框架. 软件学报, 2022, 33(3): 1005–1017. http://www.jos.org.cn/1000-9825/6451.htm

英文引用格式: Liu RQ, Li BY, Gao YJ, Li CS, Zhao HT, Jin FS, Li RH, Wang GR. Heterogeneous Task Scheduling Framework in Emerging Distributed Computing Systems. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 1005–1017 (in Chinese). http://www.jos.org.cn/1000-9825/6451.htm

Heterogeneous Task Scheduling Framework in Emerging Distributed Computing Systems

LIU Rui-Qi[1], LI Bo-Yang[1], GAO Yu-Jin[1], LI Chang-Sheng[1], ZHAO Heng-Tai[2], JIN Fu-Sheng[1], LI Rong-Hua[1], WANG Guo-Ren[1]

[1](School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

[2](School of Computer Science and Technology, Northeastern University, Shenyang 110819, China)

Abstract: With the rapid development of big data and machine learning, the distributed big data computing engine for machine learning have emerged. These systems can support both batch distributed learning and incremental learning and verification, with low latency and high performance. However, some of them adopt a random task scheduling strategy, ignoring the performance differences of nodes, which easily lead to uneven load and performance degradation. At the same time, for some tasks, if the resource requirements are not met, the scheduling will fail. In response to these problems, a heterogeneous task scheduling framework is proposed, which can ensure the efficient execution and execution of tasks. Specifically, for the task scheduling module, the proposed framework proposes a probabilistic random scheduling strategy resource-Pick_kx and a definite smooth weighted round-robin algorithm around the heterogeneous computing

* 基金项目: 国家重点研发计划(2018YFB1004402); 国家自然科学基金(U2001211, 62072034, 61772346); 中国博士后科学基金(2021M690397)

本文由“数据库系统新型技术”专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.

收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

resources of nodes. The resource-Pick_kx al-gorithm calculates the probability according to the performance of the node, and performs random scheduling with probability. The higher the probability of a node with high performance, the higher the possibility of task scheduling to this node. The smooth weighted round-robin algorithm sets the weights according to the node performance at the beginning, and smoothly weights during the scheduling process, so that the task is scheduled to the node with the highest performance. In addition, for task scenarios where resources do not meet the requirements, a container-based vertical expansion mechanism is proposed to customize task resources, create nodes to join the cluster, and complete task scheduling again. The performance of the framework is tested on benchmarks and public data sets through ex-periments. Compared with the current strategy, the performance of the proposed frame is improved by 10% to 20%.

Key words: task scheduling; load balance; autoscale; distributed computing; heterogeneous task

现今, 大数据处理模型大致可分为 3 类, 分别为批处理系统、流处理系统和针对机器学习的任务并行系统. 批处理系统最具代表性的是 MapReduce(Hadhoop)[1]、Spark[2]等, 流处理系统则有 Flink[3]、Storm[4]等, 而针对机器学习的大数据处理系统还处于发展阶段. Spark 有 Spark MLlib[5]作为常用机器学习算法的实现库, 训练是并行的但很不完善. Tensorflow[6]是主要针对机器学习的数据流图式的数值计算库, 但对分布式的处理并不十分友好. 在此基础上, Ray[7]应运而生, 填补了专门针对机器学习分布式系统的空缺. Ray 是与 Spark 同源的 UC Berkeley RISELab 出品的高可用、高性能的机器学习分布式引擎, 支持分布式异步调用, 灵活高效.

Ray 与经典的大数据计算系统相比, 除了编程模型的不同, 主要区别在于其在任务执行的过程中生成动态 DAG 图, 即来即处理, 是一种典型的在线场景; 其次, 调度架构上减少了主从结构, 采用去中心化的灵活调度方式, 设置本地调度器, 其上的调度策略对系统性能起着重要的作用. Ray 根据任务的资源要求得到备选节点集合, 再采用随机调度算法得到目标节点. 显然, 随机算法简单, 能获得较平均的结果. 但其未考虑调度时备选节点间的性能差异, 容易导致负载不均衡的情况发生. 与此同时, 由于 Ray 是根据任务的资源要求调度的,当有任务的资源要求大于集群任意节点的配置时, 任务则变为不可用. 因此, Ray 需要自动扩容的机制来满足不同的任务调度. 现有的 Ray 结合云环境、Kubernetes[8]等提供了原生的弹性伸缩机制, 但其扩容方式主要体现在横向扩容, 能够在相同任务造成的负载压力较大的情况下增加默认节点, 但是每个节点的资源配置是相同的, 当任务资源超过节点的最大上限时, 任务依旧无法被调度.

针对以上出现的问题, 本文主要贡献如下:

(1) 提出一种新的异构任务调度框架, 包括任务调度和自动纵向扩容两部分, 保证任务的高效执行和被执行.

(2) 改进并设计了两种任务调度算法: 一种是随机的 resource-Pick_kx 算法, 通过概率来随机调度任务; 一种是确定性算法, 通过平滑加权轮询来实现任务资源要求下的负载均衡.

(3) 提出了自动纵向扩容方法, 借助容器, 根据任务的资源要求进行节点创建, 将不可用任务转化为可用, 实现自动纵向扩容.

(4) 对不同规模的数据集进行任务调度的对比实验及不可用任务的可调度性实验, 表明了本文提出的框架在吞吐量、运行时间及扩容上的优越性.

