软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(12):4429−4451 [doi: 10.13328/j.cnki.jos.006362]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

数据密集作业在 GPU 集群上的调度算法研究*

汤小春, 朱紫钰, 毛安琪, 符 莹, 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: 数据密集型作业包含大量的任务, 使用 GPU 设备来提高任务的性能是目前的主要手段. 但是, 在解决数据密集型作业之间的 GPU 资源公平共享以及降低任务所需数据在网络间的传输代价方面, 现有的研究方法没有综合考虑资源公平与数据传输代价的矛盾. 分析了 GPU 集群资源调度的特点, 提出了一种基于最小代价最大任务数的 GPU 集群资源调度算法, 解决了 GPU 资源的公平分配与数据传输代价较高的矛盾. 将调度过程分为两个阶段: 第 1 阶段为各个作业按照数据传输代价给出自己的最优方案; 第 2 阶段为资源分配器合并各个作业的方案, 按照公平性给出全局的最优方案. 首先, 给出了 GPU 集群资源调度框架的总体结构, 各个作业给出自己的最优方案, 资源分配进行全局优化; 第二, 给出了网络带宽估计策略以及计算任务的数据传输代价的方法; 第三, 给出了基于 GPU 数量的资源公平分配的基本算法; 第四, 提出了最小代价最大任务数的资源调度算法, 描述了资源非抢夺、抢夺以及不考虑资源公平策略的实现策略; 最后, 设计了 6 种数据密集型计算作业, 对所提出的算法进行了实验. 通过实验验证, 最小代价最大任务数的资源调度算法对于资源公平性能够达到 90%左右, 同时亦能保证作业并行运行时间最小.

关键词: GPU; 数据密集型; 最小代价; 公平性; 数据本地化
中图法分类号: TP301

中文引用格式: 汤小春, 朱紫钰, 毛安琪, 符莹, 李战怀. 数据密集作业在 GPU 集群上的调度算法研究. 软件学报, 2022, 33(12): 4429–4451. http://www.jos.org.cn/1000-9825/6362.htm

英文引用格式: Tang XC, Zhu ZY, Mao AQ, Fu Y, Li ZH. Algorithm of Scheduling for Data-intensive Computing Operations onto GPU Cluster. Ruan Jian Xue Bao/Journal of Software, 2022, 33(12): 4429–4451 (in Chinese). http://www.jos.org.cn/1000-9825/6362.htm

Algorithm of Scheduling for Data-intensive Computing Operations onto GPU Cluster

TANG Xiao-Chun, ZHU Zi-Yu, MAO An-Qi, FU Ying, LI Zhan-Huai

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

Abstract: Data-intensive tasks include a large number of tasks. Using GPU devices to improve the performance of tasks is the main method currently. However, in the case of solving the fair sharing of GPU resources between data-intensive tasks and reducing the cost of data network transmission, the existing research methods do not comprehensively consider the contradiction between resource fairness and data transmission costs. The study analyzes the characteristics of GPU cluster resource scheduling, and proposes an algorithm based on the minimum cost and the maximum number of tasks in GPU cluster resource scheduling. The method can solve the contradiction between the fair allocation of GPU resources and the high cost of data transmission. The scheduling process is divided into two stages. In the first stage, each job gives its own optimal plan according to the data transmission costs, and in the second stage, the resource allocator merges the plan of each job. Firstly, the study gives the overall structure of the framework, and the source allocator works globally after each job giving its own optimal plan. Secondly, the network bandwidth estimation strategy and the method of computing the data transmission cost of the task are given. Thirdly, the basic algorithm for the fair allocation of resources based on the number of GPUs is given. Fourthly, the scheduling algorithm with the smallest cost and the largest number of tasks is proposed, which describing the

* 基金项目: 国家重点研发计划(2018YFB1003400)
收稿时间: 2020-05-10; 修改时间: 2020-11-30; 采用时间: 2021-04-30; jos 在线出版时间: 2021-05-20

implementation strategies of resource non-grabbing, robbing and resource fairness strategies. Finally, six data-intensive computing tasks are designed, and the algorithm proposed in the study is tested, and the experiments verifies that the scheduling algorithm can achieve about 90% of resource fairness, while also ensuring that the parallel operation time of jobs is minimized.

Key words: GPU; data intensive; minimum cost; fairness; data localization

近年来, 随着 IoT 数据规模的增长以及相互连接设备数量的扩大, 在企业领域、医学研究、大规模科学研究等领域涌现出许多大规模的数据密集型应用程序. 在这些大规模数据处理的方法上, 存在以下两种手段.

(1) GPU 集群的使用[1−3]. GPU 主要用于计算密集型任务, 不太适合大量 I/O 的数据密集型任务; 另外, 在一些涉及不规则字符串处理和文本处理领域, 则效果更差. 但是, 随着数据挖掘、生物信息学、物理模拟、模式识别、图形处理、医学影像等领域的发展[4], 其图像、视频等数据规模的增大, GPU 逐渐开始作为 CPU 加速器出现, 体现了良好的高通量处理要求. 由于这类计算中存在大量的数据计算, 伴随着用户对数据处理的效率和吞吐量要求的提高, 传统的 CPU 集群已经无法满足处理速度的要求, 新型的基于 GPU 集群的计算框架被大量使用, 成为提高数据密集型计算性能的一种手段;

(2) 数据的本地化读取. 数据密集型计算中, 数据存储位置与计算在同一个节点, 即数据的本地化读取, 这可以减少数据的传输代价. 在集群系统中, 为防止单个节点的故障而出现的数据丢失现象, 数据都存在多个副本. 在数据分析中, 充分利用副本来实现数据的本地化读取, 是另外一种提高数据密集型计算性能的手段.

在一个企业内部, 不同部门或者组织通常存在大量的数据密集型计算作业. 如果为每种作业都建立专门的 GPU 集群基础设置, 必将导致企业成本的增加. 为此, 越来越多的企业开始扩展传统的集群资源管理调度框架, 例如 Mesos[5], Yarn[6]等, 来满足不同的作业共享 GPU 集群基础设施. 但是 GPU 集群资源的管理和分配面临一些新的挑战, 例如如何保证 GPU 计算资源在不同作业之间的公平分配、如何提高资源使用率以及缩短作业的完成时间等.

为了实现多个数据密集型计算作业共享 GPU 集群资源, GPU 设备的管理和调度框架需要使用一些特别的管理机制和策略, 既要防止用户对共享 GPU 资源失去信心, 同时也需要让用户感受到共享资源带来的好处. 其中一个重要的原则是, 各个不同的作业能够得到公平的资源. 假如有 M 个作业来共享 N 个 GPU 设备, 那么集群资源管理器就需要保证每个作业分配到的计算资源不能低于 N/M. 缺乏这种特别的机制, 用户的作业要么放弃性能要求, 长时间等待集群管理器为其分配计算资源, 要么就放弃资源共享, 建立自己专门的 GPU 集群来完成作业的执行.

现有的主流资源调度框架 Yarn, Mesos, Kubernetes[7], Omega[8], Borg[9], Mercury[10], Quincy[11]等, 将 CPU、内存等作为主要资源来设计资源分配算法, 在 GPU 的管理和调度方面存在不足. 由于这些资源管理框架将 GPU 设备作为次要资源[12,13]来考虑, 所以缺乏 GPU 资源的高效分配策略, 大多数系统采用作业独占 GPU 设备的方式, 很少有共享机制的出现. 当 GPU 资源被一些作业超额预订时, 其他作业通常会经历长时间的排队, 即使是非常短的任务, 也可能需要长达数小时的等待.

