软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(12):4704–4726 [doi: 10.13328/j.cnki.jos.006356]

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

面向 Dataflow 的异构集群混合式资源调度框架研究*

汤小春, 赵 全, 符 莹, 朱紫钰, 丁 朝, 胡小雪, 李战怀

(西北工业大学 计算机学院, 陕西 西安 710129)

通信作者: 汤小春, E-mail: tangxc@nwpu.edu.cn

摘 要: Dataflow 模型的使用, 使得大数据计算的批处理和流处理融合为一体. 但是, 现有的针对大数据计算的集群资源调度框架, 要么面向流处理, 要么面向批处理, 不适合批处理与流处理作业共享集群资源的需求. 另外, GPU 用于大数据分析计算时, 由于缺乏有效的 CPU-GPU 资源解耦方式, 降低了资源使用效率. 在分析现有的集群资源调度框架的基础上, 设计并实现了一种可以感知批处理/流处理应用的混合式资源调度框架 HRM. 它以共享状态架构为基础, 采用乐观封锁协议和悲观封锁协议相结合的方式, 确保流处理作业和批处理作业的不同资源要求. 在计算节点上, 提供 CPU-GPU 资源的灵活绑定, 采用队列堆叠技术, 不但满足流处理作业的实时性需求, 也减少了反馈延迟并实现了 GPU 资源的共享. 通过模拟大规模作业的调度, 结果显示, HRM 的调度延迟只有集中式调度框架的 75%左右; 使用实际负载测试, 批处理与流处理共享集群时, 使用 HRM 调度框架, CPU 资源利用率提高 25%以上; 而使用细粒度作业调度方法, 不但 GPU 利用率提高 2 倍以上, 作业的完成时间也能够减少 50%左右.

关键词: 数据流模型; 批处理; 流处理; 作业感知; CPU-GPU; 队列堆叠

中图法分类号: TP311

中文引用格式: 汤小春, 赵全, 符莹, 朱紫钰, 丁朝, 胡小雪, 李战怀. 面向 Dataflow 的异构集群混合式资源调度框架研究. 软件学报, 2022, 33(12): 4704–4726. http://www.jos.org.cn/1000-9825/6356.htm

英文引用格式: Tang XC, Zhao Q, Fu Y, Zhu ZY, Ding Z, Hu XX, Li ZH. Research of Hybrid Resource Scheduling Framework of Heterogeneous Clusters for Dataflow. Ruan Jian Xue Bao/Journal of Software, 2022, 33(12): 4704–4726 (in Chinese). http://www.jos.org.cn/1000-9825/6356.htm

Research of Hybrid Resource Scheduling Framework of Heterogeneous Clusters for Dataflow

TANG Xiao-Chun, ZHAO Quan, FU Ying, ZHU Zi-Yu, DING Zhao, HU Xiao-Xue, LI Zhan-Huai

(School of Computer Science, Northwestern Polytechnical University, Xi'an 710129, China)

Abstract: The use of the Dataflow model integrates the batch processing and stream processing of big data computing. Nevertheless, the existing cluster resource scheduling frameworks for big data computing are oriented either to stream processing or to batch processing, which are not suitable for batch processing and stream processing jobs to share cluster resources. In addition, when GPUs are used for big data analysis and calculations, resource usage efficiency is reduced due to the lack of effective CPU-GPU resource decoupling methods. Based on the analysis of existing cluster scheduling frameworks, a hybrid resource scheduling framework called HRM is designed and implemented that can perceive batch/stream processing applications. Based on a shared state architecture, HRM uses a combination of optimistic blocking protocols and pessimistic blocking protocols to ensure different resource requirements for stream processing jobs and batch processing jobs. On computing nodes, it provides flexible binding of CPU-GPU resources, and adopts queue stacking technology, which not only meets the real-time requirements of stream processing jobs, but also reduces feedback delays and realizes the sharing of GPU resources. By simulating the scheduling of large-scale jobs, the scheduling delay of HRM is only about 75% of the centralized scheduling framework; by using actual load testing, the CPU resource utilization is increased by more than 25% when batch processing and stream processing share clusters; by using the fine-grained job scheduling method, not only the GPU utilization rate is increased by

* 基金项目: 国家重点研发计划(2018YFB1003400)
收稿时间: 2020-11-23; 修改时间: 2021-01-25; 采用时间: 2021-04-26; jos 在线出版时间: 2021-05-20

more than 2 times, the job completion time can also be reduced by about 50%.

Key words: dataflow model; batch process; streaming process; application aware; CPU-GPU; queue overlap

分布式批处理[1]是针对有界数据进行全面和精确的计算; 分布式流处理[2−4]是针对无界数据进行至少一次的无重复计算, 实时性要求高. 因此, 前者的资源调度目标是吞吐量; 而后者则是低延迟, 即保证服务等级目标(SLO).

批处理作业的资源调度框架一般采用尽力而为的策略, 通过先来先服务、资源的公平共享以及容量调度等方法, 尽量降低作业的完成时间, 提高系统的吞吐量, 确保不同批处理作业之间资源分配的公平性. 流处理作业的资源调度框架要求一次性分配足够的资源, 再通过资源的弹性扩大或者缩小, 尽量降低作业的处理延迟. 对于流处理作业, 如果采用公平资源分配, 可能出现资源不足的现象. 例如, 总的资源数为 100 个 CPU 核, 两个流处理作业都需要 60 个 CPU 核, 采用公平分配, 每个作业获得 50 个 CPU 核, 都无法达到资源需求.

面向流处理作业的资源调度框架和面向批处理作业的资源调度框架具有各自的特点, 分别被应用于流处理和批处理作业的大数据应用中. 但是自从 Google 的 Dataflow[5−8]模型的出现, 以及很多企业开始将有界的历史数据处理和无界的流数据的处理进行有机的结合[9], 开启了批处理作业和流处理作业共享同一个集群基础设施的新时代. 例如, 以 Flink[10], Spark[4,11]等为代表的基于 Dataflow 模型的大数据处理框架, 实现了批处理与流处理的统一, 既支持批处理应用, 也支持流处理应用.

但是, 现有的集群资源调度框架, 在流处理作业中采用紧耦合方式, 在批处理作业中采用松耦合方式, 因此无法感知作业的类别来提供不同的资源调度策略, 达到批处理/流处理作业的资源共享要求. 如果简单地将这两类作业共享在一个集群基础设施上, 当使用批处理资源调度框架的公平共享策略时, 就无法一次性为流处理作业分配足够的资源, 不能满足 SLO; 当采用流处理资源调度框架的弹性调度策略, 就必然会引起流处理作业和批处理作业之间资源分配的冲突问题. 在这种情况下, 就需要一种新的集群资源调度框架, 能够将批处理作业的资源分配策略和流处理作业的资源分配策略统一起来, 实现批处理作业和流处理作业之间的资源共享, 确保两类不同作业的资源分配需求.

文献[12–14]最早提出了混合式资源调度框架, 其目的在于解决长时间运行作业与短时间运行作业之间的资源分配冲突, 只适应于运行时间几十小时到几十天与运行时间几十毫秒到几秒的作业之间的集群资源共享, 这些资源调度框架无法满足流处理作业和批处理作业对资源的共享性要求.

随着大数据处理技术的发展, 大数据应用场景越来越广泛, 大数据分析过程中对于吞吐量和延迟的要求越来越高, 并且随着一些声音、图像等数据的出现, GPU 开始在大数据分析中被广泛使用. 但是, GPU 主要应用于计算密集型的大数据计算, 不太适合 I/O 需求巨大的大数据处理场景, 所以还需要 CPU 进行调度. 现有的集群资源调度框架, 要么针对大规模的 CPU 集群, 要么针对 GPU 集群, 对于 CPU 计算与 GPU 计算交替存在的大数据计算场合, 无法将 CPU 计算和 GPU 计算有机地统一起来, 缺乏 CPU-GPU 资源之间的解耦, 不能达到全面利用 CPU 和 GPU 资源的目的. 当 GPU 计算运行时, CPU 就处于空闲状态. 另外, 当 CPU 资源全部被使用时, 就可能导致 GPU 计算缺乏 CPU 资源而无法运行, 例如 Shuffle 操作等待 CPU 资源, 造成 GPU 资源的浪费.

目前, 面向大数据计算的 CPU-GPU 异构集群资源调度框架, 基本上通过扩展 CPU 集群的功能来实现, 还没有真正意义上的 CPU-GPU 集群资源调度框架. Yarn[15], Mesos[16], Kubernate[17]等集群资源调度框架将 CPU 作为主要资源, 将 GPU 设备作为次要资源来管理. 这些资源调度框架缺乏 CPU-GPU 资源的解耦. 如果 CPU 和 GPU 资源分配不合理, 就有可能发生 CPU 负载过重, 导致 GPU 操作等待 CPU 资源而产生 GPU 的“饥饿”或者相反情况的发生, 因而不适合面向 Dataflow 模型的大数据分析计算场合. 另外, 大多数系统采用粗粒度的 GPU 分配, 很少能够实现细粒度的资源共享.

鉴于以上的问题, 面向 Dataflow 模型, 建立一种适用于批处理作业和流处理作业融合的异构集群资源混合式调度框架就成为本文的核心目标. 通常情况下, 混合式调度框架应该具有以下特征: 一是可以感知作业

的特征, 不同类型的作业采用不同的资源调度策略; 二是 CPU 资源和 GPU 资源的灵活耦合方式; 三是尽可能共享使用计算资源, 以便达到集群资源的高效使用; 四是调度开销要尽可能的小, 使得调度器能够管理更多的计算资源和调度更多的作业.

本文的主要贡献是提供一种全新的面向 CPU-GPU 异构集群的混合式资源调度框架(heterogeneous resource management, HRM), 满足批处理和流处理作业的资源调度需求, 实现 CPU 计算资源和 GPU 计算资源的合理分配, 提高系统的资源利用率. 其主要工作以及创新点包括以下几个方面.

(1) 提出了基于共享状态的混合资源调度框架. 通过感知用户的作业类型, 采用乐观封锁协议和悲观封锁协议相结合的策略, 实现流处理和批处理作业的一体化资源调度框架;

(2) 提出了资源单元的概念, 可按照 CPU-GPU 之间的亲和性[18]以及 CPU 核数来设置资源单元, 实现了 GPU-CPU 之间的解耦;

(3) 通过在资源单元上实施队列堆叠, 使得流处理作业的资源可以一次性确保获得, 批处理作业的资源可以排队获得. HRM 在同一个资源单元上设置批处理队列和流处理队列, 形成队列的堆叠. 流处理队列具有高的优先级别. 当批处理作业与流处理作业产生资源竞争时, 由于流处理作业运行在流处理队列中, 具有较高的优先级别, 因此可以确保流处理作业的资源需求. 而批处理作业则采取尽力而为的调度策略;

(4) 通过容器在队列排队等待方式, 减少反馈延迟带来的资源浪费, 提高资源利用率. 资源调度过程是一个不断循环的过程, 即资源释放、状态通知、资源分配、资源再释放. 这种循环过程存在两个问题: (a) 作业释放资源, 但是心跳间隔时间还没有到来, 这部分资源就处于空闲状态; (b) 资源管理器收到心跳通知后, 进行资源的分配. 从资源管理器收到心跳信息到新的分配的容器运行之前, 计算节点上的资源空闲. 为了应对这类延迟, HRM 为计算节点设置了队列系统, 一旦运行中的容器结束, 队列中的容器立即从等待状态转换为执行状态, 避免了反馈延迟;

(5) 通过多容器在队列中同时运行机制, 实现 GPU 设备的细粒度资源分配, 使多个任务可以共享同一个资源单元, 提高 GPU 设备的使用率.