1 相关工作

大数据分布式计算引擎的任务调度策略对系统性能起着至关重要的作用. 在批处理系统中, 资源管理系统主要负责任务的调度, 如: Yarn[9]按照计算向数据靠拢的原则, 作业尽量分配给存储数据的节点; Mesos[10]实现了两级调度架构, 通过资源邀约来决定任务的调度结果等. 流处理系统中, Flink 和 Storm 默认的任务调度机制都是轮询算法. 文献[11]提出了感知资源的调度算法 R-Storm, 将内存资源和网络资源对不同节点的分配来调度任务, 来最大化利用资源. 文献[12]提出了专门针对异构环境的 G-Storm 调度算法. 文献[13]是针对云计算环境下的任务调度算法, 采用遗传模拟退火算法[14]. 文献[15]面向云数据中心提出了启发式的多目标灰狼算法, 旨在协调互相冲突的调度目标, 如吞吐量、最大完成时间及资源利用率等, 来寻找接近最优的解决方法. 文献[16]则是针对异构多核系统提出了能源和温度感知的实时调度器, 在有限的上下文切换下, 提高了资

源利用率. 文献[17]提出了面向云雾环境的感知高效调度算法, 雾计算是将云计算推向网络边缘的一种新型计算结构, 延迟低且具有异构、分布的动态资源. 主要通过提出资源感知调度器 RACE 来将传入的应用模块分配到雾层设备上, 最大限度地提高雾层的资源利用率, 以最小的应用程序执行时间和最小的带宽利用率降低使用云资源的金钱成本. 文献[18]研究的是分布式计算中的同异步模型, 针对基于局部信息的移动实体完成全局任务而提出的一种半异步的分布式调度器. 而对于 Ray 本身来说, 其默认的任务调度机制是随机方法.

负载均衡算法按照是否获取集群的实时负载可分为静态负载均衡算法和动态负载均衡算法. 常见的静态负载均衡算法有随机、轮询、一致性哈希[19]等, 动态负载均衡算法则包括最少连接数、最快响应速度、Pick-kx 算法[20]、DAWRRRLB 算法[21]和 Basic LI 算法[22]等. 静态算法在调度前给定分配策略, 在调度时不随负载变化, 调度简单快速, 但不能充分利用集群中的资源, 容易导致负载不均. 动态算法根据集群实时的负载分配状况进行调度, 有利于负载均衡, 但实时状态的获取消耗系统性能并且存在延迟, 无法做到真正的实时.

随着虚拟化技术尤其是容器的发展和普及, 扩容变得愈发简便和重要. 容器编排工具 Kubernetes 具有弹性伸缩的能力, 可以通过 metric-server 监控集群的 CPU 利用率和内存利用率, 从而进行自动扩缩容. 其中, 是为现有 Pod 分配更多的 CPU 或内存, 横向扩容已投入使用, 但纵向扩容还处于测试阶段. Kubernetes 的纵向扩容(vertical pods autoscaler, VPA)可以监视所有 Pod 的历史资源使用情况和内存不足事件, 来建议要求的资源值. 其原理是内部推荐器使用一些智能算法来根据历史指标计算内存和 CPU 值, 通过将现有的 Pod 重启重新配置得到推荐资源的 Pod. 文献[23]基于 Kubernetes 提出一种自动缩放机制, 结合响应式扩展与弹性伸缩容忍度. 文献[24]基于 Kubernetes 提出了自适应弹性伸缩机制, 能够自动地检测得到一个 Pod 最适合的资源. 同时, 文献[25]采用机器学习的方法进行预测, 使应用可以提前根据负载状况调整, 以提高资源利用率. 但以上和 Ray 的使用场景并不符, Ray 已知需要的资源并不需要推荐, 并且以上都是针对 Kubernetes 来实现, 并不是从应用系统的角度进行设计实现, 在系统层面缺少通用性.

2 Ray 系统基本介绍

Ray 是针对机器学习的分布式计算引擎, 系统架构如图 1 所示. Ray 集群有且仅有一个头节点(head node)和若干个工作节点(worker nodes). 全局控制存储(global control store, GCS)位于头节点, 使用 redis[26]实现, 包含各种表数据来存储全局状态. Ray 采用分布式调度, 所以在集群中的任意节点都存在有本地调度器(scheduler). 在每个节点, 驱动进程(driver)用来提交任务, 工作进程(worker)用来执行任务, 这两者的数量取决于节点的 CPU 数量和用户提交任务的数量. 即存在任务在本地节点提交时, 此节点才有驱动进程产生; 否则没有. 如图 1 所示, 工作节点 1 没有任务提交. 但工作进程每个节点都存在, 个数默认为本地节点的 CPU 数. 在整个 Ray 集群中, 任意两个节点都可进行通信. 由于 Ray 依赖于每个节点的本地调度器, 整个调度过程没有中心调度节点的瓶颈限制, 因此更贴合当下机器学习对低延迟、高吞吐的要求.

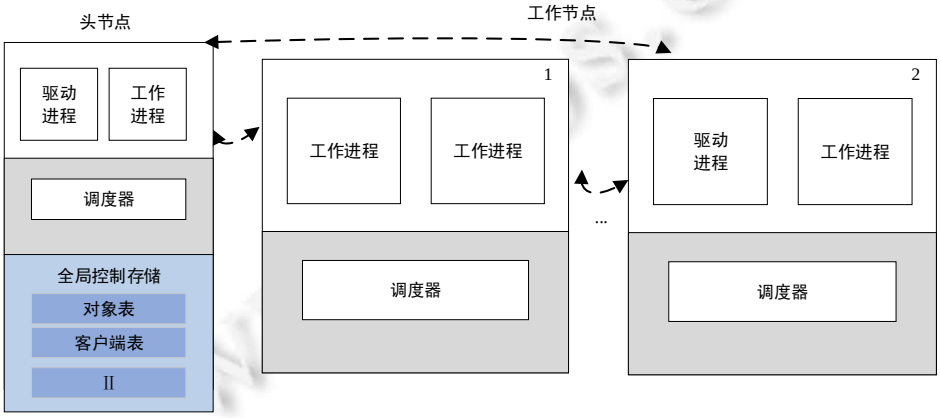


图 1 Ray 系统架构图

在任务(task)调度方面，调度器将用户提交的任务分配给集群中的任意可用节点，特点如下:

(1) 任务之间可以互相独立也可以有依赖.

(2) 任务带有资源要求，默认或自定义皆可.

(3) 节点具备资源信息.

资源是 Ray 任务调度和扩容的重要信息，共分为 3 类: 整体资源、可用资源和负载资源. 整体资源是通过启动集群初始化时自动获取或指定的; 可用资源是集群中目前不被任务所占用的空闲资源; 负载资源是任务要求的资源. 每个节点都具备上述 3 类资源. 由于 Ray 是专门面向机器学习的分布式计算系统，所以能够识别 CPU、GPU 等异构计算资源. 用户通过指定异构资源需求创建异构任务. 因此，本文所指的异构资源和异构任务定义如下:

定义 1(异构资源). 在 Ray 计算系统当中，可以使用不同类型的指令集和体系架构的处理器进行联合计算，目前包括 CPU 和 GPU，作为系统的异构计算资源.