GPU 是具有多个 SIMD 多处理器(SM)的多核计算机, 可以运行数千个并发线程. 不同于 CPU 的共享, 目前没有相应的硬件支持来满足 GPU 的细粒度共享[14−16], 尽管存在支持共享的软件机制, 但它们通常具有较高的系统开销, 不同作业之间共享资源具有很大的挑战性[17,18]. 在不同的作业之间共享 GPU 资源, 存在两个主要的延迟影响因素.

- 第一, 资源的公平分配. 同一计算节点上不同作业竞争 GPU 设备会导致相互干扰, 而现有的共享机制中, 不管使用短作业优先(SF)、剩余时间少者优先(SRSF), 都存在着一定的缺陷[19]. 短作业优先可能导致长作业饥饿; 若通过计算任务的剩余时间来分配资源, 由于时间预测的准确性, 常常会影响调度效果;
- 第二, 作业中的任务对数据本地化访问的要求[20]. 对于 GPU 计算任务, 程序的初始化、数据的加载、数据在不同节点之间传输过程以及结果数据写入文件等过程, 往往花费大量的时间, 特别是跨越不

© 中国科学院软件研究所 http://www.jos.org.cn

同的机架来读取数据时, 数据的加载代价更大. 如果数据与 GPU 计算任务不在同一个节点上, 或者数据存储节点与计算节点之间网络带宽无法保证, 那么任务的延迟就会增大, 甚至会出现错误. 另外, 同时运行的多个 GPU 任务跨越多个集群节点时, 它们之间必然会竞争网络带宽, 这为任务调度增加了一定的困难. 基于这些原因, 计算任务所需要的数据本地化, 是资源共享的另外一个主要调度目标.

因此, 在考虑 GPU 资源分配时, 要更多地注意公平性与数据本地化读取的矛盾. 特别重要的是: 当多个任务请求多个 GPU 资源时, 就需要考虑多个任务与多个资源的最优分配. 如果只考虑单一的调度目标, 就可能造成分配结果的次优化问题(第 2.1 节对这个问题有详细的描述).

基于上述挑战, 出现了专门为GPU集群而设计的资源调度框架, 其目的是建立计算任务与GPU设备之间的映射关系, 解决不同数据密集型作业中的任务共享 GPU 设备的问题. 本文主要研究不同作业之间的任务到 GPU 设备的共享调度问题, 不涉及任务到 CPU 的调度. 调度过程中, 综合考虑计算任务的数据本地化问题和 GPU 资源在各个计算作业之间的公平分配问题, 保证每个作业至少能够获得 N/M 的计算资源; 同时, 充分利用数据的主副本位置, 减少数据的网络传输代价, 避免出现单个作业延迟完成的现象. 为实现这个目标, 我们采用以下策略.

(1) 公平性与数据传输代价统一考虑. 如果在资源分配过程中要求数据本地化读取, 一些任务就会因为无法满足本地化而放弃一些可用资源, 从而产生延迟执行. 如果采用公平资源分配策略, 那么每个作业都会得到全部资源中的一部分, 但是获得资源的任务所需数据并不一定在本地节点, 因此等待数据的传输同样会导致任务完成时间的延长. 本文中, 在每个任务的资源描述中给出了任务数据副本的存储位置和数据的大小, 通过调度器提供的 API 以及集群资源的网络结构信息, 可以很快确定计算节点和数据存储节点之间的关系. 利用网络带宽、数据存储位置、GPU 设备所在的计算节点位置以及数据的大小等信息, 计算出每个任务在不同计算节点执行时的数据传输代价. 调度器综合考虑数据的本地化以及资源的公平分配两个因素, 通过优化的方式选择合适的资源作为任务的计算节点, 从而减少单个作业完成时间的延迟;

(2) 作业之间资源的统一分配. 已有的研究工作在考虑资源公平分配时采用固定策略分配资源, 即仅从单一因素来决定作业的资源分配顺序, 其分配结果往往不是最优的. 如 YARN, Mesos 中, 资源管理器按照作业对资源的请求量以及正在使用的资源数量, 为同时请求资源的作业确定一个资源分配次序. 作业获得资源后, 按照各个任务的本地化约束级别或者优先级别来确定任务的分配次序. 这种策略能够满足最先分配资源的作业, 但后续作业的分配要求往往无法保证. 相关研究表明: 这样的调度决策算法可能导致极低的资源利用率, 如 Twitter 平均资源利用率小于 20%; 或者引起作业性能下降 80%, 如违反优先级约束;

(3) 资源公平分配的主体为作业级别, 现有的资源管理系统主要针对 CPU 计算资源, 考虑的是 CPU 核数和内存使用量大小, GPU 则采用粗粒度独占方式, 因此通过设置资源队列, 为不同的用户分配一定的资源配额, 然后采用诸如 DRF 等公平策略实现为作业进行资源分配. 但是, 这种方式对于不同用户中作业数量、任务使用多少、GPU 以及数据大小等因素考虑较少. 本文依据全部作业的任务, 将公平性和数据访问代价同时考虑, 实现了全局的最优调度.

本文实现过程中, 将 CPU 和 GPU 计算资源分开调度, CPU 采用传统的调度策略, 在本文中不再详细叙述. 本文的主要工作是:

(1) 分析了 GPU 任务调度的特点, 实现一种针对 GPU 集群的二阶段调度框架, 确保各种不同的计算作业能够公平地使用 GPU 资源, 提高 GPU 资源的使用效率. 首先, 给出了二阶段 GPU 集群资源调度框架: 第 1 阶段为各个作业调度器根据数据传输代价最小原则生成任务调度方案, 第 2 阶段为资源分配器合并作业的调度方案, 根据公平原则和数据传输代价给出最后的决策;

(2) 本文从作业角度出发, 给出了按照作业的任务大小计算数据传输代价的方法以及按照作业数量计

© 中国科学院软件研究所 http://www.jos.org.cn

算公平分配资源份额的方法;

(3) 综合考虑所有作业的资源需求、集群系统可使用资源、作业之间资源分配的公平性以及数据传输代价等因素, 给出了作业调度器的局部流网络生成算法;

(4) 资源分配器使用计算出网络流图的最小代价最大流, 得到综合多种调度因素后的最优分配方案;

(5) 按照调度结果将作业中的任务提交到不同的计算节点, 完成任务的执行.

实验结果表明: 本系统与基于 GPU 数量的资源调度算法相比, 公平性至少提高 1.5 倍以上, 作业的运行时间提高 10%以上.

本文第 1 节描述 GPU 共享的相关研究现状. 第 2 节介绍 GPU 集群的资源调度框架. 第 3 节给出基于 GPU 数量的资源共享分配算法. 第 4 节给出基于最小代价最大任务数的 GPU 资源共享分配算法. 第 5 节设计实验过程和性能指标, 并对系统进行验证.

1 相关研究

对于 GPU 的共享, 有单个 GPU 的共享以及 GPU 集群的共享.