本文实现了 HRM 与 Spark 编程框架的对接, 并进行了模拟测试和实际负载的测试. HRM 系统在 8 台包含双 CPU 以及双 GPU 设备的集群上进行了实际负载的测试, 通过运行 Spark 编程框架对应的流处理和批处理的测试基准以及大量的 GPU 视频流应用, 与集中式资源调度框架进行了比较. HRM 的调度延迟只有集中式调度框架的 75%左右; 另外, 针对实际负载测试批处理与流处理共享集群资源时, 使用 HRM 调度框架, CPU 资源利用率提高 25%以上; 使用细粒度资源分配方法, 不但 GPU 利用率提高 2 倍以上, 作业的完成时间也能够减少 50%左右.

本文第 1 节介绍集群计算资源调度框架的现状. 第 2 节描述 HRM 的总体结构. 第 3 节讲述 CPU-GPU 计算资源的分割模式. 第 4 节描述批处理作业和流处理作业资源调度过程. 第 5 节介绍基于 Spark 框架的应用感知策略. 第 6 节给出 HRM 的性能模拟实验以及实际负载的测试结论. 第 7 节对全文进行总结概括.

1 相关研究进展

1.1 资源调度框架

集群资源调度框架分为集中式(Yarn, Mesos, Omega)、分布式(Sparrow, Apollo)和集中分布混合式(Mercury, Hawk, Eagle).

Yarn 提供 3 种调度策略, 即先来先服务(FIFO)、容量调度[19]和公平调度[20]. Mesos 是二级调度器, 它采用主资源公平(DRF)[21]方式, 其目的是保证不同作业的主资源占比公平. Omega[22]是共享状态调度器, 它实现了不同作业的并行调度, 减少了调度延迟. 但是, 这些调度框架无法满足流处理作业的弹性资源分配的需求. 因此, 这些资源调度框架不能提供一种既能满足流处理作业弹性资源分配要求, 又能满足批处理尽力而为的资源分配要求. 而 HRM 中的分布式调度器可以感知应用程序的类别, 并根据不同的类别的应用程序提供不同的

调度策略.

Sparrow[23], Apollo[24]等分布式资源调度框架适应于对调度延迟比较敏感的作业. 作业请求资源时, 它们向集群计算节点发出采样请求, 然后根据采样的结果, 由调度器从候选资源中选择最优资源并提交作业中任务. 但是采样策略无法获得最佳的计算资源, 也可能造成作业运行到同一个计算节点的可能, 调度效果大打折扣. 因此, 分布式调度器无法处理批处理作业和流处理作业混合的应用环境.

Mercury[12], Hawk[13]以及 Eagle[14]是混合式调度框架. 集中式调度器为长时间运行的作业分配资源, 分布式调度器为短时间运行作业分配资源. 这种方式基本上解决了短时间作业的长时间等待问题. 但是对于批处理和流处理混合的情况下, 无法为流处理作业弹性分配资源, 所以无法满足要求. HRM 采用乐观封锁协议调度批处理作业, 使用悲观封锁协议调度流处理作业, 批处理作业和流处理作业使用各自的队列, 满足流处理作业的弹性资源分配要求, 同时也能适应批处理作业的公平共享要求.

文献[25−42]是为了提高资源使用率和作业的性能, 对现有的集群资源调度框架采用了一些优化, 例如作业中任务延迟的缓解、多种资源的打包分配、基于资源保留的调度机制以及基于历史调度数据的强化学习调度等. 由于它们是在现有集群资源调度框架上完成, 所以无法解决流处理和批处理混合时的资源分配问题,也未涉及到 CPU-GPU 资源的解耦问题.

Storm[2], Naiad[3]等流处理计算框架, 它们使用专门的集群资源管理系统, 缺乏多租户之间的资源共享, 即无法完成批处理和流处理作业的资源共享. 集群资源不共享带来的主要问题是: 增加了基础设施成本; 数据在不同集群之间的迁移容易造成错误; 各个集群的资源利用率不高. HRM 让批处理作业和流处理作业共享同一个集群, 解决了这些问题.

1.2 GPU资源调度

GPU 资源调度主要包含多任务在单个 GPU 设备的共享以及多任务在多个 GPU 设备的共享. 文献[43−46]研究多任务在单个 GPU 设备上的调度, 允许多个任务共享单个 GPU 设备. 文献[47,48]针对云环境下运行时间较短的 GPU 任务, 实现了一个基于容器的批处理计算系统, 但是它无法完全利用 CPU 或者 GPU 资源. 文献[49]中提出在高性能计算中引入 GPU 集群, 采用 Torque 来调度 GPU 作业. 它是一种静态的 GPU 资源共享模式, 在作业运行之前, 为每个不同的作业分配固定的 GPU 资源, 主要应用于 HPC 上, 其适用于计算密集型任务, 不适合 I/O 复杂的大数据分析任务. 文献[18]提供了一种使用 GPU 的易用方式, 可以提高开发者的效率. 文献[50−57]将 CPU 资源和 GPU 资源都纳入动态分配范围, 但是这些系统将 CPU、内存等作为主要资源来设计资源分配算法, GPU 只是辅助. Yarn 和 Kubernate 虽然也支持 CPU-GPU 资源分配, 但是它们是粗粒度的, 也不考虑 CPU 和 GPU 的亲和性.

这些资源管理框架只考虑 GPU 资源, 缺乏 CPU 与 GPU 资源的解耦, 不考虑 CPU 与 GPU 之间的亲和性, 也不考虑 GPU 执行中对 CPU 的依赖, 应用于大数据计算时, 其后果是要么 CPU 资源的浪费, 要么 GPU 资源浪费. HRM 系统中, 将 CPU-GPU 作为独立的资源单元来管理, 为每个资源单元提供队列机制, 实现 CPU 任务和 GPU 任务的单独调度, 适应于大数据分析计算; 同时也实现了细粒度的共享和弹性资源分配, 支持流处理和批处理作业混合的计算环境.

2 HRM 系统模型

在本节中, 首先给出 HRM 的总体说明, 然后给出了资源调度的总体框架以及各个组成部分的功能介绍.

2.1 HRM总体结构

HRM 总体结构采用共享状态架构, 由多个调度器、一个资源协调器、资源监视器以及队列管理器组成. 图 1 给出了 HRM 资源调度框架的总体结构.

最上层是用户的应用程序(后面简称作业, job). 对于每个作业, AppMaster 是核心, 与集群资源相关的功能包括任务管理以及与集群资源之间的各种消息交互. 任务管理主要是任务的调度以及任务执行中的控制;

与集群计算资源的交互主要是向集群注册、获得集群资源的状态以及容器的提交等.

中间层是资源调度框架. 资源调度框架包括多个分布式调度器(distributed scheduler)、资源协调器(negotiator)、作业状态(job states)以及资源状态(resource states)这 4 个部分. 作业状态是整个集群上运行的作业的信息, 包括作业的开始时间、作业中的 stage、每个 stage 中的任务数量、未完成任务和正在执行的任务数等. 资源状态是各个计算节点上可用资源状态, 例如可用资源数量、正在运行的容器的状态和数量以及集群计算节点的健康状态等. 资源协调器是接收各个计算节点的心跳信息, 并经过安全检查和一定的预处理后, 更新全局资源状态, 同时还负责将资源仲裁的结果向作业的调度器汇报. 分布式调度器可以感知作业类型, 对于流处理作业执行流处理资源分配策略(data stream policy), 批处理作业则执行批处理资源分配策略(batch policy). 关于这两种策略, 会在第 4 节进行详细说明.

底层是计算节点的管理. 计算节点管理包括计算节点上的队列管理器(queue system)[58,59]以及节点资源监控器(resource tracker)这两部分. 采用队列的堆叠技术, 计算节点队列分为批处理队列(batch queue)和流处理队列(stream queue), 分别用于处理批处理任务和流处理任务, 两种队列的调度属性不同(第 3 节介绍). 队列管理器向用户提供接口, 用来创建队列、删除队列以及设置队列的属性, 即设置队列的资源容量信息、队列的优先级别、队列的长度等. 队列管理器对提交到队列上的容器(也称为执行器)进行调度安排和提交执行.

AppMaster 是用户作业(job)的一个进程, 运行在某个计算节点上, 拥有自己的状态并管理作业中的全部任务. 当用户提交作业时, AppMaster 负责向资源管理器注册并请求资源的分配. 一旦获得资源, AppMaster 管理作业中的任务, 监控任务的状态并控制任务的运行. 由于 AppMaster 掌握作业输入数据的存储信息、分片信息以及作业执行过程中临时数据的存储信息, 因此作业一旦获得计算资源, 就按照一定的调度策略(如数据的本地化访问等)将每个任务发送到计算节点上进行执行.

HRM 中包含两类心跳信息: 一类是资源状态的汇报, 另外一类是作业状态的汇报. 第 1 类是计算节点(node)上资源监控器(ResourceTracker)与协调器之间的通信, 定时向资源管理器汇报计算节点的状态、容器的状态等. 第 2 类是用户的应用程序(AppMaster)与分布调度器(distributed scheduler)之间的心跳信息交互, 定时更新作业状态的信息(job status), 例如作业的任务数, 已经完成, 未完成任务数量等.

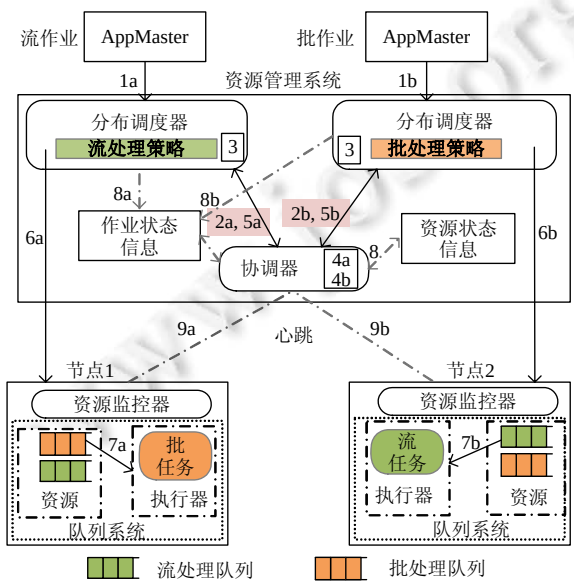


图 1 HRM 调度框架

详细的资源分配和容器调度过程图 1 所示(以下的步骤序号与图 1 中序号一一对应).

- 1a(1b). 作业的调度请求. 1a 为流处理作业向一个分布式调度器发送任务提交请求, 1b 为批处理作业

向一个分布式调度器发送的任务提交请求;

- 2a(2b). 对于批处理作业, 调度器从全局集群资源状态信息(resource status)获得一个拷贝; 对于流处理作业来说, 是发送一个资源加锁请求, 获得全部资源状态并进行调度;
- 3. 分布式调度器根据全局资源状态信息进行资源分配, 然后将资源分配结果发送到协调器(negotiator);
- 4a(4b). 对于批处理作业, 协调器对分布式调度器的调度结果进行冲突检查: 对于不存在冲突的资源分配信息, 更新全局资源状态(图 1 中的步骤 8); 如果存在冲突, 则进行冲突解决(详细参考后续的章节). 对于流处理作业, 更新全局资源状态(图 1 中的步骤 8), 释放对资源封锁;
- 5a(5b). 协调器向调度器发送调度结果;
- 6a(6b). 分布式调度器根据资源分配的最终结果, 向集群的计算节点发送消息, 将容器的启动命令提交到队列中;
- 7a(7b). 计算节点的队列管理器按照资源约束信息, 启动任务执行所需的容器;
- 8a(8b). 作业定期通过调度器向协调器汇报作业的状态信息, 比如运行的任务数量等信息;
- 9a(9b). 节点资源监控器定时向协调器汇报节点的健康状态消息以及节点上容器的状态消息.