定义 2(异构任务). 用户可以提交带有异构资源类型要求的任务，如 *task*1{*num_cpus*=4,*num_gpus*=1}，称为异构任务. 异构任务会通过任务调度算法分配到满足资源要求的异构节点上

Ray 的任务调度过程如图 2 所示，用户在本节点提交任务后，由调度器优先本地调度执行，若不满足资源要求，则提交到其他远程节点调度，称为溢出调度(spillover)，具体定义如定义 3. 溢出调度是迭代的，直到任务找到资源满足的节点为止. 本文就是基于上述调度过程设计调度算法.

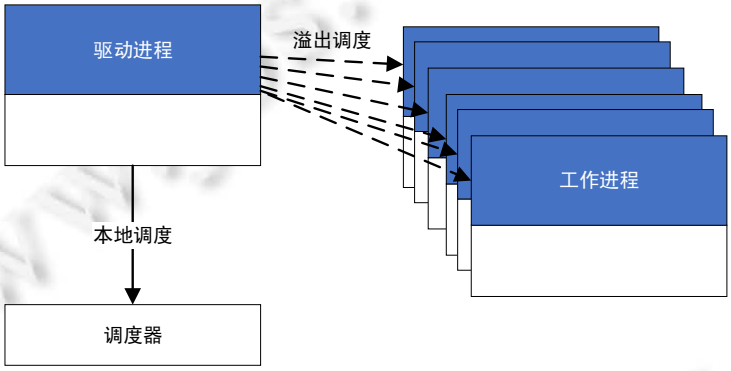


图 2 Ray 调度过程图

定义 3(溢出调度). 由于本地资源无法满足任务的调度执行，所以任务被调度到集群中的其他节点，此调度过程称为溢出调度.

3 算 法

3.1 整体思路

本文所提出的异构任务调度框架基于异构资源，面向异构任务，通过分布式调度算法和扩容策略合力改善和解决了任务分配到哪个节点的问题，整体思路如算法 1 所示.

算法 1. 异构任务调度框架算法.

输入: 调度队列 *task_queue*，集群资源 *cluster_resources*.

输出: 调度结果、扩容结果.

算法描述:

1: *cluster init*

2: pop front *task* in *task_queue*

3: **if** *task.dem*⊂*local_node.res*

4: schedule in *local_node*
5: *Spillover*(·)
6: **if** *task* is *infeasible*
7: *Autoscale*(·)

具体来讲, 集群创建后首先进行初始化, 用户提交的任务会进入到调度队列中, 本地的任务调度器会对任务执行调度操作(第 1 行、第 2 行). 通过任务自身携带的资源要求(*task.dem*), 本地调度器首先会判断自身节点的资源(*local_node.res*)是否满足: 若满足, 则将任务调度到自身节点, 即本地优先原则(第 3 行、第 4 行); 若本地可用资源不满足, 则执行溢出调度, 通过遍历集群中的所有节点来得到满足资源要求的节点的集合, 之后根据负载均衡算法在此节点集合上调度得到分配结果(第 5 行). 然而, 还有一种情况即任意节点的资源都无法满足此任务的资源要求, 此时任务转化为 infeasible 状态, 被挂起. 显然, 调度算法无法解决此问题, 集群需要加入满足任务要求的节点, 也就是进行扩容操作. 普遍采用的横向扩容只能复制默认配置的节点, 纵向扩容机制灵活多变, 可根据任务的资源要求创建节点. 每当集群中加入新节点, Head node 会遍历 infeasible 任务, 当节点满足要求时, 会使对应挂起的 infeasible 任务重新转为就绪态, 重新调度至新加入的节点上(第 6 行、第 7 行). 至此, 一个任务在当前调度器下的分配过程完整结束.

在整个调度框架中, 溢出调度, 即 Spillover 算法是调度中的核心. 如算法 2 所示, 先遍历集群中的所有节点得到可用资源(*RAY_AVA*(*node*))满足的节点集合(*RAY_ava_nodes*) (第 1 行–第 3 行), 通过负载均衡调度算法(loadbalance)在此节点集合上调度得到分配结果(第 4 行、第 5 行), 负载均衡算法为溢出调度的重心, 下文会进行详细介绍; 若集群中没有节点的可用资源满足(第 6 行), 则再次遍历集群中的所有节点得到整体资源(*RAY_TOT*(*node*))满足的节点集合(RAY_total_nodes) (第 7 行–第 9 行), 再通过调度算法在此节点集合上调度得到分配结果(第 10 行、第 11 行); 否则, 将 *task* 标记为 *infeasible*(第 13 行).

算法 2. Spillover 算法.

输入: 调度队列 *schedule_queue*, 集群资源 *cluster_resources*.

输出: 调度键值对结果 *decision*.

算法描述:

1: **for each** node in *cluster_resources* **do**
2: **if** *task.dem*⊂*RAY_AVA*(*node*) **then**
3: *RAY_ava_nodes.push*(*node*)
4: **if** *RAY_ava_nodes* not empty **then**
5: *decision.push*(*task*:*loadbalance*)
6: **else**
7: **for each** node in *cluster_resources* **do**
8: **If** *task.dem*⊂*RAY_TOT*(*node*) **then**
9: *RAY_total_nodes.push*(*node*)
10: **if** *RAY_total_nodes* not empty **then**
11: *decision.push*(*task*:*loadbalance*)
12: **else**
13: *task* is *infeasible*

3.2 负载均衡算法

对于负载均衡的调度算法, 首先依据 Ray 原生的随机调度策略进行改进, 普通的随机只能做到节点间任务的平均而未考虑集群资源状况和负载差异. 由此类比 Pick-kx[20]算法, 通过为节点设置概率来完成概率随机调度.