- 单个 GPU 共享, 主要针对 GPU 的 *kernel* 函数的共享[21], 即如何让多个 *kernel* 函数共享硬件资源, 其目标是最大化地利用 GPU 硬件资源. 文献[22]通过使用 GPU 虚拟化技术, 让 GPU 在多个不同的 VMs 上共享, 提高 GPU 利用率. 文献[23]通过合并多个 *kernel* 函数来共享 GPU 资源, 提高 GPU 利用率. 文献[24]通过对 GPU 进行物理分片, 以便达到多个 *kernel* 函数对 GPU 共享的目标. 文献[25]通过时间片策略, 实现具有抢夺策略的多个 *kernel* 函数对 GPU 的共享;
- 对于 GPU 集群资源的共享, 一部分研究不同作业对 GPU 资源的共享, 另外一部分研究单个作业的不同任务如何使用 GPU 集群资源. 文献[26]提出了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. 其采用 Master/Slave 结构, 在作业运行之前, 为每个不同的作业分配固定的 GPU 资源, 作业在运行中不能动态扩大或者缩小. 为了防止多个用户共享 GPU 产生的冲突, 通过调用 CUDA 封装库[27]的方式来访问 GPU 资源, 作业分配到对应 GPU 设备后, 就独占该 GPU, 其主要应用于 HPC 上, 使用方式有两种.
    - 第 1 种是配置所有的 GPU 作为单个定制资源. 管理者只添加一个定制资源, 所有的 GPU 使用相同的优先级别, 也不绑定作业到任何特定的 GPU 上;
    - 另一种是管理者将每个 GPU 设备当作一个 vnode, 分配给不同的作业. 这种方式的优点是作业调度时间短, 作业之间相互干涉小; 缺点是 GPU 资源无法动态共享, 可能会导致部分 GPU 的负载较重, 而部分 GPU 负载较轻.

Kubernetes 采用容器机制对 GPU 进行隔离, 其采用固定策略的资源分配方法. 另外, 当容器不释放资源时, 无法实现 GPU 计算资源的共享. 基于容器的调度框架 Kubernetes 虽然实现了 GPU 资源的管理, 但是很难满足作业之间对 GPU 计算资源进行公平分配的要求与任务的数据本地化读取的要求.

文献[5]实现了 GPU 的调度, 但是其主要针对 MPI 程序, 对于 CPU 任务和 GPU 任务混合的场景, 无法实现 GPU 资源的公平分配, 亦不能满足任务执行时的数据本地化要求.

文献[12]在 YARN 资源管理的基础上扩展了 GPU 资源的调度, 通过限制各个 GPU 任务显存的大小, 实现在单个 GPU 上的共享. 本文解决了 GPU 显存对于资源共享的限制, 但是由于采用单一约束来实现资源的公平分配, 所以对于任务执行时数据的本地化就不能很好地满足.

文献[19,20]实现了在机器学习领域的 GPU 计算资源共享, 其针对机器学习中的训练过程, 主要是针对迭代问题而提出 GPU 共享. 迭代过程需要长时间使用 GPU, 而现有的 GPU 资源调度通过任务完成释放资源, 再分配资源的方式来实现资源公平分配, 不能高效满足机器学习中的迭代需求, 因此提出依照单个作业完成时间为依据来实现公平资源分配. 它对于既存在长时间使用 GPU 的任务, 也存在短时间使用 GPU 的任务场景不太适用.

© 中国科学院软件研究所 http://www.jos.org.cn

文献[28]主要研究了MapReduce编程模型在GPU上的实现, 没有涉及GPU集群的资源管理. 文献[29]研究的是GPU任务和CPU任务如何混合调度问题, 对于GPU计算资源的公平分配策略没有太多的研究.

虽然Mesos, Yarn, Kubernetes支持GPU的资源管理, 但是其调度方式是按照每个用户的资源使用量决定分配多少GPU设备. 用户的作业根据已分配的GPU设置, 采用数据本地化优先、机架内节点优先以及任意方式. 它们按照串行方式为用户的作业分配资源, 不能从全局角度考虑所有任务的数据的位置, 所以其效率往往不是很高. 本文中避免了类似问题, 达到一体化资源分配(详细叙述见第2.1节).

本文通过最小代价最大任务数调度算法实现多个作业对GPU集群资源的公平共享, 同时尽量降低数据在网络之间的传输, 解决了资源分配的公平性与数据本地化读取的矛盾.

2 计算模型

如果计算任务所需要的数据与计算任务不在同一个计算节点上, 那么就需要通过网络读取数据, 网络带宽等因素会导致数据读取的延迟增大, 甚至会出现错误. 如果任务需求的数据与任务执行节点是同一台机器, 将带来任务的快速执行, 从而得到更好的性能. 任务需要的数据也可能存在于多个不同的计算节点上, 以一种更小的数据传输代价来实现数据读取, 是提高系统整体运行性能以及任务执行效率的一个手段.

2.1 基于GPU位置和数据位置的任务一体化调度的动机

现有的集群资源调度是按照资源释放、再分配的机制进行的, 一旦任务结束, 资源获得释放, 调度器依据调度目标将资源分配给某个作业, 作业再通过数据本地化策略提交任务到计算资源. 这种分配机制会导致次优的调度结果.

假如存在两个作业, 作业1和作业2. 作业1包含两个任务task11和task12, 作业2包含一个任务task21. 如图1所示: task11的数据在GPU1所在的计算节点上, 且在GPU2所在节点上有数据副本; task12的数据存储在GPU2所在的节点上; task21的数据存储在GPU2所在的节点上. 在资源公平共享的条件下, 每个作业获得一个GPU设备. 如果作业1获得GPU2资源, 作业2分配到GPU1资源, 此时虽然作业1的任务可以满足任务的本地化读取, 但是作业2的task21就不满足数据本地化读取, 数据的传输代价增大. 如果可以将task11分配到GPU1上, 将task21分配到GPU2上, 那么数据的网络传输就消失了. 如果只考虑数据本地化, 那么可以将task11分配到GPU1, task12分配到GPU2上, 本地化得到满足, 但是资源公平分配则欠缺. 因此, 在考虑数据本地化和资源公平分配的约束下, task11分配到GPU1上, task21分配到GPU2上, 这种方案是一个最优的选择. 本文从这一点出发, 来实现集群系统的共享调度.

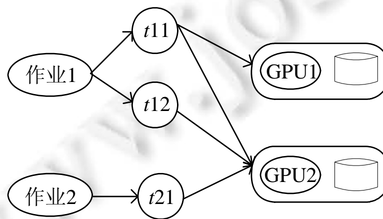

图1 任务一体化调度的例子

2.2 计算作业的抽象

在GPU集群上, 通常有多个数据密集型计算作业同时运行, 而每一个作业中亦包含多个任务同时运行, 每个任务处理一部分数据, 这些数据可能来自同一个文件, 也可能来自多个不同的数据文件. 任务在运行过程中, 相互之间不进行任何通信. 当两个任务之间存在数据依赖时, 前驱任务完成后, 后继的任务才能开始, 即作业中的任务之间通过DAG组织起来.

作业的描述可以抽象为以下3种典型形式.

(1) $y=F(x\$)$. 函数$F$对数据$x\$$进行计算, 得到新的数据y. 任务生成过程为:

```
for each (i: x$)
  task[i]
```

(2) $z=F(x\$,y\$)$. 函数 F 对数据 $x\$$和 $y\$$进行计算, 得到新的数据 z. 任务生成过程为:

```
for each (i: x$)
  for each (j: y$)
    task[i,j]
```

(3) $z=G(F((x-1)\$,y\$)\$,F(x\$,(y-1)\$)\$)$. 函数 F 分别对数据$(x-1)\$$, $y\$$以及 $x\$$, $(y-1)\$$进行计算, 各自的计算结果再进行函数 G 的计算. 任务生成过程为:

```
for each (i: x$)
  for each (j: y$)
    task[i,j]=task[i−1,j]+task[i,j−1]
```

$x\$$, $y\$$等表示数据分片的集合, F, G 等表示计算函数, 有些计算函数需要一个数据分片, 有些需要多个数据分片.

