应用感知的容器资源调度优化方法①

陆志刚 [1,2,3], 吴悦文 [1,2], 顾泽宇 [1,2], 吴启德 [4]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100049)
[3](苏州工业园区国有资产控股发展有限公司, 苏州 215028)
[4](中国科学技术大学 计算机科学学院, 合肥 230026)

摘 要: 资源调度作为容器管理的关键技术之一, 已有研究工作或满足公平性目标, 将工作负载平均调度到所有物理节点上, 关注吞吐率指标; 或满足性能目标, 将工作负载关联的多个容器载体调度到相同或相近的物理节点上, 关注响应时间. 提出应用感知的容器资源调度方法, 采用多队列模型兼顾公平性和性能目标. 实验结果显示, 对于典型的大数据处理场景, 本方法和已有公平性调度方法具有相当的吞吐率; 对于典型的事务型应用场景, 本方法相对于已有的公平性调度方法, 事务型应用的延迟最多可减少 100%.

关键词: 应用感知; 容器; 资源调度; 大数据; 事务型

Application-Aware Container-Oriented Resource Scheduling Optimized Approach

LU Zhi-Gang[1,2,3], WU Yue-Wen[1,2], GU Ze-Yu[1,2], WU Qi-De[4]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)
[3](SIP State Property Holding Co. Ltd., Suzhou 215028, China)
[4](University of Science and Technology of China, Hefei 230026, China)

Abstract: Resource scheduling is a key technique for container management. Prior work or meets the goal of fairness, the work load average is scheduled to all physical nodes, pays attention to the throughput indicator; or to meets performance targets, multiple carrier scheduling the workload related to physical nodes in the same or similar, pay attention to the response time. In this paper, it presents an application-aware resource scheduling approach, and employs a multi-queue model to meet both fairness and performance targets. Experimental results show that the approach has equal throughput for typical big data processing scenarios, and can reduce latency by up to 100% for typical transactional application scenarios.

Key words: application-aware; container; resource scheduling; big data; transaction-based web

容器是一种新型虚拟化技术, 其核心思想是操作系统复用, 模拟沙箱进程环境, 因而具有接近物理主机性能、资源利用率高的特性[1]. 近年来, 容器已在国内外产业界得到广泛应用, 比如 Docker Cloud[2], 羊年春晚微信抢红包[3], 京东 618 商品秒杀[4]等. 另具 Gartner 研究报告[5], 全球约 70%的应用将于 2019 年迁移到容器基础设施上. 因此, 容器正逐渐成为应用运行支撑的主流基础设施.

资源调度是大规模容器管理的核心组件之一, 也是学术界和工业界的关注重点, 如 Google Kubernetes[7]、Microsoft Apollo[8]、Apache Mesos[9]. 资源调度是指通过合适的调度策略或算法, 将容器合理的分配到不同的物理机器上, 以在满足特定约束条件的情况下提高资源利用率, 节省运营成本.

已有研究考虑公平性因素, 适用大数据处理应用, 其核心思想是将封装大数据处理应用的容器均分到物理服务器上, 优化目标是吞吐率[10,11]. 另有研究关注性能因素, 适合事务类应用, 其核心思想是将封装了

① 收稿时间:2016-06-16;收到修改稿时间:2016-07-19 [doi:10.15888/j.cnki.csa.005621]

© 中国科学院软件研究所 http://www.c-s-a.org.cn

事务类应用的容器尽量部署在相同的物理服务器上，优化目标是延迟[12,13]. 由此可见，已有研究通常在运行时只能满足单一优化目标，难以根据应用类型选择最优的调度算法.

本文提出应用感知的容器资源调度方法，该方法采用多队列模型，每个队列采用单一调度策略，并可根据应用类型，选择合适的调度策略，从而达到兼顾公平性和性能目标. 实验结果显示，对于典型的大数据处理场景，本方法和已有公平性调度方法具有相当的吞吐率；对于典型的事务型应用场景，本方法相对于已有的公平性调度方法，事务型应用的延迟最多可减少 100%. 例如文献[12]选取事务型应用 TPC-W 作为测试基准，采用延迟优化的调度算法，在并发 500 的场景下，应用平均响应时间从 507ms 下降到 237ms.

1 系统概述

1.1 体系结构

如图 1 所示，整个体系结构涉及应用层、资源调度层和物理资源层，其中资源调度层由应用类型声明器、数据监测器、调度决策器、应用放置器四个部分.