首先, Pick_kx 算法原理为: 在动态更新的周期中, 当有任务提交后, 从所有节点 $N_1,N_2,\ldots,N_n$ 中随机选择

节点, $W_1,W_2,\ldots,W_k$, 并设它们的负载分别为 $L_1,L_2,\ldots,L_k$, 则将当前任务以概率 P_j 分配给节点 W_j, 其中,

$$P_j=\frac{X_j}{\sum_{i=1}^{k}X_i} \tag{1}$$

$$L_{total}=\sum_{i=1}^{k}L_i \tag{2}$$

$$X_j=\frac{L_{total}-L_j}{L_{total}} \tag{3}$$

Pick-kx 算法主要依据集群各个节点的负载信息, 公式(2)和公式(3)计算得到集群的整体负载和除当前节点外的负载占总体负载的比重 X_j, 公式(1)通过 X_j 之比得到每个节点的概率, 当集群其余节点的负载比重越大, 则任务调度到本节点的概率越大. 即负载信息是调度的关键信息, 但节点的处理任务的能力不单单是由负载信息决定的, 当可以获取到更多集群状态时, 每个节点资源的使用状况同样是一个重要因素.

对集群中的每个节点来说, 存在可用资源时, 可用资源等于整体资源与负载资源之差, 即 $A=T-L$. 又有:

$$A_{total}=\sum_{i=1}^{k}A_i \tag{4}$$

$$P'_j=\frac{A_j}{\sum_{i=1}^{k}A_i} \tag{5}$$

所以, 得出每个节点的负载资源与可用资源负相关. 同时, 由公式(3)得出负载资源 L 与 X 负相关, 则 A 与 X 正相关, 那么概率 P_j 与 P'_j 正相关. 概率 P_j 越大的节点, 表明当前负载越小. 同理, 概率 P'_j 同样可以表示负载的大小, 同时改进后的概率还能表示每个节点的处理能力. 也就是说, 通过将负载信息与资源信息进行综合考虑, 概率 P'_j 更有利于整个集群的负载均衡, 更能提高任务的吞吐量. 例如, 设 4 个节点整体资源分别为 5, 6, 7, 8, 负载资源依次为 3, 2, 1, 5, 可用资源依次为 2, 4, 6, 3. 通过上述公式得到的概率 P_j 依次为 8/33, 9/33, 10/33, 6/33, 概率 P'_j 为 2/15, 4/15, 6/15, 3/15. 可以看出, 概率 P'_j 平衡了负载与节点的资源, 任务分配更合理.

算法 3 对于集群中的每个节点(第 1 行), 首先筛选满足任务要求的可用节点集合(第 2 行、第 3 行), 同时计算得到所有的可用资源总和(*Ava_total*)(第 4 行), 之后依据公式(4)和公式(5)计算每个节点的概率(第 5 行、第 6 行), 再根据节点的概率进行随机(第 7 行), 最后根据随机函数得到调度的节点, 节点处理能力越强, 任务调度到其上的概率也就越大, 最后返回任务和节点(*node_id*)的键值对(第 8 行).

算法 3. Resource-Pick_kx 算法.

输入: 集群节点 *RAY_nodes*, 调度任务 *task*.

输出: 调度结果 *task*: *node_id*.

算法描述:

```
1:  for each node in RAY_nodes do
2:    if task.dem≤RAY_AVA(node) then
3:      add node in RAY_ava_nodes
4:      Ava_total+=RAY_AVA(node)
5:  for each node in RAY_ava_nodes do
6:    compute RAY_AVA(node)/Ava_total
7:  Probabilistic randomness
8:  return task: node_id
```

Resource-Pick_kx 算法需要集群中存在可用资源, 即 Spillover 算法中的 *RAY_ava_nodes* 集合不能为空. 但当负载增大到超过整体资源时, *RAY_ava_nodes* 集合为空, 则节点概率降为 0, 无法对任务进行调度. 因此, 不

仅需要关注不同节点的处理能力和负载大小, 还要监控资源的状况, 三者缺一不可. 为满足上述要求, 将加权轮询思想与 Ray 中的 3 种资源结合, 得到基于 Ray 的平滑加权轮询(smooth weighted round robin)算法.

首先, 初始化节点权重. Ray 面向的是计算密集型的机器学习任务, 因此在指定节点的初始权重时, CPU 是必不可少的资源信息; 其次, 根据集群的物理资源和任务特点, 可以包括 GPU 和内存等资源信息, 上述资源信息可在整体资源 T 中获取. 因此, 根据经验初始权重设置为公式(6).

$$W_{ini}=0.9(\alpha CPU+\beta GPU)+0.1MEM \tag{6}$$

其中, α和β可以根据任务类型自行设定, 且始终保持$\alpha+\beta=1$.

在后续调度过程中, 权重随着调度而改变, 具体为: 当前权重为上次权重加初始权重, 第 1 次当前权重为初始权重值, 并挑选当前权重最高的为目标节点. 调度确定后, 目标节点需减去所有节点的权重. 计算公式如下.

$$W_{cur\_i}=W_{last}+W_{ini} \tag{7}$$

$$T_{total}=\sum_{i=1}^{n}W_{\mathrm{ini}} \tag{8}$$

$$W_{target}=\max(W_{cur\_1},\ldots,W_{cur\_n})-T_{total} \tag{9}$$

例如, 在某次迭代中, 4 个节点的上次权重依次为 3.5, 2.4, 6.2, 5.8, 而初始权重依次为 4.7, 5.0, 3.9, 6.2, 则此次权重累加后依次为 8.2, 7.4, 10.1, 12, 则目标节点为第 4 个节点, 最后的权重依次为 8.2, 7.4, 10.1 和−7.8.

算法 4 遍历可用资源满足的节点集合(第 1 行),依照公式(7), 对节点的当前权重进行更新(第 2 行), 记录目前的最大权重节点(第 3 行), 根据公式(8)累加权重和(第 4 行). 遍历结束, 得到权重最大的节点即为目标节点, 之后, 根据公式(9)修改目标节点的权重大小(第 5 行), 减掉权重和, 重新将权重和更新为 0(第 6 行),返回任务和目标节点的键值对, 算法结束(第 7 行).类似地,对于整体资源满足的节点,将 *RAY_ava_nodes* 更换为 *RAY_total_nodes* 即可.