2.2 全局资源状态信息

HRM 系统采用全局资源状态信息来为调度器提供分配依据, 各个分布式调度器调度前, 首先获得全局资源的状态的一个备份, 然后按照其信息进行并行的资源分配, 从而减少了次优分配的概率. 另外, HRM 的流处理和批处理作业之间设置不同的优先级别, 保证流处理作业对资源的抢占式调度, 一定程序缓解了资源分配冲突的问题.

2.3 作业状态信息

对于分布式调度框架, 由于各个调度器独立工作, 调度器之间无法进行消息通信, 因此各个调度器的负载情况就不能被共享, 导致调度器无法决策各个作业在全局资源中应该使用的资源份额. HRM 设置作业的状态信息, 用来决策全局资源在各个作业之间的公平共享. 协调器掌握整个集群的负载大小, 确保全局的分配策略, 如资源共享的公平性、任务在队列中重新设置优先级别等等.

3 计算节点上的队列管理器

计算节点上的队列管理器是任务执行的基础, 任务使用计算节点的物理资源来执行. 因此, 高效地管理集群计算节点的可用资源, 是一个非常重要的环节.

3.1 CPU-GPU资源单位的划分

大数据处理中, 对 GPU 的资源分配完全不同于高性能计算. 高性能计算的每个作业需要强大的算力, 较少涉及数据的 I/O. 但是, 基于数据流计算模型的大数据分析作业是由许多操作组成的有向无环图(DAG), 像洗牌操作(shuffle)、网络传输等操作只能在 CPU 上执行, 而且占有大量的比例.

首先, 本文假设集群中所有 CPU 核的计算能力是相同的, 但是因为 CPU 和 GPU 之间的亲和性不同, 导致 GPU 在使用不同 CPU 时所造成的延迟也是不尽相同的. 因此, 在 CPU 和 GPU 分配方面, 需要考虑以下几个方面: (1) 如果 CPU 和 GPU 分割不合理, 就可能发生 CPU 饥饿或者 GPU 饥饿的情况; (2) CPU 和 GPU 之间存在亲和性关系, GPU 对应的 CPU 核在计算过程中不断变化, 必然会导致主机内存到设备内存数据传输性能的下降.

HRM 采用资源单元策略(resource unit, RU), 使用 Linux 提供的容器技术进行隔离, 每个 RU 是一个独立的资源分配单元. 对于每个 RU, 按照亲和性为 GPU 设备设置一定数量的 CPU 核. 为此, HRM 提供 3 种方式来划分 RU: (1) 计算节点上的每个 GPU 作为一个 RU, 全部 CPU 形成一个 RU, 可以被 GPU 共享; (2) 每个节点上, 将一定数量的 CPU 核与 GPU 绑定在一起, 形成一个 RU, 剩余 CPU 可以分成一个或者多个 RU; (3) 依

照 GPU 数量, 将 CPU 核均匀地划分给 GPU 设备, 形成不同的 RU. 其结果如图 2(a)–图 2(c)所示, 图中的 *c*0, *c*1 等代表 CPU 核, *GPU*1, *GPU*2 则代表 GPU 设备. 图中每个带填充的封闭不规则矩形框代表一个资源单位 RU. 图 2(a)中, 所有的 CPU 为一个单独的 RU, 两个 GPU 分别代表两个不同的 RU, 这种分配不考虑亲和性, GPU 任务执行中共享 CPU 资源, 而 CPU 任务则使用任意数量的 CPU 核. 这种方式的最大问题是 CPU 共享容易导致资源使用的干涉, 当 CPU 被单独分配给不同的任务时, GPU 无法获得 CPU 资源而产生饥饿现象. 图 2(b)中, CPU 资源和 GPU 资源单独分配, 但是按照亲和性为 GPU 资源预留一定的 CPU 资源, 计算节点的资源分为 3 个 RU, 一个 GPU 和一个 CPU 核组成一个 RU, 剩余的 CPU 核组成另外一种类型的 RU. 这种方式的缺点是 GPU 的 kernel[49]执行时间较长时, 使得预留的 CPU 资源大量闲置. 图 2(c)中, 依照 GPU 数量, 均匀捆绑 CPU, 即为每个 GPU 安排一定数量的 CPU. 这种方式的缺点是: GPU 任务执行过程中, 由于 CPU 指令相对较少, 安排太多的 CPU 物理资源可能导致 CPU 利用率非常低. 另外, 如果任务是一个 CPU 任务, 其执行过程中不需要任何 GPU 设备, 那么 CPU 绑定的 GPU 就被闲置, 反而影响其他任务对 GPU 资源的分配请求.

CPU-GPU 资源分割由用户自己选定, 依据亲和性以及资源需求动态灵活地绑定. 但是可以制定以下启发式原则.

- 如果作业类型主要是 CPU 作业, GPU 作业数量较少时, 选择图 2(a);
- 如果作业类型既包含 CPU 作业, 也包含一定数量的 GPU 作业时, 按照亲和性关系和 CPU 核数量的要求, 选择图 2(b). 这里, 在进行 CPU 和 GPU 的绑定过程中, 若是 GPU 作业使用 GPU 并涉及大量 CPU 计算时, 则进行绑定的时候要适当增加给 GPU 绑定的核数, 但要预留一部分 CPU 核来运行 CPU 作业; 若 GPU 作业主要是用 GPU 计算, 仅需要使用 CPU 进行内存拷贝, 则只需要将与该 GPU 存在亲和性的 CPU 核绑定在该 GPU 上即可;
- 如果作业全部是 GPU 作业, 例如大量的图片检测程序时, 只需要将所有 CPU 核平均分配到各个 GPU 上即可, 选择图 2(c).

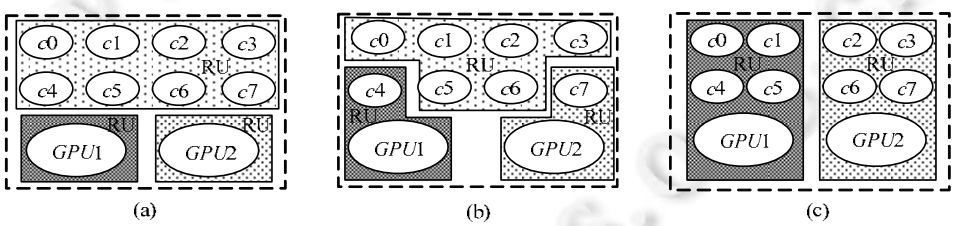


图 2 CPU-GPU 分配方式

3.2 资源单位与队列的映射

为了高效利用每个资源单元, 需要设置队列来调度任务. 通常, GPU 操作的执行离不开 CPU 的调度, 所以 GPU 操作执行中, 总是存在 CPU 的执行, 其执行模型如图 3 所示.

图 3 是一个典型的大数据分析作业中 GPU 操作执行模型, 图中的 E^{hd} 代表 GPU 需要的数据的装箱过程; E^{dh} 代表 GPU 操作的结果数据的传输; E^{g} 代表在 GPU 上执行 kernel 函数; E^{c} 代表 GPU 操作使用 CPU 来处理数据的过程, 例如转送 Map 的局部 K/V 数据到 Reduce 操作上. 一个 GPU 操作在执行过程中, 首先使用 CPU 进行初始化 GPU 设备, 然后利用 CPU 装箱数据并传输到 GPU 设备内存, 当数据装箱完成, CPU 启动 GPU 的 kernel 函数, 开始执行. kernel 函数执行过程中, CPU 资源就处于空闲状态, 一旦 kernel 函数执行结束后, 开始使用 CPU 从设备内存传输数据到主机内存, 此时 GPU 处于空闲状态. 如此反复, CPU 和 GPU 交替处于空闲状态, 造成资源利用率的降低.

为了解决 CPU 和 GPU 不能完全利用而导致的资源空闲问题, HRM 通过队列机制将资源单元与队列进行了映射, 通过 Linux 的 Cgroup 子系统以及其层次结构来实现资源单元与队列的映射. 具体过程分为 3 步: 首先为队列创建控制组, 控制组用队列编号命名; 其次, 根据资源单元为队列上的控制组设置资源参数, 确保

资源单元与其他资源单元之间的隔离; 最后, 一旦作业运行在队列上, 将作业的进程 ID 添加到控制组, 满足作业共享资源单元. 所使用的 Cgroup 子系统有 cpuset, devices, memory 和 cpuacct.

每个大数据作业的运行时系统包含一个调度器和多个执行器, 调度器管理作业中的操作, 执行器用来执行作业中的各种操作. 为了防止作业之间的干涉, 执行器以容器的方式存在. 由于 CPU 与 GPU 操作是并发, 即交替执行, 通过设置队列的同时运行容器数量, 让资源单元同时运行多个操作, 从而提高资源单元的利用率. 当然, 同时运行数量设置太多, 可能会导致资源的竞争, 反而影响作业的性能. 具体策略为:

(1) CPU 操作与 GPU 操作之间的资源共享. 为了提高 CPU 资源利用率, 将 GPU 操作的剩余 CPU 时间分配给其他 CPU 操作. 为了防止 CPU 操作与 GPU 操作对 CPU 资源的竞争, 通过队列为 GPU 操作设置较高的优先级别, 为 CPU 操作设置较低的优先级别, 从而保证 GPU 操作能够得到足够的 CPU 计算资源, 避免两种不同类型的操作执行过程中的等待或者干涉;

(2) GPU 操作与 GPU 操作之间的资源共享. 单个 GPU 操作独占 GPU 设备时, 会造成 GPU 资源的浪费. 为了提高 GPU 设备的利用率, 通过队列机制, 当 GPU 设备的显存还存在剩余的话, 加载其他作业的操作到同一个 GPU 设备, 实现 GPU 计算资源在多个操作之间的资源共享, 如图 4 所示. 图 4 中包含 2 个操作共享 GPU 设备, GPU 操作的执行模型与图 3 描述的一样. 但是我们可以看到: *task*1 和 *task*2 共享一个 GPU 设备时, 两个 GPU 操作的 kernel 函数可以交替并发, 提高了 GPU 设备的利用率; 另外, 不同操作的 CPU 指令和 GPU 的 kernel 函数之间可行并行运行, 也提高了整体物理资源的使用效率. 但是, task1 和 task2 的执行中, 为了保证两个操作使用的显存之和不超过 GPU 设备的最大显存, 需要在本地节点建立队列机制, 防止 GPU 显存的溢出.

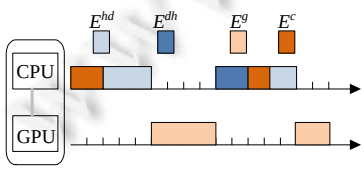


图 3 大数据分析中 GPU 操作的执行模型

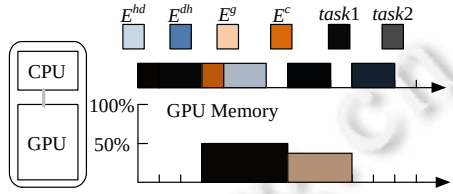


图 4 GPU 设备资源的细粒度共享

3.3 计算节点上的队列堆叠模型

在单个计算节点上, 设置传输队列和执行队列. 如图 5 所示, 图的节点(node)代表一个计算节点, 执行队列与资源单元 RU 相关, 用于执行用户的请求(用户的容器). TP 代表传输队列, 传输队列向执行队列转送请求, 根据任务请求的优先级别以及请求需要的资源要求, 将请求提交到不同的执行队列上.