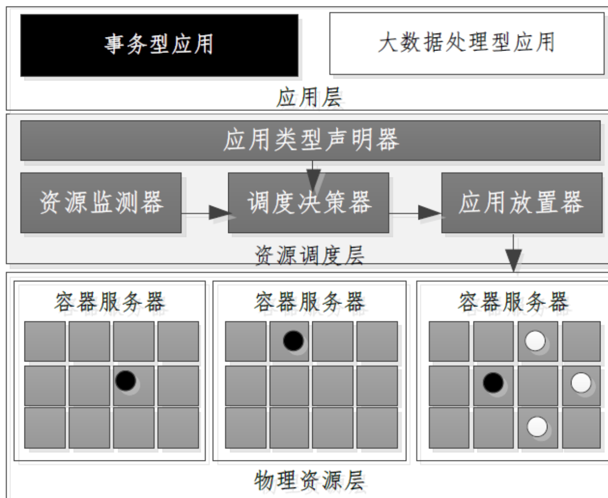

图 1 体系结构

应用声明器：用户在部署应用的时候，需要显示声明应用的类型是事务型，还是大数据处理型，见 1.2 节；

资源监测器：用于监测物理资源的使用率，基于开源监测软件 Zabbix 实现；

调度决策器：考虑资源空闲和应用类型两个维度因素，采用多队列模型进行刻画，并实现相关的资源调度算法，见第 2 节；

应用放置器：依据调度决策器输出结果，调用容器暴露 API 进行应用放置，构建应用与物理资源的映射关系.

1.2 应用声明器

1.2.1 手动设定

本文应用 App 由向三元组刻画应用<ID,TYPE, ComponentSet>，其中 ID 用于标识是应用的唯一标识；TYPE 表示应用的类型，在本文中用于描述事务型或者大数据处理型；ComponentSet 用于描述应用的组件组成，比如事务型应用通常由 Web 服务器、应用服务器和数据库服务器三部分组成；大数据处理型应用通常包括任务调度和分发器(如 Hadoop 的 Master 节点)、任务执行器(如 Hadoop 的 Map 和 Reduce 节点)和中间数据存储(如 Hadoop 的 HDFS)三个部分.

ComponentSet 采用有向无环图进行刻画，用于描述应用组件的规模，以及各个组件的资源需求. 其中，点代表应用组件实例，点的属性包括资源需求，由二元组<CPU、MEM>组成，边代表应用组件实例之间的调用关系，比如对于典型事务型应用 TPC-W，点分别代表 Web 服务器、应用服务器和数据库服务器，边代表 Web 服务器与应用服务器调用关系，或者应用服务器与数据库服务器的调用关系，如图 2 所示.

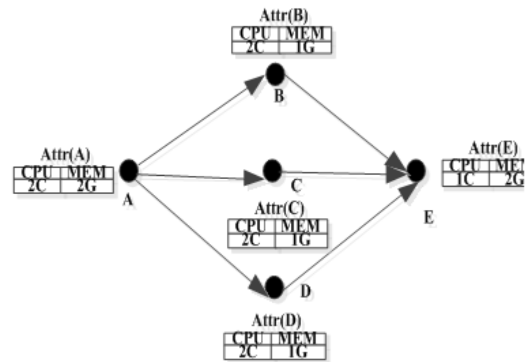

图 2 体系结构

1.2.2 自动识别

相关研究工作尝试自动识别并挖掘应用构成，这些方法通常采用代理机制，适用于离线测试阶段，其核心思想是将 Agent 部署在容器中，监测和记录网络交互信息，采用 PCA 等算法过滤噪音数据，基于图模型分析获取容器(及其应用组件)的关联关系.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

2　各模块的算法设计与实现

2.1 基于多队列资源调度机制

如图 3 所示, 资源调度器由多队列模型组成, 分别实现公平性资源调度算法和延迟敏感调度算法, 以满足公平性或性能目标, 其具体流程:

① 根据应用声明<ID,TYPE,ComponentSet>实例化队列, 其中队列的长度为应用组件的规模, 队列中每个元素刻画该应用组件的资源需求;

② 如果应用为事务型, 调用公平性调度算法, 最终通过容器 API, 构建应用组件与容器服务器的映射关系, 见 2.2 节;

③ 如果应用为大数据处理型, 调用延迟敏感调度算法, 最终通过容器 API, 构建应用组件与容器服务器的映射关系, 见 2.3 节.