算法 4. Smooth Weighted Round Robin 算法.

输入: 集群节点 *RAY_nodes*, 调度任务 *task*, 初始权重 *T_res*.

输出: 调度结果 *task*: *node_id*.

算法描述:

```
1:  for each node in RAY_ava_nodes do
2      cur_weights[node]+=T_res[node]
3:     node_id=max_cur_weights_node
4:     total_weights+=T_res[node]
5:  cur_weights[node_id]−=total_weights
6:  total_weights = 0
7:  return task: node_id
```

3.3 纵向扩容机制

对于在上述调度策略下无法满足的任务, 需要对集群进行纵向扩容, 增加节点资源来完成调度, 并与横向扩容共同降低集群负载.

框架的自动扩容原理如以下公式所示, 其目标是得到当前负载下集群的理想节点数目.

$$N_1=ceil[cn/f] \tag{10}$$

$$N_2=ceil[cd/cpn] \tag{11}$$

$$N=\max(N_1,N_2)=H_n+V_n+n \tag{12}$$

以 CPU 计算资源为例, 首先计算当下使用的节点数 *cn* 除以目标节点利用率 *f*(公式(10)), 以保持节点利用率始终保持在一个合理的水平; 然后计算目标 CPU 核数 *cd* 除以每个节点默认的 CPU 核数 *cpn*, 向上取整(公式(11))得到目标节点数. 两者中取最大值得到理想的节点数目, 其值为纵向扩容的节点数 V_n 和横向扩容的节

点数 H_n 与现有节点数 n 的和(公式(12)). 同时, 纵向扩容的执行优先于横向扩容, 并且无论现有节点是否等于理想节点, 纵向扩容一旦触发都会执行.

整个扩容基于容器, 容器即为 Ray 中的节点. 本文在 Ray head 节点中设置的监控模块(monitor)来管理扩容过程, 其中的弹性伸缩功能(autosaler)具体负责扩容的执行. 纵向扩容过程如图 3 所示, *task* 在节点 I 提交, *task* 的资源要求为 *resource* II, 并且 *resource* II>*resource* I.

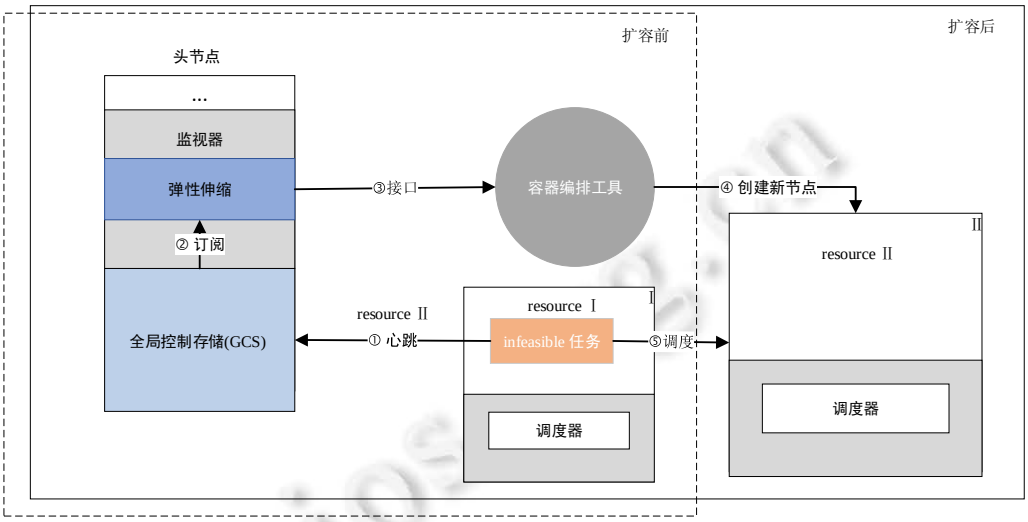


图 3 纵向扩容

整个纵向扩容机制由 5 部分组成.

(1) 当集群节点无法满足任务的资源要求时, 任务转化为 infeasible. 节点 I 要求的资源 *Resource* II 借助 Heartbeat 传递到 Head node 的 GCS 模块.

(2) Head node 中的 monitor 通过订阅心跳来监控集群状况. 心跳是节点同 GCS 交互集群信息的方法, monitor 每隔一个心跳时间接受 GCS 传来的信息, 其中包括 infeasible 任务的资源要求 *resource* II, 随后纵向扩容被触发, autoscaler 接管此部分.

(3) 拿到资源要求 *resource* II 后, autoscaler 自定义节点类型加入到队列中, 逻辑如算法 5 所示.

(4) 容器编排工具接收到创建新节点的指令后, 在队列中读取节点信息, 新建节点并加入到 Ray 集群中, 节点创建完成后, 则将相应节点类型出队.

(5) Head node 会检测到新加入的节点, 唤醒 infeasible 任务转化为可用状态再次调度. 节点 II 满足 *task* 的资源要求, 所以任务被调度到此节点; 若依旧不满足, 则任务再次转化为 infeasible 状态.

在纵向扩容算法中, 根据每一个 infeasible 任务的资源信息(第 1 行)设定节点类型 *node_type*(第 2 行), 如果队列包含节点类型, 则继续迭代查找下一个 infeasible 任务(第 3 行、第 4 行), 否则将节点类型入到队列中(第 5 行、第 6 行). 通过设置队列, 缓冲节点的创建时间, 防止一个 infeasible 任务创建过多的节点.

算法 5. Autoscale 纵向扩容算法.

输入: infeasible 任务要求的资源信息 *Res_inf*.

输出: 扩容队列 *q*.

算法描述:

```
1:  for each res in Res_inf do
2:      res→node_type
3:      if q.contains(node_type)
4:          continue
```

5: **else**
6: put *node_type* in *q*
7: **return** *q*

4 实 验

为验证本文提出的调度框架的高效性和可用性, 本文从任务吞吐、运行时间、扩容效果等方面进行了对比实验.