例如矩阵加法运算, 函数 F 代表矩阵加法, F 与数据集合 $x\$$, $y\$$进行结合后, 就生成一个 task, 一个大数据被分割成多个数据分片后, 就形成大量的数据集合, 不同数据集合与算子的结合, 就产生出作业中包含的任务集合. 这些任务是并行运行的, 而且运行期间任务相互独立, 即任务之间不需要任何通信和交换数据, 停止一个任务的执行不影响其他任务的执行.

2.3 GPU集群的二阶段调度框架

图 2 给出了 GPU 集群资源调度框架, 其主要负载是 GPU 计算任务, 调度器为每个 GPU 任务分配可用的计算资源, 并提交任务到 GPU 设备. 系统包括两个重要框架: 一个是用户的作业框架, 另外一个是 GPU 资源调度框架. 对于用户来说, 无论要求如何, 应尽可能快速完成用户的任务; 对于集群资源来说, 应尽可能地提高 GPU 的利用率; 对于不同的用户作业, 每个作业都能尽可能地共享相等的 GPU 资源.

作业中包含的任务由AppMaster来进行调度, 它是一个进程, 运行在某个计算节点上, 拥有自己的状态并管理作业中的任务. 作业中的每个任务被分配一定的资源并独立运行. 由于任务执行中存在这样或者那样的错误, 一个任务可能被运行多次. 为了提高处理效率, 任务通常需要加载到 GPU 设备上进行运算. AppMaster 掌握作业所需数据的存储信息、分片信息以及作业执行过程中临时数据的存储信息. 每个作业中包含的计算函数存储在动态库中, 动态库在作业提交前部署到各个集群计算节点上. 一旦用户提交作业, 资源管理器为 AppMaster 进程分配资源并运行. AppMaster 管理作业中的任务, 监控任务的状态并控制任务的运行.

当 AppMaster 进程启动后, 首先对数据文件进行分片, 分片信息被存储在 AppMaster 的数据管理模块中. 然后它根据数据分片形成任务的集合, 任务之间通过数据依赖关系形成 DAG.

整个资源调度管理采用类似 Mesos 的调度框架, 作业注册后, 资源管理器向其提供 GPU 资源. 每个计算节点定期向资源管理器汇报资源的状态. 资源调度框架的运行过程按照以下步骤完成(以下的步骤序号与图 2 中序号一一对应).

第 1 阶段为各个作业并行运行, 产生局部调度结果, 即步骤①–步骤③; 第 2 阶段为全局最优结果的生成, 即步骤④–步骤⑦:

① 各个集群中的计算节点(node)定时向资源分配器汇报 GPU 的状态信息;

② 资源分配器接收到计算节点的状态信息后, 向所有注册的作业提供全部的可用 GPU 设备资源;

③ 各个作业(job)接收到集群的全部资源状态信息后, 按照数据分片特性以及位置为任务分配 GPU 设备, 给出一个初始资源分配方案;

④ 各个作业向资源分配器汇报初始资源分配方案;

⑤ 资源分配器接收到所有作业的初始资源分配方案后, 按照公平原则进行资源分配的裁决, 得到一个全局的资源分配方案;

⑥ 资源分配器向各个作业返回全局的资源分配方案, 以及哪些任务无法分配到资源的消息;

⑦ 各个作业根据全局资源分配方案, 向自己的执行器(executor)发送提交任务的请求. 如果一个计算节点是第1次运行某个作业的任务, 资源分配器负责启动该作业的执行器.

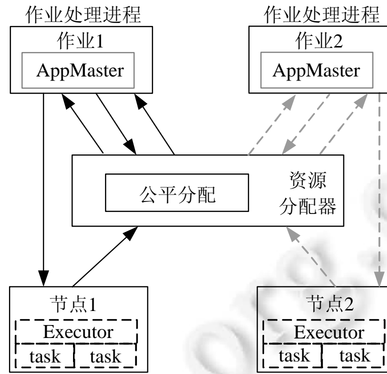

图2 GPU集群资源调度框架

2.4 数据传输代价的计算模型

任务的数据存储在集群各个计算节点上, 为保证高可靠性,数据文件有多个副本, 存储在不同的计算节点上. 对于每个任务, 可能只需要一个数据源, 也可能需要多个数据源, 这取决于任务的计算函数. 任务计算过程中, 使用来自于任何一个集群节点上的数据副本. 一个任务的数据也可能来自于当前任务的前一个任务产生的中间结果, 这就要求前驱任务结束后, 后继任务才可以被调度. 一个任务只有在全部数据都准备好之后, 才可以进行任务的执行. 任务执行时所需的输入数据的存储信息、数据大小以及副本位置都可以通过系统提供的 API 获得. 按照这种方式, 作业获得计算资源后, 就可以估算任务在该计算资源上执行时需要的数据传输代价, 通常依据网络带宽来评估传输代价的大小. 例如: 当作业 j 的第 k 个任务 t_j^k 获得了一个GPU设备时, AppMaster根据GPU所在的计算节点位置以及任务所需要的数据的存储位置, 计算出任务使用GPU设备时需要的数据传输代价和数据位置. 当任务需要跨越不同计算节点获取数据时, 还需要考虑网络带宽.

我们使用一个例子来说明考虑网络带宽条件时, 如何根据数据传输代价选择任务的计算节点. 如图 3 所示: 集群系统包括两个机架, 机架1和机架2, 两个机架之间通过一个交换机连接. 计算节点 A 和 B 在机架1上, 计算节点 C 和 D 在机架2上, 它们具有相同的带宽 B_r. 机架1和机架2之间具有带宽 B_s. 假设存在两个计算节点 A 和计算节点 B, 那么它们之间的带宽可以使用以下方式得到.

- 如果 $A \in rack1$, $B \in rack1$, 或者 $A \in rack2$, $B \in rack2$, 那么它们的带宽 $B=B_r$;
- 如果 $A \in rack1$, $B \in rack2$, 或者 $A \in rack2$, $B \in rack1$, 那么它们的带宽 $B=B_s$.

在数据中心的机架内以及机架之间都设置自己的带宽管理器, 那么同一机架内监视的带宽数据为 B_r, 不同机架之间监视的带宽数据为 B_s. 通过任意两个节点之间的测量数据的交换延时, 可以估算得到它们的带宽.

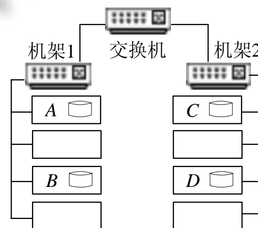

图3 GPU集群网络拓扑结构

假如一个数据密集型计算任务, 其需要两个数据分片: 一个数据分片的大小为 S_a, 存储在计算节点 A 上;

另外一个数据分片的大小为 S_c, 存储在计算节点 C 上. 如果任务获得的可用计算资源分别为 A 和 D, 那么调度器选择时, 就需要考虑各自的数据传输代价. 选择计算节点 A 时, 数据 S_a 在本机, 其数据传输代价可以看作0, 只包含数据 S_c 在机架之间的传输代价, 任务的数据传输代价为 $W_s=S_c/B_s$; 选择计算节点 D 时, 其包含 S_c 在同一机架内的数据传输代价和 S_a 在机架间的数据传输代价, 任务的数据传输代价为 $W_s=S_c/B_r+S_a/B_s$. 作业的调度器比较这个两个数据传输代价的大小, 选择一个最小代价的计算节点作为任务的执行节点.