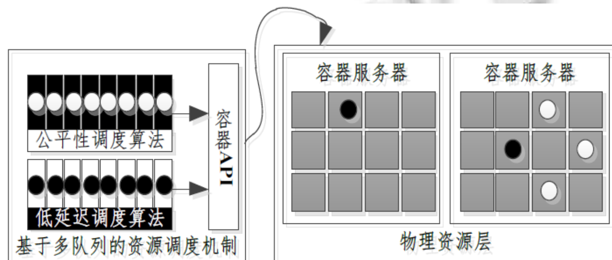

图 3　基于多队列资源调度机制

2.2 公平调度算法 SAF

公平性资源调度算法的核心思想是均衡, 将容器服务器根据资源的空闲状态进行排序, 将应用中每个组件依次进行调度, 每次调度将应用组件部署在资源空闲的容器服务器上.

算法 1. 公平调度算法 SAF

输入: A set of physical machine PMSet;

　　A specified application *App*.

输出: The relationship between componenets and PMSet f(*componenet*, PMSet), where $componenet_i \in App$.

1. **For** $componenet_i$ **in** *App*
2. 　　PMSet= **DescendingSort(**PMSet**)**
3. 　　PM_j = PMSet.remove(0)
4. 　　PM = PM − Attr($componenet_i$)
5. 　　PMSet.add(PM_j)
6. 　　Add $f(componenet_i, PM_j)$
7. **End For**

算法 1 为公平调度算法 SAF(Scheduling algorithm based on fairness)实现, 遍历应用包含的所有组件 $componenet_i$, 然后根据容器物理主机的资源空闲进行降序排列, 则第一个元素即为最为空闲的容器主机(第 1 行-2 行). 将应用组件调度到资源最后空闲的容器主机上, 同时更新该容器主机的空闲资源状态, 并记录下此时应用组件和容器服务器的关联关系(第 3 行-6 行).

2.3 公平调度算法 SAL

公平调度算法 SAL(Scheduling algorithm based on latency-aware)是一种延迟敏感的资源调度算法, 其核心思想是优先将应用组件调度到相同的容器物理机上, 即最大资源利用率每台容器物理服务器.

算法 2 为公平调度算法 SAL 实现. 首先根据容器物理主机的资源空闲进行降序排列, 则第一个元素即为最为空闲的容器主机(第 1 行-2 行). 遍历应用包含的所有组件, 优先调度到相同的容器物理服务器上, 直到该容器服务器资源饱和, 此过程中记录下应用组件和容器服务器的关联关系(第 3 行-5 行, 第 9 行). 如果容器服务器资源饱和, 且此时还有应用组件未被调度(6 行-8 行), 则重新根据容器物理主机的资源空闲进行降序排列, 选择空闲的容器服务器接着调度, 直到应用组件全部调度完成为止.

算法 2. 公平调度算法 SAL

输入: A set of physical machine PMSet;

　　A specified application *App*.

输出: The relationship between componenets and PMSet f(*componenet*, PMSet), where $componenet_i \in App$.

1. PMSet = **DescendingSort(**PMSet**)**
2. PM_j = PMSet.remove(0)
3. **While** $PM_j > 0$
4. 　　**For** $componenet_i$ **in** *App*
5. 　　　　PM = PM − Attr($componenet_i$)
6. 　　　　If(PM_j<0)
7. 　　　　　　PM = PM + Attr($componenet_i$)
8. 　　　　　　Break;
9. 　　　　Add $f(componenet_i, PM_j)$
10. 　　**End For**
11. 　　PMSet.add(PM_j)
12. 　　PMSet = **DescendingSort(**PMSet**)**
13. 　　PM_j = PMSet.remove(0)

© 中国科学院软件研究所　http://www.c-s-a.org.cn

14. **End While**

3 实验与验证

实验包括三个部分, 首先是实验环境的介绍, 接着对比 SAF 和 SAL 算法在事务型应用场景下, 性能的差异性; 最后对比两种算法在大数据处理应用场景下, 吞吐率的差异性. 从而验证本方法的有效性. 其中 SAF 算法本质是公平调度, 可反映 Kubernetes、Omage 等系统的资源调度效果; SAL 算法本质为延迟敏感, 可反映 Apollo、Nomad 和 DelaySchedule 的资源调度效果.