4.1 实验环境

本实验搭建集群测试框架整体效果和扩容效果. 实验基于 Python3.6.8 和 Ray 1.0.0rc2 版本. 任务调度方面, 使用 C++完成算法撰写, 使用 Python 作为测试编程语言. 框架整体效果采用 Ray 自带的 Microbenchmark (1.0.0rc2/python/ray/ray_perf.py)、TPC-DS (http://www.tpc.org/)大数据基准测试. 扩容对比实验皆采用 Python 作为编程语言, 未扩容的默认环境为一个 head 节点和一个工作节点. 配置参数见表 1.

表 1 集群配置

配置项	参数
操作系统	CentOS Linux release 7.9.2009
处理器型号	Intel(R) Xeon(R) Silver 4110 CPU@2.10 GHz
处理器个数	2
处理器核数	8
内存(GB)	250
Kubernetes 版本	v1.15.1
Docker 版本	20.10.1
基础镜像	ubuntu: focal

4.2 实验数据

在实验使用的数据方面, 主要集中在 TPC-DS 测试, 其余实验主要是 Ray 基本的异步任务, 数据可直接放进程序, 无须读入. 对于 TPC-DS_2 测试, 使用 TPC-DS_2 工具生成数据, 得到 25 张表. 通过设定不同的参数, 可以生成不同规模的数据. 查询操作所用数据为 store.dat, 连接操作所用数据为 store.dat 和 store_sales.dat.

4.3 实验结果和分析

在整体效果对比实验中, 每种测试分别使用 Ray 默认的调度策略、轮询算法、Pick-kx 算法和本文提出框架中的 resource-Pick_kx 算法(RPK)以及平滑加权轮询算法(SRR)进行对比.

首先, 使用 Ray 官方的 Microbenchmark 来测试任务的吞吐量. 同步任务具体任务为 ray.put 和 ray.get, 任务资源要求使用 Ray 默认配置. 同步执行后, 实验结果见表 2: resource-Pick_kx 算法效果最好, 但同步任务并不能充分发挥分布式框架的效果; 异步任务具体任务与同步任务相似, 但通过循环异步执行.

表 2 Microbenchmark 同步任务

算法	吞吐量(s)
Ray1.0.0	997
轮询	996
Pick-kx	1 012
RPK	1 115
SRR	1 080

如图 4 所示, 并行度依次取 1 000, 10 000 和 100 000, 整体吞吐量逐渐增加, 但平滑加权轮询算法始终略高一筹, 性能提升 20%左右; 但随着设置的并行度越来越大, 集群资源全部被占用, 所以吞吐量趋于饱和. 在整个 microbenmark 测试过程中, 轮询算法与 Ray 原生的随机算法在实验效果上相差无几, 这在理论上也是相符的.而 Pick-kx 效果并不理想, 甚至有的任务吞吐量比随机算法更低, 而且不稳定. 究其原因, 可能是由于用户提交的任务资源要求与实际任务所用资源并不一致, 只通过负载信息判断可能会出现判断失误的情况.

在 Microbenchmark 中，测试了基本的同步和异步任务，各种算法的优劣势得到了展现. 但任务仅仅执行简单的返回操作，且资源要求一致. 为了进一步对比本文提出的框架和 Ray 原生的调度算法，设计补充了测试用例，设置多种异构资源要求的任务，使任务完成多种算术逻辑运算. 具体任务为通过循环异步执行加减乘除算数运算. 实验结果如图 5 所示，随着并行度的提升，完成的任务数逐渐增多，且前期并行度在 10–300 阶段，resource-Pick_kx 更有优势；而并行度在 300–700 阶段，平滑加权轮询更高效持久，提升效果在 10%左右. 可以看出，由于前期集群中可用资源充裕，resource-Pick_kx 算法可以充分发挥作用；随着任务的增多，可用资源消耗殆尽而整体资源满足时，平滑加权轮询效果更好.