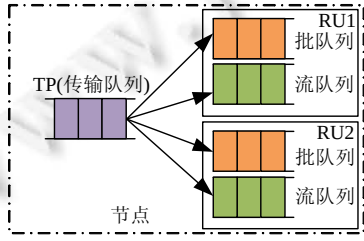


图 5 RU 上队列的堆叠模型

每个 RU 上采用队列堆叠技术, 设置两个执行队列, 分别为批(batch)队列和流(stream)队列, 这两个执行队列共享 RU 对应的物理资源. 这就意味着: 相同的物理资源上, 虚拟了两个可执行队列. 执行队列具有以下属性.

- 队列类型. 如果 RU 中包含 GPU 设备, 那么队列分类为 GPU 队列; 如果 RU 中不包含 GPU 设备, 那

么队列分类设置为 CPU 队列;

- 同时运行数. 每个 RU 包含有具体的资源数量, 单个容器无法完全使用全部的资源数量时, 就可以提交多个容器同时运行, 提高 RU 的利用率;
- 排队数量. 容器在队列中排队, 当存在空闲资源时, 可以立即执行排队中的容器, 降低调度过程中的反馈延迟;
- 优先级别. 优先级别决定队列中等待容器的优先执行级别, 高优先级别的流处理队列采用抢占式调度策略, 优先获得资源. 堆叠技术可以确保流处理作业对于资源的弹性分配. 如果流处理作业负载较高, 那么批处理作业获得资源的机会减少, 但是批处理作业可以通过排队来获得需要的资源;
- 资源限制. 容器请求的资源超过限制, 则请求不被调度执行.

3.4 队列参数的计算

3.4.1 队列上同时运行容器数量的计算

作业的容器在队列上由等待状态变为运行状态, 是由队列上的可用计算资源决定的, 容器的资源总和不能超过 RU 上的可用资源总和.

对于 CPU 队列, 假如 RU 中的可用 CPU 核数为 $RU.vCore$ 个, 可用内存 $RU.men$. 此时, 如果队列中运行着 p 个容器, 即 $t_1,t_2,\ldots,t_p$, 如果新到来的容器为 t_{p+1}, 它们使用的 CPU 核数以及内存大小分别为

$$t_1.vCore,t_2.vCore,\ldots,t_{p+1}.vCore,\ t_1.men,t_2.men,\ldots,t_{p+1}.men.$$

此时, 只要满足 $\sum_{i=1}^{p+1}t_i.vCore \leqslant RU.vCore$ 并且 $\sum_{i=1}^{p+1}t_i.mem \leqslant RU.mem$, 那么新容器 t_{p+1} 就可以在从排队状态转入运行状态, 此时队列中同时运行的容器数为 p+1.

对于 GPU 队列, 假如 RU 中 GPU 的显存大小为 $RU.gmen$. 如果队列中运行着 p 个容器, 即 $t_1,t_2,\ldots,t_p$, 每个容器使用的显存大小为 $t_1.gmen,t_2.gmen,\ldots,t_p.gmen$. 如果新到来的容器为 t_{p+1}, 它需要的显存为 $t_{p+1}.gmen$. 此时只要满足 $\sum_{i=1}^{p+1}t_i.gmem \leqslant RU.gmem$, 那么新容器 t_{p+1} 就可以在从排队状态转入运行状态, 此时队列中同时运行的容器数为 p+1.

3.4.2 队列中排队状态容器数量的计算

在 HRM 中: 对于流处理作业, 使用流处理队列来管理容器的执行; 对于批处理作业, 使用批处理队列来管理容器的执行. 由于流处理作业的运行时间通常是难以预料的, 所以调度器分配流处理作业的容器时, 不设置容器在队列中的排队机制, 即流处理作业的资源分配仍然采用释放、分配、使用、再释放的循环模式. 对于批处理作业, 由于其运行时间相对较短, 因而在批处理作业调度过程中, 为了减少反馈延迟, 采用容器在队列中排队的机制. 当前一个运行的容器结束后, 后续的容器从等待状态直接转换为执行状态, 防止反馈延迟带来的资源空闲, 提高了资源的使用效率.

如果队列排队长度设置太短, 就会存在资源空闲而导致集群资源利用率下降; 反之, 如果队列排队长度设置太长, 就可能导致一些容器等待延迟变大. 实际上, 只要集群中的资源无法满足作业对资源的需求, 那么一些容器就需要在队列中等待.

在实际的应用中, 超过 60%的作业是重复发生的. 因此, 可以通过作业完成时间来估计队列的长度. 对于这样的作业, 我们依据历史的作业运行记录来预估容器的运行时间. 如果缺乏这样的估计, 使用默认的作业完成时间, 并扩展了作业调度器动态调整的功能, 即: 通过观察实际的容器持续时间, 并随着作业的运行不断完善初始估计.

所有的队列都存在一个预定义的队列排队长度 b, 资源管理则只允许存在 b 个容器在队列中排队.

(1) 基于固定执行时间的队列长度. 假设所有的容器的运行时间都为 $1/\mu$(μ是容器中任务的处理速率), RU 上同时运行的容器数为 r(初始值等于该队列上对应资源单元内所包含的 CPU 核数), 设τ是心跳的时间间隔, b 是在队列中等待的容器数量, 那么 RU 的队列需要满足如下条件才能不会使资源空闲, 即 $r+b \geqslant r\mu\tau$. 根据此公式, 可确定允许排队长度 b 的值;

(2) 基于延迟的队列长度. 当作业运行时间变化较大时, 维护队列的固定长度就不能很好地利用资源. 当多个短时间运行的容器碰巧出现在队列中, 那么就可能造成资源的空闲. 当队列中存在长时间运行的容器, 就可能造成某些容器启动的延迟. 因此, 当容器的运行时间能够确定时, 使用基于延迟的策略来设置队列长度.

在特殊情况下, 指定一个容器在队列中等待的最大时间 $WT_{\max}$. 当我们向第 n 个 RU 的队列上提交容器 c 时, 需要计算 RU 上需要等待的时间 WT_n. 如果 $WT_n<WT_{\max}$, 那么容器 c 就可以提交在该队列上. 容器一旦在队列中排队, 资源协调器在考虑容器 c 的基础上, 使用一个简单的公式来更新 WT_n 和资源状态. 当一个新的心跳信息达到, WT_n 就会被刷新. 使用这种方式, 在 RU 的队列上处于等待状态的容器数量就会动态变化, 动态改变的依据是计算节点的负载以及正在运行和排队中的容器数量.

4 基于队列状态的分布式资源调度

作业一旦向资源调度器注册后, 资源调度器就需要为这些作业申请资源. 作业获得资源后, 再启动容器并运行任务. 流处理作业的资源分配频率低, 但是需要确保能够获得足够的资源.

4.1 资源单位的状态信息

作业请求资源时, 资源调度器需要根据整个集群的可用资源状态来进行决策, 所以全局资源状态信息是调度的依据. 一个计算节点的资源信息一般包含 CPU 核数量(*vCore*)、可用的内存大小(*Mem*)、可用的磁盘容量(*Store*)、可用的磁盘带宽(*IO*)、可用网络带宽(*Net*)、可用 GPU 设备(*GPUID*)数量、单个可用的 GPU 设备的显存大小(*GMem*)等信息.

HRM 使用队列等待矩阵来描述这些信息. 每个队列记录着需要一定 CPU、内存大小或者 GPU 等资源的容器的预期等待时间. 基于正在运行的容器以及等待中的容器, HRM 建立一个预期等待时间的矩阵. 对于每一个容器, 也包含一个容器的执行时间, 这个时间是作业的理想运行时间. 队列等待矩阵随着容器的执行不断地被更新着. 资源调度器开始调度时, 获得队列等待矩阵的最新更新值, 并作为调度依据.

为了预测队列的等待时间, 集群资源信息分为资源单元状态表和容器状态表. 资源单元信息表存储可用的 RU, 采用关系模型存储, 表示为 *RS*(*RUId*,*MId*,*type*,*vCore*,*Mem*,*Gmem*,*Wtime*), 其中, *RUId* 表示资源单元编号, *hostId* 代表计算节点编号, *type* 代表资源的类型, *vCore* 代表可用 CPU 的核数, *Mem* 代表可用内存的大小, *Gmem* 代表 GPU 设备可用显存大小, *Wtime* 代表最后一个容器的等待时间, 也代表着队列的预测等待时间. 容器状态表是队列中正在运行的容器和等待中的容器信息, 亦采用关系模式, 表示为 *RT*(*queueId*,*TaskId*,*Time*,*vCore*,*GPU*,*GMem*,*status*). *queueId* 代表队列的编号, 其用于和队列信息关联; *TaskId* 代表容器的编号; *Time* 表示任务预计执行时间; *vCore* 代表任务需要的 CPU 核数; *GPU* 代表需要的 GPU 设备数; *GMem* 代表 GPU 设备的显存, 如果只使用 CPU, 那么 *GPU* 和 *GMem* 的值为 0; *status* 代表容器的当前状态, 比如运行状态或者等待状态. 如果一个 RU 上存在多个容器, 那么 RT 表中就存在多行.

4.2 队列中容器的等待时间预测

当集群的工作负荷较轻时, 各个作业的资源需求都会得到满足, 因此队列的预测等待时间为 0; 当集群的工作负荷较重时, 各个队列上同时运行的容器数达到物理资源的最大值, 如果还存在作业的资源请求时, 作业对应的容器就需要在队列上排队. 如果存在多个队列时, 调度器需要选择一个排队时间最短的队列来运行容器, 而计算排队时间最短的依据就是队列的负载信息. 计算节点计算出各个队列的负载并提供给调度器, 调度器依据负载来提供资源. 队列负载小, 被优先选择的机会更高.

在计算队列的预测等待时间方面, 我们将作业分为均匀作业和不均匀作业两类.

(a) 均匀作业的队列等待时间的计算

资源调度器为作业分配计算资源前, 需要给出一个等待时间最少的队列, 作为候选资源来反馈给作业. 当作业的执行时间相对均匀时, 其等待时间就等于队列的排队长度. 均匀作业是计算每个队列排队长度. 队

列排队长度较小的计算节点提供优先分配的机会. 这种方式可能导致一个失败的结果, 例如: 对于一些作业异构的场合, 一个计算节点上有 2 个容器在排队, 每个运行时间是 500 s; 而一个计算节点有 5 个容器在排队, 每个运行时间只有 2 s. 那么由于前者队列的长度是 2, 具有较高的优先级别, 调度器选择这个计算节点. 实际上, 这个选择不是最好的, 从等待时间上看, 我们应该选择后者.

所以, 通过计算队列中等待作业的长度的方法只能适应于那些任务大小非常规整的场合, 例如图片的实时检测, 每个图片数据大小一样, 花费的检查时间也一样, 资源需求变化不大. 但是, 如果任务大小是变化的, 计算时间也变化, 这种调度方式就不是最好. 因此, 对于任务大小变化、计算时间也变化的场合, 我们通过估计每个 RU 的预计开始时间来进行调度, 选择最短的开始时间, 提供给需要调度的作业.