2.5 公平的资源分配

假设每个作业在独占整个集群资源的情况下运行时间是 t 秒, 那么公平资源分配的目标就要求: 如果有 $|J|$ 个作业并行运行, 每个作业的运行时间不要超过 $|J|\times t$ 秒. 在实际中, 如果没有大量作业竞争资源, 可以保证公平性. 通过控制任务的数量, 保证同一个时间只有 K 个任务同时运行, 就可以满足公平性要求. 当达到最大限制数量 K 后, 后续的任务就进入排队状态. 一旦有任务完成, 队头任务优先获得计算资源. 如果存在资源竞争, 就可能存在作业的延迟. 这就是说, 如果设置较大的 K 值, 就可能出现多个任务的数据都存放在同一个节点导致资源竞争加剧的情况. 这在保证数据本地化读取时很难保证公平性. 但是, 选择太小的 K 值, 由于任务的不足, 可能导致某些集群节点资源处于闲置状态.

当一个作业提交时, 实际上只能允许其部分任务执行. 假如给定一个比例系数 α, 集群中 GPU 的数量为 M, 该作业实际运行的任务数就可以定义为 αM. 资源分配比例系数可以根据作业的提交以及作业的完成进度动态改变. 因每个作业包含的任务资源需求预测十分困难, 我们目前限制每个GPU上同时只能运行一个任务.

3 基于 GPU 数量的共享调度策略

本节中, 我们给出了一种基于 GPU 设备数量的作业调度策略, 这些策略作为基本共享调度算法, 用于和我们的最小代价最大任务数调度算法进行比较, 基本调度算法系统也使用了一些调度策略来支持数据本地化读取.

基于GPU数量的调度方式是一种基本的资源公平分配方式, 实现简单, 所以本文对于基于GPU数量的调度方面只进行一些基本的介绍.

3.1 基于GPU数量的共享调度算法

基于 GPU 数量的共享调度(简称 GS)中, 最简单的公平原则就是防止一个作业占用大量的 GPU 设备来执行大量的任务, 从而导致其他作业缺乏 GPU 设备而长时间的等待. 本文在 GPU 计算资源上, 采用均匀分配GPU 设备的公平调度方法.

基于 GPU 数量共享调度算法中, 每个作业包含一个重要参数, 称为基本资源份额. 它是每个作业应该分配到的资源数量, 通过总体的资源数量以及运行中的任务数量来计算. 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, Q 代表集群节点中的 GPU 设备数量, K 代表集群中正在运行的作业数量, N_j 代表作业 j 中包含的未执行的任务数量. 若 K 个作业的基本资源份额之和小于 GPU 设备数量, 即 $\sum_j A_j^*<Q$, 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 $\sum_j A_j=\min(Q,\sum_j N_j)$ 时, 分配过程就停止. 在作业并行运行的过程中, 存在任务的开始与结束, 意味着 GPU 设备不断地被分配和释放, 那么分配给某个作业的 GPU 设备数量就会产生波动, 这会导致资源的不公平使用.

基于 GPU 设备数量的公平资源分配算法是, 计算出每个作业正在运行的任务数量以及基本资源份额, 如果正在运行的任务数量低于基本资源份额, 则可以向该任务分配 GPU 资源; 否则不予分配. 这种方式可以确保作业中正在运行的任务数没有低于基本资源份额之前, 不可能再给当前作业分配 GPU 设备, 从而保证其他作业获得它自己的基本资源份额. 当所有任务运行一段时间后到达稳定状态, 每个作业都可以得到其基本的资源份额. 当一个作业拥有的任务数量小于其基本份额, 即 $N_j<\lfloor Q/K\rfloor$, 将会出现一段时间的不平衡, 一些达到

© 中国科学院软件研究所 http://www.jos.org.cn

基本资源份额的作业可以再提交一些任务. 当一个新的作业开始, 也将出现 GPU 设备分配的不平衡, 此时基本资源份额需要重新计算. 这个不平衡时间的长短取决于任务执行的时间. 如果任务执行的时间较长, 那么只有作业的一些任务结束后才能重新平衡. 算法 1 给出基于 GPU 数量的资源分配的实现.

算法 1. 基于 GPU 数量的资源分配.

输入: 作业 J, GPU 集合 G;

输出: 任务与 GPU 的映射.

(1) 资源管理器向作业提供 *ResourceOffer*:

S1 $G=\{g_1,g_2,\ldots,g_{|G|}\}$ /*可用资源*/

S2 $J=\{j_1,j_2,\ldots,j_n\}$ /*n 个作业*/

S3 **for each** (j_i)\{

S4 A_i /*计算每个作业使用的 GPU 数量*/

S5 \}

S6 $j_l=\min\{A_1,A_2,\ldots,A_n\}$ /*选择使用资源最小的作业*/

S7 send *ResourceOffer*(G) to j_l

(2) 作业调度任务 *SchedulerTask*:

S8 receive *ResourceOffer*(G)

S9 **for each** $(g_i\in G)$

S10 **for each** $(t\in j_l)$

S11 **if** $(\exists(t_j^l.r < g_i.r))$

S12 *compute*(*t.cost*) /*计算代价*/

S13 **return** min(*t.cost*)

算法 1 中, 当存在空闲的 GPU 设备时, 资源调度器选择使用 GPU 数量最小的作业, 将空闲 GPU 分配给该作业. 作业收到可用资源后, 选择一个数据传输代价最小的任务来使用该资源. 这种方法通过阻止占有 GPU 较多的作业获得资源来实现 GPU 设备在作业之间的公平分配. 算法的 S3–S5 计算每个作业使用的 GPU 设备数量, S6 选择使用 GPU 数量最小的作业. S7 向使用资源最小的作业提供可用 GPU 资源, 即 *ResourceOffer*. S8 是作业接收到资源管理器提供的 *ResourceOffer*. S9–S13 为作业分配 GPU 资源给任务的过程: S11 给出了选择条件, 即任务所需的 GPU 显存要小于 GPU 提供的显存大小; S12 计算任务执行的数据传输代价; S13 为选择一个最小数据传输代价的任务, 并分配对应的 GPU 资源. 算法的复杂度为 O(mn), m 为处于等待状态的任务数量, n 代表可用资源数量.

3.2 调度策略

(1) 非抢夺策略(GS)

非抢夺策略下无法一直维持 GPU 数量在各个作业之间的公平分配. 假如某个作业的任务数量小于基本资源份额, 此时一些作业就超额占有资源. 当作业结束, 新的作业到来, 可能会造成该作业无法得到其拥有的 GPU 资源份额. 另外一种情况是新的作业到达, 需要重新计算基本资源份额, 但是所有资源正在被任务使用, 新作业无法获得自己的资源份额. 当调度算法不允许强制终止占用资源较多的作业的任务时, 就会出现一段时间内作业之间使用资源的不平衡的情况.