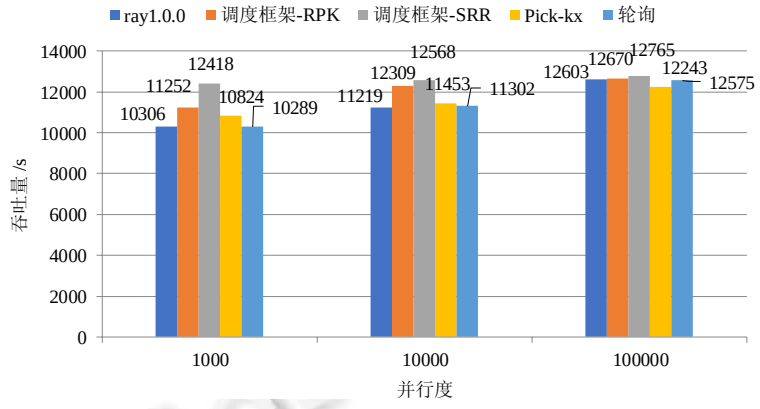


图 4 Microbenchmark 异步任务

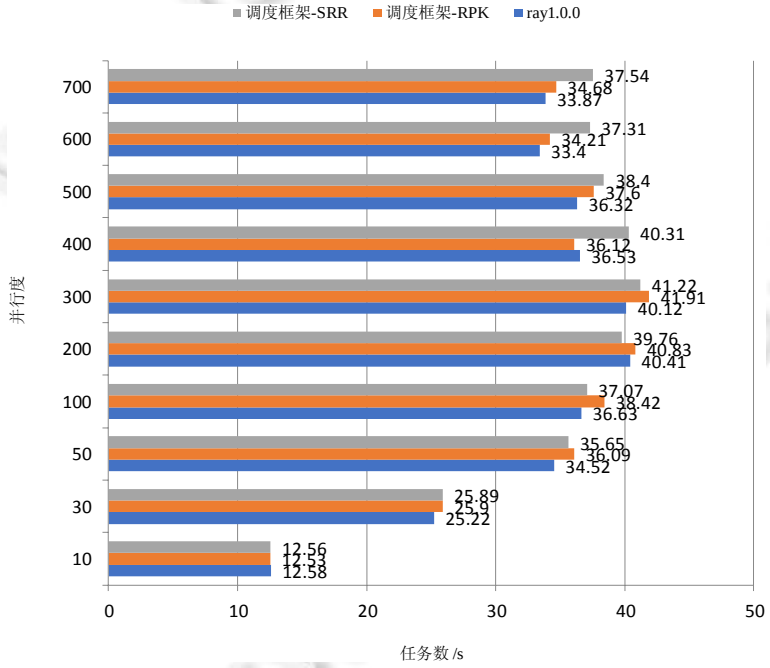
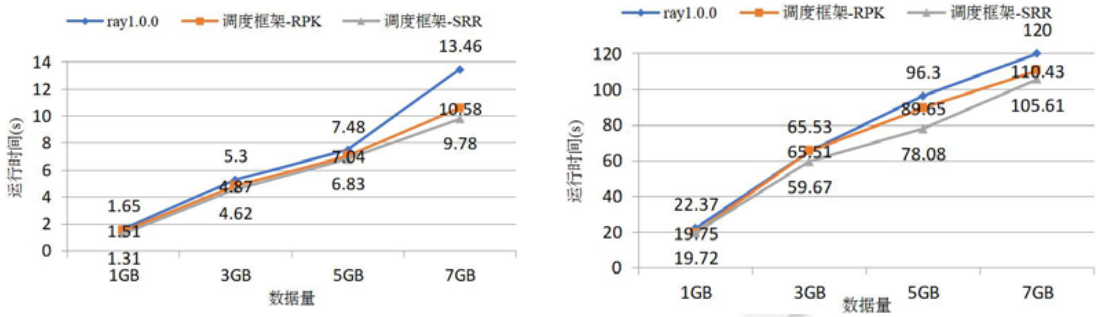


图 5 异构任务

此外，本文还使用 TPC-DS 生成不同规模的数据集进行测试. 如图 6(a)、图 6(b)所示，具体任务为查询连接操作，测试其运行时间. 数据集规模分别为 1 GB, 3 GB, 5 GB 和 7 GB. 图 6(a)执行查询操作，图 6(b)执行连接操作.可看出，随着数据量的增大，random 算法始终运行时间最长；其次为 resource-Pick_kx 算法，平滑加权轮询时间最短.

另外, 除了对吞吐量的实验对比以外, 本文还设计了负载均衡的实验, 通过执行不同数量级的 CPU 密集型任务, 统计集群中各节点的 CPU 使用率, 来对比各框架的负载均衡效果. 具体执行任务数为 100, 1 000 和 10 000, 对比结果见表 3.



(a) 查询运行时间 (b) 连接运行时间

图 6 不同规模数据集运行时间比较

表 3 负载 CPU 使用率对比

Task	Node	Ray1.0.0 (%)	调度框架-RPK (%)	调度框架-SRR (%)
1 000	1	90	91	89
	2	22	23	22
	3	15	17	17
10 000	1	275	276	275
	2	90	88	87
	3	75	80	78
100 000	1	419	417	416
	2	132	130	129
	3	122	123	124

在构成集群的 3 个节点中, 由于本地优先原则, 提交任务的头节点 *Node*1 的 CPU 使用率最大, 其余节点相对较小. 在 Ray1.0 中, *node*2 和 *node*3 中, CPU 使用率差距最大可为 15%, 在调度框架 RPK 和调度框架 SRR 中为 5%–9%. 对比来看, 相比于原始的 Ray1.0.0, 框架 SRR 和框架-RPK 的比例更均衡, 即其任务分配更“平均”,负载均衡效果好. 另外, 本文通过提供不同资源要求的任务来检验纵向扩容功能, 具体任务为返回固定字符串. 任务要求见表 4.

表 4 任务资源要求参数

Task	CPU	Mem (MB)
*Task*1	1	512
*Task*2	2	512
*Task*3	2	1 024

初始, Kubernetes 集群只有 *Pod*1 类型的节点 4 个, 即只满足 *Task*1 的调度执行, 其中, *Pod* 与 *Task* 相对应. 测试过程是: 提交若干 *Task*1、2 个 *Task*2 和 1 个 *Task*3. 当表 4 的 3 种任务都已提交, 经过纵向扩容后, 集群组成结构如图 7 所示. 纵向扩容了 2 个 *Pod*2 和 1 个 *Pod*3, 使得所有 *task* 都被调度执行.

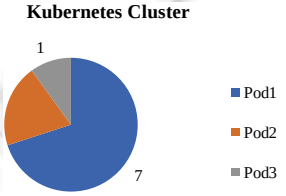


图 7 Kubernetes 集群

上述任务执行情况见表 5. 可以看出, 在 Ray1.0.0 横向扩容的情况下, *Task*1 可以正常执行完成, *Task*2 和

*Task*3 未完成；而在本文提出的调度框架下横向+纵向扩容，完成了所有类型任务的执行．其中，心跳时间 100 ms，扩容触发 2.3 ms 左右，所以整个扩容响应时间可以等同于心跳时间．本实验所用 Kubernetes 扩容一个 *Pod* 时间约为 10 s．可以看出，自动扩容功能更适用于长作业和特殊资源要求的作业．

表 5 框架扩容状况对比

Task	Ray1.0.0			调度框架		
	任务执行	扩容触发(ms)	Kubernetes 扩容(s)	任务执行	扩容触发(ms)	Kubernetes 扩容(s)
*Task*1	完成,24.75s	2.37	31.02	完成,24.48s	2.38	30.53
*Task*2	未完成	–	–	完成,20.40s	2.29	21.34
*Task*3	未完成	–	–	完成,17.93s	2.31	10.45

5 结束语

本文提出了一种新的任务调度扩容管理框架，包括两种任务调度算法和一个纵向扩容机制．实验表明：两种算法各有优势，与纵向扩容一起完整地解决了任务分配到哪个节点的问题，保证任务能高效执行．由于调度和扩容都基于 Ray 中的逻辑资源，不能完全代表实时的资源状况，所以在接下来的工作中，我们将从 Ray 的逻辑资源、扩容的完整性等角度继续研究．

References:

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In: Proc. of the 6th Conf. on Symp. on Opearting Systems Design & Implementation, Vol.6. San Francisco: USENIX Association, 2004. 259−272.

[2] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster computing with working sets. In: Nahum EM, Xu DY, eds. Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2010). USENIX Association, 2010.