(b) 不均匀作业的队列等待时间的预测

当一个作业调度器为自己的容器提交资源申请时, 资源管理器开始计算容器运行在每个 RU 上的预计开始时间, 最后从这些 RU 上选择一个预计开始时间最小的队列, 作为容器运行的资源.

算法 1 给出了队列中容器的等待时间计算方法. 算法的输入是作业的一个容器, 算法的输出是容器到计算节点的资源单元 RU 的一个预计等待时间最短的映射. 另外, 算法的输入中包括一个作业类型变量, 分别代表均匀作业和不均匀作业. 算法 1 中, 首先检查计算节点上的全部 RU, 如果有可用计算资源满足容器的资源需求且无任何排队的任务, 那么该容器的等待时间为 0(算法 1 中的 S2–S7); 否则就需要遍历所有的 RU, 计算 RU 上的运行中的容器以及等待中的容器, 得到最小的等待时间, 从而得到请求资源的容器的预计开始时间.

算法 1. 作业等待时间计算.

输入: *executor*: 代表作业中的一个容器,

RUs: 所有的 RU 列表,

Type: 作业类型 //*Type*=0 代表均匀作业, *Type*=1 代表不均匀作业;

输出: *result*: 包含队列长度(等待时间)以及对应的资源单元.

```
S1:  wTime, minRu, queue
S2:  foreach (ru in RUs) do
S3:    if ru.res≥Executor.res and ru.waitTask==0 then
S4:      result←(0,ru)
S5:      return result
S6:    end
S7:  end
S8:  foreach (ru in RUs) do
S9:    if Type==0 then   //均匀作业
S10:     len=ru 中排队的任务数量和
S11:     if waitL>len then
S12:       waitL←len
S13:       minRu←ru
S14:     end
S15:   else
S16:     if ru.res<executor.res then
S17:       continue;
S18:     end
S19:     foreach (t in ru.runExecutors) do
S20:       t.time←t.d−t.e
S21:       把 t 放入运行队列 queue 中
```

```
S22:        end
S23:        waittime←simulaterun(queue,executor,ru)
S24:        if waittime<wTime then
S25:           wTime←waittime
S26:           minRu←ru
S27:        end
S28:     end
S29: end
S30: result←(wTime,minRu)
S31: return result

Function: simulaterun(queue,key,ru)
S1:   task←queue 中等待时间最小的任务
S2:   将 task 从 queue 中删除
S3:   ru.res←ru.res+task.res        //该任务完成释放资源
S4:   foreach (t in ru.waitTasks) do
S5:      if t.res≤ru.res then       //检查等待队列中是否有任务满足资源要求
S6:         t.time←t.d+t.time
S7:         ru.res←ru.res–t.res     //ru 的资源被使用
S8:         把 t 放入运行队列 queue 中
S9:      end
S10: end
S11: if key.res≤ru.res then         //ru 有空闲资源满足需求
S12:    return task.time            //返回
S13: end
S14: if task 不存在  then           //ru 有空闲资源满足需求
S15:    return MAXtime              //返回
S16: end
S17: return simulaterun(queue,key,ru)   //递归模拟新任务进入执行状态后, 最短等待时间
```

针对最小等待时间的计算, 算法 1 在计算的时候将作业区分为均匀作业和不均匀作业. 均匀作业只需要简单地统计队列上的排队长度即可, 无需像不均匀作业计算最小等待时间那样复杂. 算法 1 中的 S9 代表均匀作业, S15 代表不均匀作业. 对于均匀作业, S10–S14 分别统计每个队列中等待作业的数量, 然后选择一个等待作业最小的队列作为输出.

对于不均匀作业, 算法以当前运行中的容器的剩余时间以及排队中的容器的运行时间为输入, 模拟容器的执行过程. 算法 1 中, S8 遍历计算节点上的所有 RU. S20–S22 计算出运行中的容器的剩余运行时间, 然后将这些容器信息添加到队列 *queue* 中. S23 中的函数 *simulaterun* 用来模拟等待队列中的所有容器的执行过程, 这些容器在执行完成后就会释放资源, 如果释放的资源满足请求资源的容器的资源要求, 那么就得到一个当前资源单元上的预计开始时间. S24–S27 用来获得 RU 上的最小预计等待时间, S31 将结果返回该给调度器, 用于提交容器.

函数 *simulaterun* 是一个递归调用过程, 来模拟队列上已有容器的运行过程. 一旦运行中的一个容器结束(如函数 *simulaterun* 中的 S2 步骤. 后面的描述中直接用标记来代表算法的过程), 容器释放资源(S3), 然后从等待队列中选择满足资源要求的容器, 容器的状态从排队状态转变为执行状态. 此时, 计算当前资源状态下

该容器的预计运行时间(S6). *ru* 可用资源减少(S7), 容器进入队列(S8). 遍历完 *ru* 中等待状态的容器后, 检查 *ru* 是否存在可用资源满足请求资源的容器的资源需求: 如果满足, 则预计开始时间为已经执行完毕的容器的预计运行时间; 如果不满足, 递归调用函数 *simulaterun*(S17), 一直到获得一个满足新容器需求的资源(S12), 返回预计开始时间, 或者所有容器都执行完成, 此时得到的预计开始时间为所有任务包括运行任务以及等待任务的剩余运行时间(S12). 预测的等待时间是调度器选择资源的依据. 根据等待时间进行预测, 可以最大限度地保证每个任务都可以分配到最“理想”的资源.

算法 1 中, 只使用不均匀作业分支, 也适应于均匀作业的场合, 但是会导致算法的复杂性增加. 假如等待状态的任务和运行状态的任务数为 n 的话, 对于均匀作业, 计算最小等待时间的时间复杂度最坏情况下是 $O(n)$; 对于不均匀作业, 使用 *simulaterun* 过程进行递归, 计算最小等待时间的时间复杂度最坏情况下是 $O(n^2)$. 因此, 为了提高均匀任务场合下的计算效率, 算法 1 对于两种类型作业设置不同的计算过程.

4.3 资源调度过程

4.3.1 流处理作业的资源调度

流处理作业的容器运行时间一般也比较长, 但是每个微批次数量不大, 运行中可能只是部分弹性扩展或者缩小. 基于这些信息, 资源调度器分配采用悲观封锁协议来分配资源. 其过程主要包含以下步骤.

(1) 流处理作业向某个分布式调度器注册;
(2) 分布式调度器向协调器发出资源封锁的消息;
(3) 封锁成功后, 协调器将最新的流处理队列状态信息发送给分布式调度器;
(4) 分布式调度器根据流处理作业容器的资源要求, 按照算法 1 计算出等待时间为 0 的可用资源;
(5) 分布式调度器向协调器发送调度结果, 并释放封锁;
(6) 分布式调度器根据资源所在计算节点信息, 启动容器.

当存在两个或者两个以上的流处理作业同时请求资源时, 协调器根据先来先服务的策略, 为每个作业分配足够的计算资源. 如果集群的整体资源无法满足两个以上的流处理作业的资源需求, 那么无法获得足够资源的流处理作业将被拒绝运行.

4.3.2 批处理作业的资源调度

批处理采用运行一次的方式, 每个批次的数据规模较大, 资源调度器采用乐观封锁协议来进行批处理作业的资源分配. 分布式调度器可以获得一个不断更新的批处理队列状态信息的副本, 再利用状态信息副本来进行调度. 一旦调度器决策了作业的资源分配, 就请求协调器进行一次资源状态的更新. 在大多数情况下, 更新操作能够成功进行, 发生冲突的可能性比较小. 不管更新是否成功, 分布式调度器都会再次获得资源最新状态信息, 继续进行下一次的资源分配操作.

HRM 的分布式调度器是完全并行运行, 为了避免资源更新时的冲突, 调度器采用增量更新方式, 也就是说发生冲突的资源被放弃, 没有发生冲突的资源正常更新. 批处理作业资源分配流程包含以下步骤.

(1) 各个集群节点向协调器汇报最新状态;
(2) 分布式调度器获得全局计算资源后, 进行资源分配;
(3) 调度器提出资源信息更新, 协调器经过检查后没有冲突, 更新全局资源状态矩阵; 如果发现存在资源冲突, 协调器将冲突资源写入队列中;
(4) 将分配结果通知分布式调度器, 由调度器向计算节点发起容器启动消息.

4.4 资源分配算法的实现

实现过程中, 主要包括两个部分: 一是分布式调度器根据全局资源的信息对任务的调度, 二是资源协调器对全局资源更新消息的提交两个过程(算法 2、算法 3).

算法 2. 分布式调度器的调度过程.

输入: *queue*: 作业集合队列 *queue*,

RUs: 所有的 RU 列表;

输出: 资源分配单位集合.

```
S1:  while (job=get(queue)) do          //从队列获得作业
S2:    async res                        //获得全局集群资源状态副本
S3:    claimRes←scheduleJob(job,res)    //调度
S4:    send commit(claimRes.hosts)      //向协调器更新资源
S5:    if claimRes.job.untask!=0 do
S6:    then
S7:      add(queue,job)                 //未分配完的作业再次进入队列
S8:    end
S9:    将资源发送给作业
S10:   sleep
S11: end
```

Function *scheduleJob*(*job*,*res*)

```
S1:  foreach (task in job.untasks) do
S2:    foreach (hostres in res) do
S3:      call jobWait(task,hostres)     //执行算法 1
S4:      hosts←hosts+hostres.host
S5:      job.untasks←job.untasks−task
S6:    end
S7:  end
S8:  claimRes←(hosts,job)
S9:  return claimRes
```

算法 3. 协调器的仲裁过程 commit.

输入: *hostres*: 资源分配集合,

RUs: 所有的 RU 列表;

输出: 更新资源.

```
S1:  for (ares:hostres) do              //对于每一个分配的资源单元
S2:    if mac=host and mac.availres≥ares then
S3:      update mac.availres            //没有冲突, 更新可用资源
S4:    else
S5:      add(wqueue,hostres)            //有冲突, 插入等待分配队列
S6:    end
S7:  end
```

对于分布式调度器, 遍历作业队列中的每个作业, 算法 2 的 S1, S2 是从全局资源状态获得一个副本. S3 为作业分配资源. S4 将分配后的结果提交到协调器. S5 为判断返回的结果. 如果需要的资源还没有满足, 当前作业加入作业队列(算法 2 的 S7), 等待下一次调度.

函数 *scheduleJob* 中, 针对作业中的每个容器, 从批处理队列的可用资源中获得等待时间最短的队列信息. 函数遍历每个队列, 调用算法 1 来获得最佳的可用计算资源.

算法用来检测资源分配的冲突. 协调器接收到资源更新请求后, 检查当前集群计算节点的资源是否满足任务分配的资源(算法 3 的 S2): 如果满足, 则不产生冲突, 更新全局集群资源状态(算法 3 的 S3); 如果不满足, 则将当前的分配请求插入等待队列(算法 3 的 S5), 由协调器在下一次心跳信息到达时再分配.

流处理作业依据流处理队列的状态, 采用先来先服务方式, 这里就不详细说明.

4.5 全局资源分配策略

- 流处理作业之间的资源策略.

对于流处理作业, 必须保证作业的 SLO, 需要一次性分配足够的资源. 为防止流处理作业资源分配冲突, 采用悲观封锁协议, 减少流处理作业之间的资源分配冲突. 按照先来先服务分配计算资源, 资源不足时, 不许可其他作业的注册.

- 批处理作业之间的资源策略.