(2) 抢夺式调度策略(GSP)

上面的情形(1)中, 如果任务一直处于运行状态, 资源被长时间超额占据, 新来的作业没有资源可以分配或者无法得到应有的基本资源份额, 资源使用就不平衡了. 这个时间也许很长, 所以可以使用一种抢夺式策略: 如果作业 j 的正在运行的任务数大于其基本资源数量 A_j^*, 那么资源管理器就强制停止其部分任务. 当然, 强制停止的任务需要选择运行时间最小的任务, 这样可以避免浪费更多的计算力.

对于基本的公平分配, 只要作业分配的资源数没有降低到 0, 即使使用资源抢夺策略, 作业就会一直运行下去. 因为其运行时间最长的任务在没有结束前不可能会被强制停止, 所以作业至少能拥有自己的部分资源.

(3) 延迟调度策略(GSD)

数据密集型计算任务中, 数据规模比较大, 而网络带宽是一种稀缺资源, 数据在网络节点之间的传输会导致计算的代价增大. 所有的任务都试图以数据传输代价最小的方式获得资源. 在一个 GPU 集群上, 如果一个任务结束, 其资源被释放, 那么排队中的任务就可以申请这个资源. 调度器从 $\min(A_j)$的作业中选择一个任务来使用这个资源, 并计算该任务需要的数据到 GPU 设备位置之间的数据传输代价. 一种可能是该数据传输代价不是该任务的最小代价, 针对这种情况, 一种处理方式是立刻将任务的数据传输代价从最小代价降低为次最小代价, 一直到最大传输代价为止, 然后分配该资源给任务; 另外一种处理策略是, 作业 A_j 放弃本次分配, 等待下一次分配机会. 通过延迟等待, 能够戏剧性地增加任务选择数据传输最小代价资源的机会, 从而降低任务计算过程中的数据传输代价. 这种调度称之为延迟调度, 在 Spark[30]的任务调度中已经得到验证.

假设作业 j 中正在运行的任务数为 N_j, 集群系统中基本任务数量为 A_j^*, 当作业 j 的 $\delta=A_j^*-N_j$ 且$\delta>0$ 时, 作业 j 的一个任务 t 会被分配到一个 GPU 设备. 如果 GPU 资源不满足任务的最小数据传输代价, 那么任务 t 放弃本次资源分配, 并增加分配次数, 即设置$\delta=\delta+1$. 当分配次数δ超过一个阈值时, 任务 t 即使不满足最小数据传输代价, 如果满足次最小数据传输代价, 任务 t 被分配到该资源. 如果不满足次最小数据传输代价, 再设置$\delta=\delta+1$, 当δ超过另外一个阈值时, 任务 t 必须被分配, 不再考虑数据传输代价.

当使用延迟调度时, 作业的调度器不会简单地使用它收到的第 1 个调度机会, 而是等待设定的最大分配次数到来前的调度机会. 最后, 作业才降低数据本地化读取限制并接收下一个调度机会.

当然, 延迟调度在一定程度上会造成不公平的现象发生, 但是它可以带来作业执行效率的提高.

4 基于最小代价最大任务数的调度

基于GPU设备数量的共享调度算法扩展了抢夺以及延迟策略, 问题是哪一个作业的哪个任务能够被调度到基本资源配额 A_j^*中, 这个问题对于调度的公平性以及本地化数据读取非常重要. 在前面的基本算法中, 本文采用了一些启发式策略, 当新的任务到达或者离开系统, 基本上采用贪心、被动方式去解决.

在本节中, 本文介绍一种新的调度方式来处理同时请求资源的作业. 这种方式中最主要的数据结构是一个流网络图, 数据结构中包含资源信息、等待中的任务和它们的数据传输代价信息等描述. 图中的每一条边都包含有特有的权重值以及容量值. 为了达到了预期的调度目标, 本文还使用了一种标准的策略将预期的调度目标转换为调度实例, 以期满足设计的要求.

流网络图数据结构中的一个核心是边的代价估算, 代价的估算依据是数据通过网络在计算节点之间的传输开销. 如果能够大致估算出一个开销, 那么就可以计算出最小代价的调度结果.

实现最小代价最大任务数的另外一个需要处理的问题是: 如何构建一个图的拓扑结构, 并利用一定的算法来解决资源的公平分配问题. 在本文中, 使用了一种特殊的拓扑结构图, 通过设置额外的顶点(后面章节中的不分配顶点)、顶点对应边的代价、边的容量等, 解决了 GPU 资源在作业之间的公平分配问题.

4.1 最小代价流的基本概念

本文选择使用一个标准的流网络来解决调度实例. 流网络是一个有向图, 图中的每一条边 e 有一个非负的容量值 c_e 以及一个代价值 w_e, 图中的每个顶点 v 有一个流量提供值ε_v, 存在条件 $\sum_v \varepsilon_v=0$. 一个可行的流的含义是设置一个非负的整数流 $f_e \leqslant c_e$ 给每一个边, 因此, 对于每一个顶点 v, 存在 $\varepsilon_v+\sum_{\varepsilon\in I_v} f_e=\sum_{\varepsilon\in O_v} f_e$. 其中, I_v 代表顶点 v 的输入边, O_v 代表顶点 v 的输出边. 在一个可行流中, $\varepsilon_v+\sum_{\varepsilon\in I_v} f_e$ 代表的是通过顶点 v 的流, 一个最小代价可行流可以表示为最小化 $\sum_\varepsilon f_e w_e$.

计算最小代价最大任务数的算法是: 若 f 是流量为 $v(f)$的最小代价流, p 是关于 f 的从 v_s 到 v_t 的一条最小代

价增广路径, 则 f 经过 p 调整流量 q 得到新的可行流 $f'=f+q$. 通过不断迭代, 直到无法找到这样的路径 p 为止.

最坏情况下, 对于有 V 个顶点、E 条边的流网络, 最小代价最大任务数的计算复杂度为

$$O(E\log(V)(E+V\log(V))).$$

4.2 流网络拓扑结构的建立

GPU 设备到任务的分配问题被简化为一个流网络中的最小代价最大流的计算. 一个流网络可以看作是系统进行一次资源分配时的快照, 包含一个将要执行的任务集合、可以使用的资源集合、任务运行时的数据传输代价以及每个作业可以运行的任务数量. 使用流网络的最大优点是很容易将一个包含复杂权重的任务到资源的匹配过程进行优化. 通过构建一个简单的流网络, 设置一些常量参数, 就可以既简单又清楚得到 GPU 集群资源在任务之间的共享调度效果.

整个流网络的结构可以进行物理的解释: 流网络图中的一个单位容量的流代表着一个任务到一个 GPU 资源的分配. 流网络图被分为两个部分: 图的一个部分代表需要执行的任务的集合, 另外一个部分代表可以提供 GPU 设备的集合. 两个部分之间的边代表任务到 GPU 设备的匹配, 任务之间不存在边, 而 GPU 设备之间也不存在边, 这个匹配过程似乎是一个二部图, 但是由于存在数据传输代价最小等优先匹配问题, 所以不能简单地简化为一个二部图. 在实现过程中, 为流网络增加一个源顶点 S 和汇节点 E. 汇节点接收所有流入的可行流, 即可行的任务到GPU设备的匹配结果, 可行流的路径代表着一个任务到达GPU设备的路径. 当任务需要的数据源包含多个时, 并且某些数据源可能跨越一个机架, 即任务的分配就需要考虑数据传输跨越机架的代价, 也有可能跨越一个远程交换机, 即任务的分配是任意的. 在每一个 GPU 设备上, 它的输入边的容量是单位 1, 即代表只有一个任务可以分配到该计算资源, 其他任务是不能同时分配到该计算资源上. 这就代表着: 如果等待执行的任务数量超过可用 GPU 设备数量的话, 一些任务是无法分配到计算资源, 这些任务需要等待. 通过控制 GPU 设备到汇节点之间的边的容量, 可以很好地控制任务分配的数量. 对于每个作业, 通过设置每个作业不被调度的任务的最大和最小数量, 就可以得到每个作业最小和最大的可以被分配到资源的任务数, 实现作业的基本资源份额控制.