4.1 实验环境与负载

如图 4 所示, 实验环境由 21 台刀片机组成, 每台刀片的配置都是 16 cores, 2.4GHz Intel Xeon CPU 和 24G 内存: 其中 10 台刀片作为 HDFS 服务器; 10 台刀片安装容器软件, 用于部署容器, 1 台刀片作为负载发生器, 模拟用户进行压力测试, 1 台刀片安装本文所述应用敏感的资源调度器. 所有刀片设备之间通过千兆交换机互联.

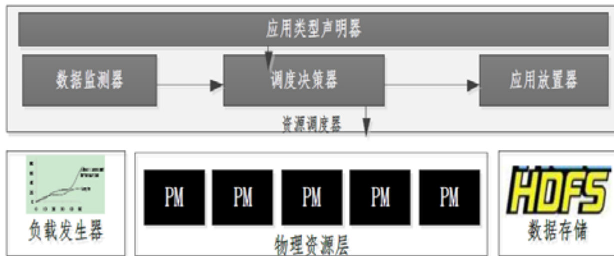

图 4 实验环境

事务型应用采用 TPC-W 基准测试, 前端 Web 服务器组件选用 HttpServer 2.0, 中间应用服务器组件 tomcat 8.0, 后端数据库服务器组件选用 Mysql 5.6, 数据库采用默认设置, 即 10 000 件商品和 1 440 000 个用户. 上述应用组件均部署在 2CPU 和 2GB 内存的容器中. 其中, 工作负载来源于 1998 年法国世界杯网站, 分别模拟 50、175、250 和 325 并发用户, 图 5 所示.

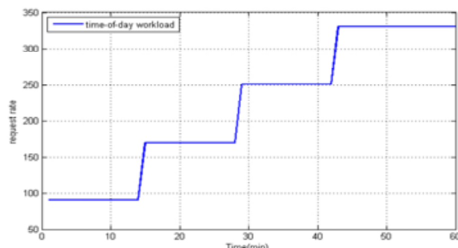

图 5 事物型负载

大数据处理应用采用 WordCount 和 Sort 两类基准测试, 数据存在 HDFS 中, 前者数据总大小为 10GB, 后者数据总大小为 1GB, 且应用组件均部署在 2CPU 和 2GB 内存的容器中. 表 1 给出了相关的配置.

表 1 Hadoop 基础测试配置表

类型	#Map	#Reduce
WordCount	5	1
	15	1
	30	1
Sort	5	1
	15	1
	30	1

4.2 事务型应用场景下本方法有效性

本方法调度事务型应用时会采用 SAL 算法, 而 Kubernetes、Omage 等系统会默认采用 SAF 算法. 因此, 本方法与 Kubernetes、Omage 等系统资源调度算法对比本质可转换为事务型应用在两种调度算法下的性能(响应时间刻画)差异如图 6 所示.

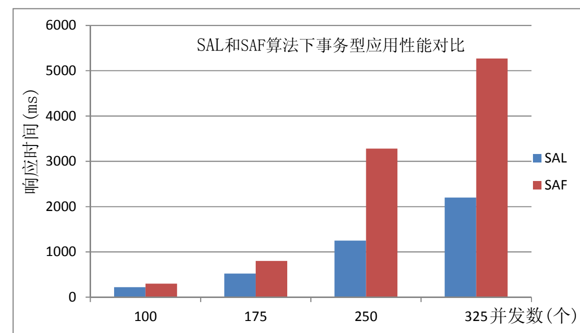

图 6 SAL 和 SAF 算法下事务型用性能对比

随着并发用户的增加, 事务型应用在 SAF 算法和 SAL 算法下的响应时间的比例差异越大, 最大可达到近 3 倍. 这是因为响应时间度量指标是指用户请求经过 TPC-W 测试基准 Web 服务器、应用服务器、数据库服务器处理, 最终返回用户的总延迟时间总和. 采用 SAL 算法, 由于应用组件调度在相同容器物理服务上, 其延迟时间仅仅为 SAF 算法跨主机的网络延迟是少 1/4.

4.3 大数据处理应用场景下本方法有效性

本方法调度事务型应用时会采用 SAF 算法, 而 DelaySchedule 等系统会默认采用 SAL 算法. 因此, 本方法与 DelaySchedule 等系统资源调度算法对比本质可转换为事务型应用在两种调度算法下的性能(吞吐

© 中国科学院软件研究所 http://www.c-s-a.org.cn

率刻画)差异如图 7 所示.