对于批处理作业, 通过集群系统总的资源数量以及运行中的作业数量, 调度器计算出作业的资源份额. 例如: 假如集群系统中有 Q 个 GPU 设备, 集群系统有 K 个批处理作业. 对于作业 j, 其应该分配得到的 GPU 数量为 $A_j^* = \min(\lfloor Q/K \rfloor, N_j)$, 其中, N_j 代表作业 j 中包含的未执行的任务数量. 如果 K 个作业的基本资源份额之和小于 GPU 设备数量, 即 $\Sigma_j A_j^* < Q$, 说明存在空闲的 GPU 资源, 可以将空闲资源均匀地分配给有等待任务的作业. 假如各个作业分配到的最后结果是 A_j, 并且满足条件 $\Sigma_j A_j = \min(Q, \Sigma_j N_j)$ 时, 分配过程就停止. 协调器计算出每个作业使用的资源份额后, 在调度过程中按照这个最大的资源份额来确认资源请求, 以此来保证资源在各个作业的公平分配.

如果当前队列有长时间执行的任务, 导致任务长时间等待, 而其他节点有空闲资源的情况, HRM 的全局资源策略通过队列之间的负载平衡来实现任务的迁移[59].

5 基于 Spark 框架的应用感知策略

在提交作业时, 如果批处理作业和流处理作业使用相同的编程框架, 那么资源管理器很难选择调度策略. 要么人工设置作业的类型, 即设置批处理作业或者流处理作业, 调度器根据作业类型采用不同的调度策略; 另外一种方式是自适应感知作业的类型, 然后使用不同的调度策略. 前者简单, 后者复杂.

为了实现自适应感知, 调度器需要获得作业的一些特征, 例如数据批次大小、作业延迟要求、作业触发时机等特征. 调度器收集这些特征, 然后进行决策.

目前, 论文只实现了对 Spark 执行引擎的自适应感知. Spark 作为数据流计算模型的编程框架, Spark Structure Stream 中的无界表结构, 使流处理作业和批处理作业共用一套代码, 因此 Spark 计算框架不但适合批处理作业, 也适用于流处理作业. 为了实现 HRM 中调度器的批流感知, 对 spark 的触发器进行了改造, 当应用程序的触发方式为执行一次时, 向 HRM 发送批处理特征; 当触发方式为按照窗口执行多次时, 向 HRM 发送流处理特征. HRM 通过计算框架的触发方式来感知作业的类型, 对流处理作业采用悲观封锁协议分配资源, 对于批处理作业采用乐观封锁协议分配资源.

6 实验评价

系统在一个集群上进行实验, 集群中包含 8 台 NF5468M5 服务器, 作为计算节点; 1 台中科曙光服务器 620/420, 作为调度节点. 每个服务器节点包含 2 个 Xeon2.1 处理器, 每个处理器包含 8 个核, 32 GB DDR4 内存, 2 块 RTX2080TI GPU 卡, 10 GB 显存. 集群包含 1 台 AS2150G2 磁盘阵列. 服务器操作系统为 Ubuntu 7.5.0, CUDA 版本为 10.1.105, 采用 C++11 作为编程语言. 为了测试框架的性能, 流处理和批处理作业使用 Spark 计算框架开发, 采用 WordCount, Sort, TPC-H 等负载, GPU 应用主要使用 YOLOv3 为模型的视频流检测, 使用 Java native 来实现 kernel 函数的调用.

6.1 基于模拟器的性能评价

6.1.1 模拟机器数量的设置

模拟测试中, 使用一台机器模拟 Negotiator, 其余 7 台机器模拟工作节点. 每台工作节点机器设置 150 个

队列, 一个队列模拟一台物理机器, 称为逻辑机器. 队列的 runlimit 设置为 16, 代表每个逻辑机器拥有 16 个 CPU 核. 这样的话, 一台物理机器就可以模拟 150 台逻辑机器, 7 台物理机器模拟 1 050 台逻辑机器, 每个逻辑机器上模拟 16 个 CPU 核, 那么整个模拟系统的 CPU 计算资源为 16 800 个 CPU 核.

6.1.2 模拟作业的设置

作业按照一定的时间间隔向集群资源调度器注册并请求资源. 每个作业中设置数量不同的任务数. 当作业注册后, 调度分配计算资源, 并启动执行器. 执行器启动后, 向作业请求任务并执行任务, 任务的执行采用 *sleep* 函数, 等待时间随机生成. 每个作业提交时间、作业中每个任务的提交时间、任务的实际开始、任务的实际结束时间都会被记录下来. 作业的实际执行时间是第一个任务开始到最后一个任务的结束.

6.1.3 模拟器评价结果

为了说明 HRM 的调度性能, 与集中式调度器对比. 集中式调度器一次性提交 60 个、120 个以及 180 个作业; 使用 HRM 时, 使用了 6 个分布式调度器上, 每个调度器提交 10 个、20 个以及 30 个作业. 其中, 每个作业包含 40 个容器(执行器), 容器中任务的执行时间使用固定值或者随机值. 两组不同的测试中, 测试作业的调度延迟时间以及作业完成时间. 每一组至少测试 3 次, 统计平均值, 其结果如图 6 所示.

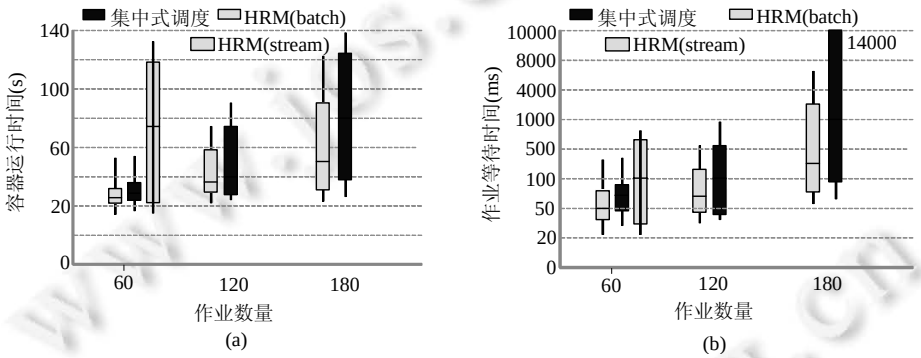


图 6 不同作业数量时的性能

从图 6 可以看出: 当作业数量为 60 时, 集中式调度器与 HRM 资源调度框架相比, 作业完成时间与作业的调度延迟基本相同, 差别不大, 作业完成时间大约 25 s, 作业的调度延迟 50 ms 左右; 当作业数量为 120 时, 两者的作业调度延迟就存在一定的差距, 基本上有 40 ms 误差; 当作业数量为 180 时, 不论从作业完成时间还是作业的调度延迟都出现了较大的变化, 启动集中式调度器的作业的调度延迟最大值达到 14 s, 而 HRM 的作业的调度延迟最大值只有 2.5 s. 因此, 随着作业数量的增加, HRM 在作业调度延迟方面有大幅度的减少.

图 6 中的 HRM 流处理由于需要一次性分配作业需要的最大资源, 而且采用先进先出方式, 所以其作业的调度延迟以及作业完成时间与整个资源的大小相关, 我们只测试了 60 个作业时的作业的调度延迟. 当作业数量为 120 个以及 180 个时, 由于实际中企业不可能同时运行太多的流处理作业, 所以未进行性能对比的评价. 但是从其等待时间看, 其比集中式调度器还长. 从这里也可以看出: 流处理作业需要资源调度器为一个作业分配足够的资源后, 才开始另外一个作业的资源分配.

当大量作业注册到调度器上的时候, 作业首先进入调度器的等待队列, 等待调度器进行资源调度. 因此, 可以用某一时刻下等待调度的作业数量与注册的所有作业数量的比值来评价调度器的繁忙程度, 比值越大, 说明需要该调度器调度的作业的数量越多, 则说明该调度器越繁忙. 为了评价调度器的繁忙程度, 可以加大作业提交的数量, 通过作业数量的不断变化来评价调度器的特性, 包括作业的调度延迟以及调度器繁忙程度的变化情况. 图 7 给出了调度器的性能测试. 图 7(a)中, 随着作业达到频率的提高, 集中式的调度延迟加大, 当作业达到频率提高 10 倍时, 调度器延迟达到 2 倍以上. 图 7(b)中, 测试 HRM 与集中式调度器的繁忙程度. 从图中曲线看出: 随着作业达到速率的增长, 集中式调度器繁忙程度接近于线性增长, 而 HRM 调度的繁忙程度则增长缓慢.

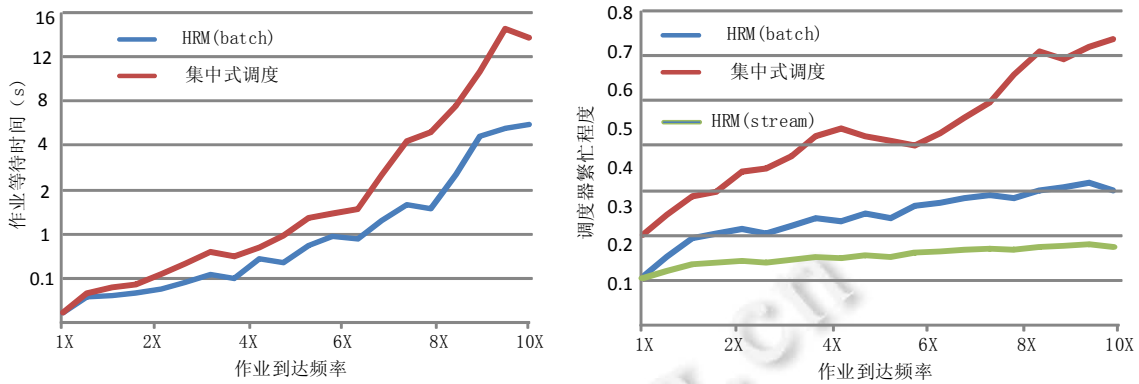


图 7 调度器性能

图 8 给出了 HRM 资源分配过程中，随着作业数量的增大，乐观封锁协议产生的资源分配冲突的比例. 从图中可以看出: 随着作业数量的增大，乐观封锁协议带来的分配冲突也增加; 当达到 10 倍数量时，容器资源分配冲突最高达到 45%左右.

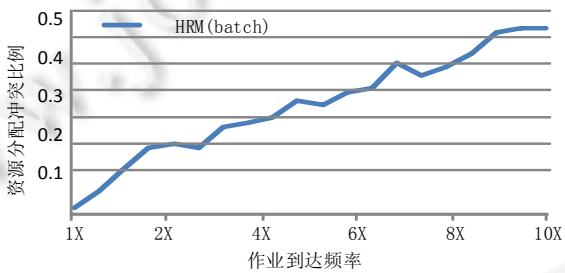


图 8 资源分配冲突比例

6.2 真实负载的性能分析

为了测试真实负载情况下 HRM 的性能，使用了 4 类负载: 第 1 类是 CPU 批处理作业，由 WordCount, Sort 以及 TPC-H 查询组成; 第 2 类是 CPU 流处理作业，为 Top-*k* 查询; 第 3 类是 GPU 批处理作业，为 *k*-means 聚类计算; 第 4 类为 GPU 流处理作业，为基于 YOLOv 3 的视频流检测. 所有的应用程序都使用 Spark 编程框架来实现，采用 mesos 和 Yarn 为基准调度框架.

6.2.1 基准性能评价

为了测试 HRM 的基准性能,与 mesos 资源调度框架的 RDF、Yarn 的容量调度以及 Yarn 的公平调度策略进行了对比，在集群数量有限的情况下, HRM 的性能基本与 mesos 相当，比 Yarn 的两种调度策略有提高.