不像基于 GPU 数量的资源分配方式, 基于流网络的调度模型可以很好地利用任务的一些属性, 如第 j 个作业的第 k 个任务、任务的数据块大小、数据块所在的计算节点位置信息. 根据这些信息, 第 m 个计算节点的第 n 个 GPU 设备在执行任务之前, 任务读取数据所需要的代价也就可以估计出来. 这些信息都将被融入到流网络的拓扑结构中, 通过数据传输代价的方式被反映出来, 即流网络中任务对应的顶点到 GPU 设备对应的顶点 g_m^n 之间的代价. 如果任务所需要的数据位置与GPU设备位于同一个计算节点, 其代价最小; 所需要的数据位置与GPU设备位于同一个机架, 其代价次之; 所需要的数据位置和GPU设备位置跨越机架, 其代价最大.

(1) 流网络中顶点的建立

对于任何一个请求执行的作业, 系统提供一个 AppMaster, 流网络 JNW 中就会形成一个顶点 j. 如果存在 k 个作业, 即存在作业的集合 $J=\{j_1,j_2,\ldots,j_k\}$, 那么在 JNW 中就会存在 k 个顶点, 分别表示为 $j_1,j_2,\ldots,j_k$.

对于第 i 个作业 j_i, 其任务数量表示为 $x=|j_i|$, 任务集合 $j_i=\{t_l^i\}$, 那么在 JNW 中就形成 x 个顶点, 分别表示为 $t_1^i,t_2^i,\ldots,t_x^i$.

假如 GPU 集群中存在 m 台计算节点, 即 $M=\{m_i\}$. 对于每台计算节点, 其包含有 n 个 GPU 设备, 我们为每个GPU设备建立一个执行队列, 即 $G=\{g_i\}$ 且 $n=|G|$, 那么在JNW中就形成 n 个顶点, 分别表示为 $g_1^i,g_2^i,\ldots,g_n^i$.

由于GPU设备数量的限制, 当执行作业中的任务数量大于可以使用的GPU数量时, 必然有一部分任务无法被分配到计算资源. 为了保证每个作业能够公平的分配到计算资源, 我们为每一个作业设置一个不调度任务顶点, 记为 u. 对于第 i 个作业, 不调度顶点就表示为 u_i.

对于一个流网络, 我们设置一个源顶点 S 和一个汇顶点 E. 顶点 S 只有流出的流量, 没有流入流量; 顶点 E 只有流入的流量, 没有流出的流量.

(2) 流网络中的边的建立

© 中国科学院软件研究所 http://www.jos.org.cn

流网络中的每一条边 e, 至少包含两个参数: 一个是容量 c, 另外一个是代价 w.

假设存在 k 个作业, 即存在作业的集合 $J=\{j_1,j_2,\dots,j_k\}$, 源顶点 S 到每一个作业顶点之间建立一条边 e, 边 e 的容量 $c=|j_i|$, 代表作业 j_i 中任务数量. 从源顶点 S 到作业顶点 j_i 的代价 w, 对于任何一个作业, 分配资源的机会都是公平的, 所以设置 $w=\varepsilon$, 即ε的取值几乎为 0, 代表源点到作业顶点之间的路径没有代价.

作业顶点 j_i 到每个任务顶点之间, 构建一条边. 由于一个任务只能属于某一个作业, 所以作业和其他作业的任务之间不能建立边. 每个作业与其任务之间的边的容量 $c=1$, $w=\varepsilon$. 它表明: 作业到每个任务之间只能分配一个计算资源,而且作业到任务之间的分配过程不产生任何代价.

对于任何一个任务, 执行过程中所需要的数据是一个非常重要的参数, 假设任务需要 z 个数据, 每个数据的大小分别为$\{d_1.size,d_2.size,\dots,d_z.size\}$, 每个数据所在的计算节点的位置分别为$\{d_1.loc,d_2.loc,\dots,d_z.loc\}$.

第 1 种情况是: 如果作业 j_i 的任务顶点 t_x^j 的数据全部在计算节点 m 上, 并且计算节点 m 上存在一个可用资源, 即存在顶点 g_j^m; 同时, 任务顶点 t_x^j 所需要的 GPU 显存不超过 g_j^m 所对应的 GPU 的可用显存, 那么顶点 t_x^j 与 g_j^m 就建立一条边, 边的容量 $c=1$, 边的代价 $w=(d_1.size+d_2.size+\dots+d_z.size)/D$, 其中, D 代表磁盘的 IO 过程的吞吐量. 同样, 对于当前计算节点 m 上的其他 GPU 显卡, 如果其显存要求满足条件, 那么亦建立任务顶点 t_x^j 到这些顶点 g_j^m 之间的边, 其边的容量 $c=1$, 建立该边所对应的代价为 $w=(d_1.size+d_2.size+\dots+d_z.size)/D$.

作业 j_i 的任务顶点 t_x^j 到同一机架内的其他计算节点 p, 如果任务顶点 t_x^i 所需要的 GPU 显存大小不超过 g_j^p 所对应的 GPU 的可用显存, 那么顶点 t_x^i 与 g_j^p 就建立一条边, 边的容量 $c=1$, 边的代价 $w=(d_1.size+d_2.size+\dots+d_z.size)/B_r$, 其中, B_r 代表同一个机架内的网络带宽.

作业 j_i 的任务顶点 t_x^i 到跨越机架内的其他计算节点 q, 如果任务顶点 t_x^i 所需要的 GPU 显存大小不超过 g_j^q 所对应的 GPU 的可用显存, 那么顶点 t_x^i 与 g_j^q 就建立一条边, 边的容量 $c=1$, 边的代价 $w=(d_1.size+d_2.size+\dots+d_z.size)/B_s$, 其中, B_s 代表机架之间的网络带宽.