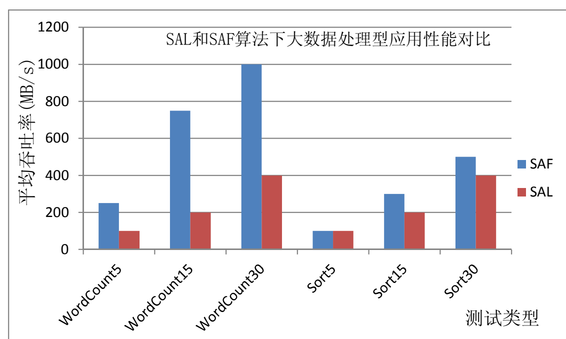

图 7　SAL 和 SAF 算法下事务型应用性能对比

无论是 WordCount 测试基准, 还时 Sort 测试基准, 随着 Map 实例数增加, Map 阶段在 SAF 算法和 SAL 算法下的吞吐率的比例差异越大, 其中 WordCount 非常明显, 最差可相差 2.5 倍. 这是因为实验环境全部是千兆网络, SAL 算法会将应用组件优先调度到相同容器物理服务器上, 而单台容器服务器理论网络吞吐率上限为 125MB/s, 会 Hadoop 应用实例会因为网络资源竞争而成为瓶颈. 仔细分析实验数据, 单个 WordCount 的 Map 阶段吞吐率约为 50MB/s, Sort 约为 20MB/s. 在 SAL 算法下, 由于 Hadoop 应用每个实例的资源配置为 2CPU 和 2GB 内存, 即每台容器服务器可部署 7 个 WordCount 实例和 Sort 实例, 故导致资源瓶颈.

4　结语

资源调度作为容器管理的关键技术之一, 已有研究工作难以同时适用大数据处理应用和事务型应用场景. 本文提出应用感知的容器资源调度方法, 其核心思想是采用多队列模型, 兼顾两种场景. 实验结果显示本方法具有有效性. 本文下一步工作拟开展基于学习的应用类型自动识别技术, 进一步提高本方法的实用性.

参考文献

1 Abraham L, Allen J, Barykin O, et al. Scuba: Diving into data at facebook. Proc. of the Vldb Endowment, 2013, 6(11): 1057–1067.

2 Ananthanarayanan G, Agarwal S, Kandula S, et al. Scarlett: Coping with skewed content popularity in MapReduce clusters. In EuroSys, 2011: 287–300.

3 Akidau T, Balikov A, Bekiro, et al. MillWheel: Fault-tolerant stream processing at internet scale. Proc. of the Vldb Endowment, 2013, 6(11): 1033–1044.

4 Andersen DG, Franklin J, Kaminsky M, et al. FAWN: A fast array of wimpy nodes. Communications of the ACM, 2011, 54(7): 101–109.

5 Petrucci V, Laurenzano M, Doherty J, et al. Octopus-man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). IEEE Computer Society. 2015. 246–258.

6 Aksanli B, Venkatesh J, Zhang L, et al. Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. Association for Computing Machinery, 2011: 53–57.

7 Marchukov M, Song YJ, Bronson N, et al. TAO: Facebook's distributed data store for the social graph. Proc. of the 2013 USENIX Conference on Annual Technical Conference. USENIX Association. 2013. 49–60.

8 Baker J, Bond C, Corbett J, et al. Megastore: Providing scalable, highly available storage for interactive services. 5th Biennial Conference on Innovative Data Systems Research (CIDR '11). Asilomar, California, USA. 2011. 223–234.

9 Baumann A, Barham P, Dagand PE, et al. The multikernel: A new OS architecture for scalable multicore systems. ACM Sigops, Symposium on Operating Systems Principles. ACM. 2009. 29–44.

10 Belay A, Bittau A, Mashtizadeh A, et al. Dune: Safe user-level access to privileged CPU features. Symposium on Operating Systems Design & Implementation (OSDI). 2012. 335–348.

11 Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: Flexible, scalable schedulers for large compute clusters. ACM European Conference on Computer Systems. ACM. 2013. 351–364.

12 Tune E. Large-scale cluster management at Google with Borg. Proc. of the 10th European Conference on Computer Systems. ACM. 2015. 18.

© 中国科学院软件研究所 http://www.c-s-a.org.cn