图 9 给出了 CPU 批处理/流处理作业执行时间的对比. 作业从提交到完成所花费的时间叫完成时间. 图 9(a)代表不同作业的完成时间，从图中可以看出: HRM 与 mesos 的性能基本相同，差别不大. 原因在于集群中资源数量有限, mesos 的悲观资源分配协议与 HRM 的乐观分配协议运行时差别不大. 但是 TPC-Q9, TPC-Q10, TPC-Q21 的差别有点大，主要是这些作业运行中需要的执行器数量较多，乐观分配协议的延迟少的原因. 对比 Yarn 的容量调度以及公平调度, mesos 与 HRM 都具有一定的优势. 分析原因在于: HRM 以及 mesos 都采用二级调度方式，而 Yarn 采用的单一路径的集中式调度，所以调度延迟相对大一些.

图 9(b)是流处理作业 Top-*K* 的性能数据，我们选择了一段时间内每个微批次处理时间. 从图中可以看出: 流处理作业每个微批次运行时间大致相同，没有差别. 其原因是流处理过程中不存在其他作业请求资源，因此, HRM 与其他调度策略差别不大.

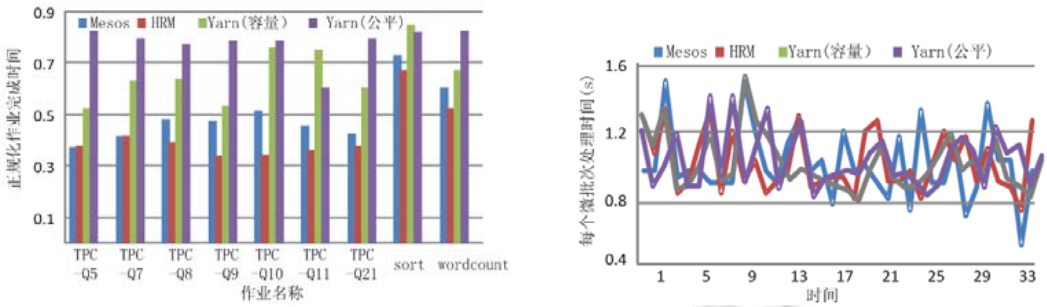


图 9 CPU 批/流处理作业完成时间

对于 GPU 流处理以及批处理，由于 mesos 目前只支持整个 GPU 设备一次性分配给一个作业，测试的集群只有 16 块 GPU 设备，当 HRM 也采用粗粒度方式调度批处理与流处理作业时，其性能基本一致，所以论文没有给出基准测试数据的对比.

6.2.2 批处理作业与流处理作业混合运行性能

流处理和批处理作业同时运行时，流处理作业和批处理作业必然同时向集群资源管理器请求计算资源. 图 10 给出了流处理作业和批处理作业同时运行时流处理作业的性能，图中的纵坐标代表每个微批次的处理时间，横坐标代表批处理作业的数量.

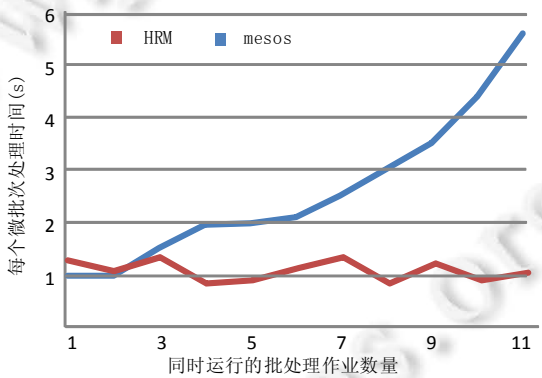


图 10 流处理与批处理作业同时运行性能

从图 10 可以看出:

- 当同时运行的批处理作业数量较少(1–3 个)，流处理作业在 HRM 和 mesos 上，对每个微批次数据的处理时间大致相同;
- 当同时运行的批处理作业数量扩大(5–9 个), HRM 资源管理下，流处理作业的每个微批次数据处理时间变化不大; 而 mesos 资源管理下，其微批次数据处理时间增加了 2–3 倍. 其原因在于: 批处理和流处理作业开始竞争资源，当同时运行的批处理作业再增加，可以看出，流处理作业每个微批次数据处理时间增加的更大，这说明流处理作业获得的计算资源数量大大减少. 由于 HRM 采用队列堆叠技术，流处理和批处理作业分别使用各自的队列，虽然同时运行的批处理作业数量增加，但是流处理作业的队列的优先级较高，其能够充分获得计算资源，所以每个微批次数据处理时间变化不大.

6.2.3 队列排队机制测试

HRM 对每个物理资源设置一个排队队列，当前容器结束，排队中的容器立即开始执行，就可以减少资源调度的延迟. 图 11 中给出了不同的排队长度时，各种批处理作业的运行时间. 图中的横坐标代表不同的批处

理作业, 纵坐标代表经过正规化后的各个作业的执行时间. L=1 代表队列中的排队长度为 1, L=2 代表队列中的排队长度为 2, L=3 代表队列中的排队长度为 3. 从图中可以看出: 当队列的排队长度为 2 时, 其性能最好, 可以减少作业的运行时间大约 15%左右; 当队列的排队长度为 3 时, 其中 5 个作业的性能反而下降. 这说明排队长度较大时, 一些作业的容器在队列中出现延迟等待的情况, 导致这些作业的执行时间变长. 因此, 设置合理的队列排队长度, 有利于减少反馈延迟.

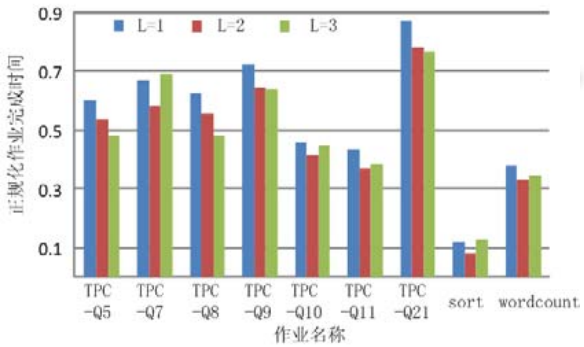


图 11 容器排队性能

6.2.4 GPU 作业的细粒度资源分配

HRM 通过对 GPU 设备的显存进行管理, 在每个 Executor 不超过显存最大值的情况下, 能够在不同的作业之间共享同一个 GPU 设备, 提高 GPU 利用率. 表 1 给出了使用 YOLOv3 进行图片检测时, 粗粒度和细粒度性能评价表, 其中: 并行度为 1 代表粗粒度, 作业独占 GPU 设备; 并行度为 2 代表同一个 GPU 设备可以运行 2 个 Executor; 并行度为 3 代表同一个 GPU 设备可以运行 3 个 Executor. 图片的总数量为 720 个. 同时运行 3 个图片检测作业.

表 1 GPU 粗细粒度性能评价表

并行度	最大显存占用(MB)	GPU 峰值利用率(%)	处理时间(s)
1	2 865	24	151.3
2	5 720	61	103.2
3	8 575	73	69

从表 1 的数据看: 随着资源分配粒度变小, 性能提升明显升高. 粗粒度时, 处理 720 张图片需要 151.3 s; 当粒度增加到 3 时, 处理时间减少为 69 s. 当并行度为 2 时, 性能提升 32%; 当并行度为 3 时, 性能提升达到 50%以上. 从 GPU 的资源利用率看: 随着并行度的增加, GPU 的资源利用率从 24%上升到 73%左右. 因此, 在 GPU 显存许可范围内, 增加资源分配的粒度不但可以提高作业的性能, 也能提高 GPU 资源的利用率.

6.2.5 CPU 批处理作业与 GPU 流处理作业混合性能

为了测试 GPU 流处理和 CPU 批处理作业混合时的资源利用率, 测试中, GPU 流处理为使用 darknet 检测 1 000 张图片, CPU 批处理为 WordCount 作业, 数据大小为 25 GB. 测试时间大约为 230 s 左右, 我们计算了 GPU 利用率和 CPU 利用率的平均值, 其结果如图 12 所示.

图 12 中, 横坐标代表 3 种不同的作业类型: detection 是单独检测 1 000 张图片时, GPU 的利用率和 CPU 利用率平均值; wordcount 是单独计算 25 GB 数据时的 CPU 利用率; decection&wordcount 是混合执行两种作业时的 GPU 利用率和 CPU 利用率. 从图中可以看出:

- 单独执行 detection 时, GPU 利用率为 67%左右; CPU 利用率较低, 大约为 35%左右;
- 单独执行 wordcount 时, 其 CPU 利用率在 78%左右;
- 混合执行 decection 和 wordcount 时, 其 GPU 利用率变化不大, 但是 CPU 利用率提高到 93%左右.

因此, 使用 HRM 资源管理系统, 对于 CPU 任务与 GPU 任务混合场景, 在流处理作业的不受影响的情况

下, CPU 利用率会得到进一步的提高.

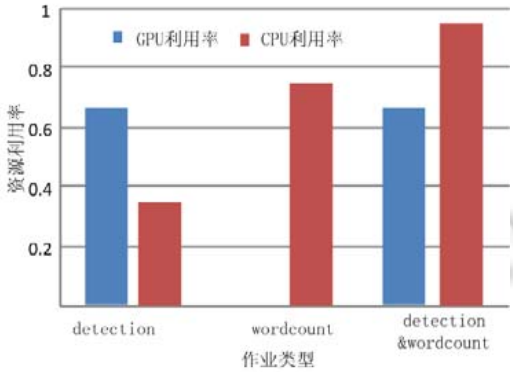


图 12 混合作业时资源利用率

7 结 论

随着 GPU 设备在大数据分析中的广泛应用, 为批处理/流处理融合的应用提供 CPU-GPU 资源共享也越来越重要. HRM 满足这些要求, 采用应用程序可感知的调度策略, 为流处理作业实现低延迟, 为批处理作业提供高通量处理. 在 CPU-GPU 资源调度方面, 利用队列机制将物理资源虚拟化并提供抢占式调度策略, 满足流处理作业实时性要求. 队列的使用, 也减少了反馈延迟问题.

References:

[1] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In: Proc. of the 6th Conf. on Symp. on Operating Systems Design & Implementation (OSDI 2004), Vol.6. USENIX Association, 2004. 10. [doi: 10.1145/1327452.1327492]

[2] https://github.com/nathanmarz/storm

[3] Murray DG, Mcsherry F, Isaacs R, *et al*. Naiad: A timely dataflow system. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP 2013). New York: Association for Computing Machinery, 2013. 439–455. [doi: 10.1145/2517349.2522738]

[4] Zaharia M, Das T, Li HY, *et al*. Discretized streams: Fault-tolerant streaming computation at scale. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP 2013). New York: Association for Computing Machinery, 2013. 423–438. [doi: 10.1145/2517349.2522737]

[5] Gates AF, Natkovich O, Chopra S, *et al*. Building a high-level dataflow system on top of Map-reduce: The pig experience. Proc. of the VLDB Endowment, 2009, 2(2): 1414–1425. [doi: 10.14778/1687553.1687568]

[6] https://github.com/GoogleCloudPlatform/ DataflowJavaSDK

[7] Google. Google cloud DataFlow. 2015. https://cloud.google.com/dataflow/

[8] Akidau T, Bradshaw R, Chambers C, *et al*. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out- of-order data processing. Proc. of the Vldb Endowment, 2015, 8(12): 1792–1803. [doi: 10.14778/2824032.2824076]

[9] Pishgoo B, Azirani AA, Raahemi B. A hybrid distributed batch-stream processing approach for anomaly detection. Information Sciences, 2020, 543: 309–327.