[3] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 38(4): 28−38.

[4] Iqbal MH, Soomro TR. Big data analysis: Apache storm perspective. Int'l Journal of Computer Trends and Technology, 2015, 19(1): 9−14.

[5] Meng X, Bradley J, Yavuz B, Sparks E, Talwalkar A. Mllib: Machine learning in apache spark. The Journal of Machine Learning Research, 2016, 17(1): 1235−1241.

[6] Abadi M, Barham P, Chen J. TensorFlow: A system for large-scale machine learning. In: Proc. of the 12th Symp. on Operating Systems Design and Implementation. USENIX Association, 2016. 265−283.

[7] Moritz P, Nishihara R, Wang S. Ray: A distributed framework for emerging AI applications. In: Proc. of the 13th Symp. on Operating Systems Design and Implementation. USENIX Association, 2018. 561−577.

[8] Burns B, Grant B, Oppenheimer D. Borg, Omega, and Kubernetes. Communications of the ACM, 2016, 59(5): 50−57.

[9] Vavilapalli VK, Murthy AC, Douglas C. Apache hadoop yarn: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. ACM, 2013. 1−16.

[10] Hindman B, Konwinski A, Zaharia M. Mesos: A platform for fine-grained resource sharing in the data center. In: Proc. of the 8th USENIX Symp. on Networked Systems Design and Implementation. USENIX Association, 2011. 295−308.

[11] Peng B, Hosseini M, Hong Z. R-Storm: Resource-aware scheduling in Storm. In: Proc. of the Middleware Conf. ACM, 2015. 149−161.

[12] Chen YR, Lee CR. G-Storm: A GPU-aware storm scheduler. In: Proc. of the 2016 IEEE 14th Int'l Conf. on Dependable. Auckland: IEEE, 2016. 9−16.

[13] Gan GN, Huang TL, Gao S. Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In: Proc. of the Int'l Conf. on Intelligent Computing and Integrated Systems. IEEE, 2010. 60−63.

[14] Wang XM, Wang YH. Combination of simulated annealing algorithm and genetic algorithm. Chinese Journal of Computers, 1997(4): 381−384 (in Chinese with English abstract).

[15] Alsadie D, TSMGWO: Optimizing task schedule using multi-objectives grey wolf optimizer for cloud data centers. IEEE Access, 2021, 9: 37707−37725.

[16] Moulik S. RESET: A real-time scheduler for energy and temperature aware heterogeneous multi-core systems. Integration, 2021, 77: 59−69.

[17] Arshed JU, Ahmed M. RACE: Resource aware cost-efficient scheduler for cloud fog environment. IEEE Access, 2021, 9: 65688−65701.

[18] Cicerone S, Di Stefano G, Navarra A. “Semi-Asynchronous”: A new scheduler in distributed computing. IEEE Access, 2021, 9: 41540−41557.

[19] Karger D, Lehman E, Leighton T. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc. of the 29th Annual ACM Symp. on Theory of Computing. ACM, 1997. 654−663.

[20] Genova Z, Christensen KJ. Challenges in URL switching for implementing globally distributed Web sites. In: Proc. of the Int’l Workshop on Parallel Processing. IEEE, 2000. 89−94.

[21] Wang JW, Wang SP, Xu LB, Guo JJ, Yu CH. Optimization of load balancing algorithm based on weighted polling. Computer System Applications, 2018, 27(4): 138−144 (in Chinese with English abstract). [doi: 10.15888/j.cnki.csa.006284]

[22] Dahlin M. Interpreting stale load information. IEEE Trans. on Parallel and Distributed Systems, 2000, 11(10): 1033−1047.

[23] Chen Y, Huang JX. Elastic scaling strategy based on Kubernetes application. Computer Systems & Applications, 2019, 28(10): 213−218 (in Chinese with English abstract). [doi: 10.15888/j.cnki.csa.007106]

[24] Balla D, Simon C, Maliosz M. Adaptive scaling of Kubernetes pods. In: Proc. of the 2020 IEEE/IFIP Network Operations and Management Symp. (NOMS 2020). IEEE, 2020. 1−5. [doi: 10.1109/NOMS47738.2020.9110428]

[25] Giannakopoulos P, Petrakis EGM. Smilax: Statistical machine learning autoscaler agent for apache FLINK. In: Proc. of the Int’l Conf. on Advanced Information Networking and Applications. Springer, 2021. 433−444.

[26] Vasavi S, Gokhale AA. Framework for visualization of geospatial query processing by integrating Redis with Spark. Int’l Journal of Natural Computing Research, 2019, 8(3): 1−25.

附中文参考文献:

[14] 王雪梅, 王义和. 模拟退火算法与遗传算法的结合. 计算机学报, 1997(4): 381−384.

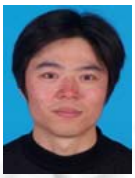
[21] 汪佳文, 王书培, 徐立波, 郭家军, 俞成海. 基于权重轮询负载均衡算法的优化. 计算机系统应用, 2018, 27(4): 138−144. [doi: 10.15888/j.cnki.csa.006284]

[23] 陈雁, 黄嘉鑫. 基于 Kubernetes 应用的弹性伸缩策略. 计算机系统应用, 2019, 28(10): 213−218. [doi: 10.15888/j.cnki.csa.007106]

刘瑞奇(1996－), 女, 硕士生, CCF 学生会员, 主要研究领域为分布式计算.

赵恒泰(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为图计算, 分布式计算.

李博扬(1992－), 男, 博士, CCF 学生会员, 主要研究领域为分布式数据分析, 机器学习.



金福生(1977－), 男, 博士, 副教授, CCF 高级会员, 主要研究领域为大数据, 区块链, 人工智能.

高玉金(1974－), 男, 博士, 讲师, 主要研究领域为计算机体系结构, 片上网络系统.

李荣华(1985－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为图数据管理与挖掘, 图计算系统.

李长升(1985－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为机器学习, 计算机视觉.

王国仁(1966－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为不确定数据管理, 数据密集型计算, 可视媒体数据分析管理, 非结构化数据管理, 分布式查询处理与优化, 生物信息学.