[10] Apache. Apache flink. 2014. http://flink.apache.org/

[11] Zaharia M, Chowdhury M, Das T, *et al*. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Conf. on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2012. 2. [doi: 10.5555/2228298.2228301]

[12] Karanasos K, Raol S, Curino C, *et al*. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In: Proc. of the 2015 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2015. 485–497.

[13] Delgado P, Dinu F, Zwaenepoel W, *et al*. Hawk: Hybrid datacenter scheduling. In: Proc. of the 2015 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2015. 499−510.

[14] Delgado P, Didona D, Dinu F, *et al*. Job-aware scheduling in Eagle: Divide and stick to your probes. In: Proc. of the 7th ACM Symp. on Cloud Computing. New York: ACM, 2016. 497−509. [doi: 10.1145/2987550.2987563]

[15] Vavilapalli VK, Murthy AC, Douglas C, *et al*. Apache hadoop Yarn: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. New York: ACM, 2013. 1−16. [doi: 10.1145/2523616.2523633]

[16] Hindman B, Konwinski A, Zaharia M, *et al*. 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. Berkeley: USENIX Association, 2011. 295−308. [doi: 10.5555/1972457.1972488]

[17] Burns B, Grant B, Oppenheimer D, *et al*. Borg, omega, and kubernates. Communications, 2016, 59(5): 50−57. [doi: 10.1145/2890784]

[18] Cuda_wrapper project at SourceForge website. 2010. https://sourceforge.net/projects/cudawrapper/files/

[19] Hadoop capacity scheduler. https://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

[20] Hadoop fair scheduler. https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

[21] Ghodsi A, Zaharia M, Hindman B, *et al*. Dominant resource fairness: Fair allocation of multiple resource types. In: Proc. of the 8th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2011. 323−336. [doi: 10.5555/1972457.1972490]

[22] Schwarzkopf M, Konwinski A, Abd-El-Malek M, *et al*. Omega: Flexible, scalable schedulers for large compute clusters. In: Proc. of the 8th ACM European Conf. on Computer Systems. New York: ACM, 2013. 351−364. [doi: 10.1145/2465351.2465386]

[23] Ousterhout K, Wendell P, Zaharia M, *et al*. Sparrow: Distributed, low latency scheduling. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. New York: ACM, 2013. 69−84. [doi: 10.1145/2517349.2522716]

[24] Boutin E, Ekanayake J, Wei L, *et al*. Apollo: Scalable and coordinated scheduling for cloud- scale computing. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2014. 285−300. [doi: 10.5555/2685048.2685071]

[25] Tumanov A, Cipar J, Ganger GR, *et al*. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In: Proc. of the 3rd ACM Symp. on Cloud Computing. New York: ACM, 2012. 1−7. [doi: 10.1145/2391229.2391254]

[26] Yao Y, Tai JZ, Sheng B, *et al*. LsPS: A job size-based scheduler for efficient task assignments in Hadoop. IEEE Trans. on Cloud Computing, 2015, 3(4): 411−424.

[27] Goder A, Spiridonov A, Wang Y. Bistro: Scheduling data-parallel jobs against live production systems. In: Proc. of the 2015 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2015. 459−471.

[28] Grandl R, Chowdhury M, Akella A, *et al*. Altruistic scheduling in multi-resource clusters. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2016. 65−80.

[29] Ghodsi A, Zaharia M, Shenker S, *et al*. Choosy: Max-min fair sharing for datacenter jobs with constraints. In: Proc. of the 8th ACM European Conf. on Computer Systems. New York: ACM, 2013. 365−378. [doi: 10.1145/2465351.2465387]

[30] Mao HZ, Schwarzkopf M, Bojja-Venkatakrishnan S, *et al*. Learning scheduling algorithms for data processing clusters. In: Proc. of the ACM Special Interest Group on Data Communication. New York: ACM, 2019. 270−288. [doi: 10.1145/ 3341302.3342080]

[31] Zaharia M, Borthakur D, Sen Sarma J, *et al*. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: Proc. of the 5th ACM European Conf. on Computer Systems. New York: ACM, 2010. 265−278. [doi: 10.1145/1755913.1755940]

[32] Ananthanarayanan G, Ghodsi A, Shenker S, *et al*. Effective straggler mitigation: Attack of the clones. In: Proc. of the 10th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2013. 185−198. [doi: 10.5555/2482626.2482645]

[33] Isard M, Prabhakaran V, Currey J, *et al*. Quincy: Fair scheduling for distributed computing clusters. In: Proc. of the 22nd ACM Symp. on Operating Systems Principles. New York: ACM, 2009. 261−276. [doi: 10.1145/1629575.1629601]

[34] Gog I, Schwarzkopf M, Gleave A, *et al*. Firmament: Fast, centralized cluster scheduling at scale. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2016. 99−115.

[35] Henzinger TA, Singh V, Wies T, *et al*. Scheduling large jobs by abstraction refinement. In: Proc. of the 6th ACM European Conf. on Computer Systems. New York: ACM, 2011. 329−342. [doi: 10.1145/1966445.1966476]

[36] Zhang Z, Li C, Tao YY, *et al*. Fuxi: A fault-tolerant resource management and job scheduling system at Internet scale. Proc. of the VLDB Endowment, 2014. 1393−1404. [doi: 10.14778/2733004.2733012]

[37] Grandl R, Kandula S, Rao S, *et al*. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2016. 81−97. [doi: 10.5555/3026877.3026885]

[38] Ferguson AD, Bodik P, Kandula S, *et al*. Jockey: Guaranteed job latency in data parallel clusters. In: Proc. of the 7th ACM European Conf. on Computer Systems. New York: ACM, 2012. 99−112. [doi: 10.1145/2168836.2168847]

[39] Venkataraman S, Panda A, Ananthanarayanan G, *et al*. The power of choice in data-aware cluster scheduling. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2014. 301−316.

[40] Zaharia M, Konwinski A, Joseph AD, *et al*. Improving MapReduce performance in heterogeneous environments. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 29−42. [doi: 10.5555/1855741.1855744]

[41] Abdu-Jyothi S, Curino C, Menache I, *et al*. Morpheus: Towards automated SLOs for enterprise clusters. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2016). USENIX Association, 2016. 117−134. [doi: 10.5555/3026877.3026887]

[42] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware cluster management. In: Proc. of the 19th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014. 127−144. [doi: 10.1145/2644865.2541941]

[43] Suzuki Y, Kato S, Yamada H, *et al*. GPUvm: GPU virtualization at the hypervisor. IEEE Trans. on Computers, 2016. 2752−2766.

[44] Kato S, McThrow M, Maltzahn C, *et al*. Gdev: First-class GPU resource management in the operating system. In: Proc. of the 2012 USENIX Conf. on Annual Technical Conf. (USENIX ATC 2012). USENIX Association, 2012. 37.

[45] Krieder SJ, Wozniak JM, Armstrong T, *et al*. Design and evaluation of the gemtc framework for GPU-enabled many-task computing. In: Proc. of the 23rd Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC 2014). New York: Association for Computing Machinery, 2014. 153−164. [doi: 10.1145/2600212.2600228]

[46] Yeh T, Sabne A, Sakdhnagool P, *et al*. Pagoda: Fine-grained GPU resource virtualization for narrow tasks. In: Proc. of the 22nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2017). New York: Association for Computing Machinery, 2017. 221−234. [doi: 10.1145/3155284.3018754]

[47] Shao J, Ma J, Li Y, *et al*. GPU scheduling for short tasks in private cloud. In: Proc. of the 2019 IEEE Int'l Conf. on Service-Oriented System Engineering (SOSE). IEEE, 2019. 215−220.

[48] Shirahata K, Sato H, Matsuoka S. Hybrid map task scheduling for GPU-based heterogeneous clusters. In: Proc. of the 2010 IEEE 2nd Int'l Conf. on Cloud Computing Technology and Science (CLOUDCOM 2010). IEEE Computer Society, 2010. 733−740. [doi: 10.1109/CloudCom.2010.55]

[49] Fan Z, Qiu F, Kaufman A, *et al*. GPU cluster for high performance computing. In: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing (SC 2004). IEEE Computer Society, 2004. 47. [doi: 10.1109/SC.2004.26]

[50] Verma A, Pedrosa L, Korupolu M, *et al*. Large-scale cluster management at Google with Borg. In: Proc. of the 10th ACM European Conf. on Computer Systems, Vol.18. New York: ACM, 2015. 1−17. [doi: 10.1145/2741948.2741964]

[51] Fukutomi D, Iida Y, Azumi T, *et al*. GPUhd: Augmenting YARN with GPU resource management. In: Proc. of the Int'l Conf. on High Performance Computing in Asia-Pacific Region (HPC Asia 2018). New York: Association for Computing Machinery, 2018. 127−136.

[52] The Apache Software Foundation. Add GPU as a resource type for scheduling. 2016. https://issues.apache.org/jira/browse/YARN-5517

[53] Gu JZ, Liu H, Zhou YF, *et al*. DeepProf: Performance analysis for deep learning applications via mining GPU execution patterns. arXiv:1707.03750v1, 2017.

[54] Gu JC, Chowdhury M, Shin KG, *et al*. Tiresias: A GPU cluster manager for distributed deep learning. In: Proc. of the 16th USENIX Conf. on Networked Systems Design and Implementation (NSDI 2019). USENIX Association, 2019. 485−500.

[55] Kshiteej M, Arjun S, Arjun B, *et al*. Themis: Fair and efficient GPU cluster scheduling. In: Proc. of the USENIX Annual Technical Conf. 2019. Boston: MA 02199, 2019. 947−960.

[56] Stuart JA, Owens JD. Multi-GPU MapReduce on GPU clusters. In: Proc. of the 2011 IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS 2011). IEEE Computer Society, 2011. 1068−1079. [doi: 10.1109/IPDPS.2011.102]

[57] Liu JQ, Hegde N, Kulkarni M. Hybrid CPU-GPU scheduling and execution of tree traversals. In: Proc. of the 2016 Int'l Conf. on Supercomputing (ICS 2016). New York: Association for Computing Machinery, 2016. 1−12. [doi: 10.1145/2925426.2926261]

[58] Rasley J, Karanasos K, Kandula S, *et al*. Efficient queue management for cluster scheduling. In: Proc. of the 11th European Conf. on Computer Systems (EuroSys 2016). New York: Association for Computing Machinery, 2016. 1−15. [doi: 10.1145/2901318.2901354]

[59] Tang XC. The research and implementation of job managementsystem based on cluster computing technology [Ph.D. Thesis]. Xi'an: Northwestern Polytechnical University, 2002. (in Chinese with English abstract).

附中文参考文献:

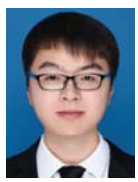
[59] 汤小春. 基于集群技术的作业管理系统的研究与实现[博士学位论文]. 西安: 西北工业大学, 2002.

汤小春(1969－), 男, 博士, 副教授, 主要研究领域为大数据计算, 大图数据挖掘, 集群资源管理.

赵全(1997－), 男, 硕士, 主要研究领域为大数据计算, 集群资源管理.

符莹(1996－), 女, 硕士, 主要研究领域为大数据计算, 集群资源管理.

朱紫钰(1996－), 女, 硕士, 主要研究领域为大数据计算, 集群资源管理.



丁朝(1996－), 男, 硕士, 主要研究领域为大数据计算, 集群资源管理.

胡小雪(1996－), 女, 硕士, 主要研究领域为大数据计算, 集群资源管理.

李战怀(1961－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为海量数据管理, 大数据计算.