第 2 种情况是: 如果作业 j_i 的任务 t_x^j 所需的数据源跨越多个机架, 资源分配器分配一个 GPU 设备后, 就需要分别计算数据的 I/O 代价、机架内数据传输代价以及机架之间的数据传输代价. 例如: 假设 z 个数据的大小分别为$\{d_1.size,d_2.size,\dots,d_z.size\}$, 这些数据分别存储在多个计算节点$\{m_1,m_2,\dots,m_z\}$上, 即 $d_1.loc\in m_1$, $d_2.loc\in m_2,\dots, d_z.loc\in m_z$. 如果 t 个计算节点在机架 $R1$ 上, 即$\{m_1,m_2,\dots,m_t\}\subseteq R1$; 如果 $z-t$ 个计算节点在机架 $R2$ 上, 即$\{m_{t+1},m_{t+2},\dots,m_z\}\subseteq R2$ 上. 对于计算计算节点 m 上存在一个顶点 g_j^m, 如果任务顶点 t_x^i 所需要的 GPU 显存大小不超过 g_j^m 所对应的GPU的可用显存, 那么顶点 t_x^i 与 g_j^m 就建立一条边, 边的容量 $c=1$, 边的代价为 $w=w1+w2+w3$, $w1$, $w2$ 及 $w3$ 分别为数据 I/O 带宽代价、数据机架内带宽代价以及数据机架之间的带宽代价. 如果 $m\in R1$, 并且$\{d_1.loc+d_2.loc+\dots+d_c.loc\}\subseteq m$, 此时 I/O 代价为 $w1=(d_1.size+d_2.size+\dots+d_c.size)/D$, 其中, D 代表磁盘的 IO 过程的吞吐量. 如果 $m\in R1$, 而且$\{d_{c+1}.loc+d_{c+2}.loc+\dots+d_t.loc\}\subseteq R1$, 则机架内带宽代价为 $w2=(d_{c+1}.size+d_{c+1}.size+\dots+d_t.size)/B_r$, 其中, B_r 代表机架内带宽. 剩余的数据的位置与 m 不在同一个机架内, 此时, 跨越机架的带宽代价为 $w3=(d_{t+1}.size+d_{t+2}.size+\dots+d_z.size)/B_s$, 其中, B_s 代表机架之间的带宽. 如果 g_j^m 上不存在任务 t_x^i 所需要的任何数据, 并且$\{m_1,m_2,\dots,m_t\}\subseteq R1$, 那么顶点 t_x^i 与 g_j^m 边的代价就只包含机架内数据传输代价和机架间的数据传输代价, 即为 $w=w1+w2$, $w1=(d_1.size+d_2.size+\dots+d_t.size)/B_r$, $w2=(d_{t+1}.size+d_{t+2}.size+\dots+d_z.size)/B_s$. 如果任务 t_x^i 所需要的数据源与 g_j^m 的位置全部为不同机架之间, 那么其数据传输代价就表示为

$$w=(d_1.size+d_2.size+\dots+d_z.size)/B_s.$$

每一个计算节点的 GPU 顶点 g_j^m 到汇顶点之间建立一条边, 容量 $c=1$, $w=\varepsilon$.

对于每一个作业 j_i, 需要设置一个不调度顶点 u_i, 作业顶点 j_i 到不调度顶点 u_i 之间存在一条边, 边的容量 $c=N_j-A^*$, 其物理含义是作业的任务数量与基本资源份额的差值. 边的代价表示为 $w=\alpha$, α代表一个惩罚代价. 不调度顶点 u_i 到汇顶点 E 之间同样构建一条边, 边的容量 $c=N_j-A^*$, 边的代价 $w=\alpha$. 设置不调度顶点, 并且边

© 中国科学院软件研究所 http://www.jos.org.cn

的容量和代价可以调节. 对于边的容量的设置, 保证作业 j_i 中包含全部的任务都有机会参与到资源分配的过程中, 但是哪些任务能够分配到计算资源, 需要按照代价最小的原则来选择. 对于边的代价的设置, 目的是达到资源在各个作业之间公平分配, 通过调整α代价, 使得作业 j_i 的一部分任务从不调度顶点 u_j 上建立可行流, 避免其任务过多地占用可用的GPU设备资源, 导致其他作业无法获得足够的GPU设备资源, 所以其边的容量设置以及代价的设置非常重要.

流网络的拓扑结构如图 4 所示. 图 4 中包含 4 台计算节点, m_1, m_2, m_3 以及 m_4, 其中, m_1 和 m_2 位于同一个机架, m_3 和 m_4 位于另外一个机架, 计算节点 m_1 包含 2 块 GPU 卡, m_2 包含 1 块 GPU 卡, 计算节点 m_3 包含 1 块 GPU 卡, m_4 包含 2 块 GPU 卡. 包含 3 个作业 j_1, j_2 以及 j_3. 作业 j_1 中包含 4 个任务, 作业 j_2 中包含 3 个任务, 作业 j_3 中包含 2 个任务. 流网络中的边的代价以及容量省略.

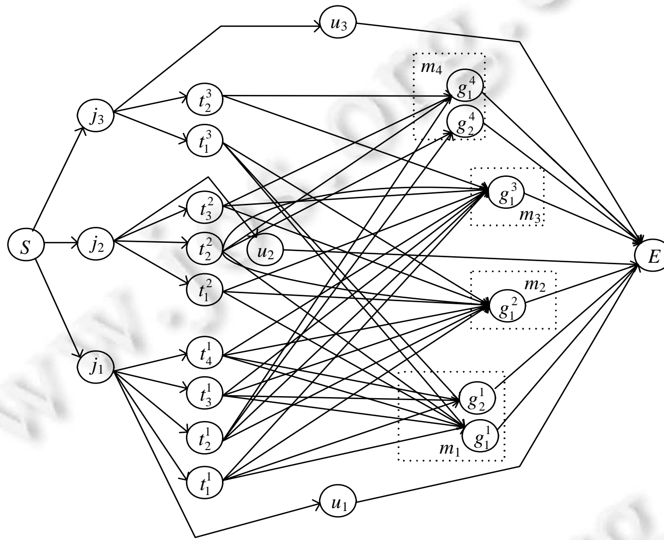

图 4 流网络拓扑结构图

针对代价, 我们设置了 3 个代价惩罚系数: α_1, α_2 以及α_3. 3 个代价惩罚系数的值均大于 1. α_1 用于计算机架内数据传输代价时的惩罚系数, α_2 用于计算机架之间数据传输代价时的代价惩罚系数, α_3 代表作业的任务到不调度顶点之间边的代价惩罚系统. 通过对这 3 个代价惩罚系数的调节, 可以很好地实现数据传输代价优先以及资源调度的公平性. 当α_1 和α_2 非常大时, 数据传输代价最小策略就能很好地体现; 当α_3 非常大时, 各个作业分配到的 GPU 资源的公平性就会受到影响.

基于流网络的最小代价最大任务数调度算法可以在全局范围内得到一个最优资源分配解. 每当出现资源扩大的情况, 例如集群中添加了新的计算节点, 或者有任务执行结束, 流网络的结构就会发生变化. 因此, 在调度时间到达时, 就需要重新计算出最小代价下的最大任务数量, 即对新出现的资源进行分配, 即任务到资源的映射. 每当出现资源减少的情况, 例如某个集群中计算节点失效, 或者有新的作业提交到集群系统中, 此时流网络的结构亦会发生变化, 需要计算变化后流网络的可行流, 即任务到资源的重新分配. 另外, 一些作业中的任务可能运行在失效的计算节点上, 就造成了新的资源分配的不公平. 或者新的作业的提交后, 其资源被正在运行中的任务全部占用, 新作业无法得到计算资源, 从而导致资源分配的不公平.

对于资源扩大的情况, 只需要对每个作业已经使用的计算资源进行计算, 将空闲的资源平均分配给各个作业, 即修改任务顶点到不调度顶点资源的容量, 按照图 4 的流网络, 可以得到相对公平的分配方案. 对于资源减少的情况, 此时需要考虑公平性策略来修改流网络中一些边的代价值.

© 中国科学院软件研究所 http://www.jos.org.cn

4.3 资源调度的公平策略

在上一节中, 对于不调度顶点, 通过设置不调度顶点到汇顶点之间的容量值, 就可以控制作业中可以运行的任务数量, 即控制可分配到资源的任务数量. 这是控制资源分配公平的一个关键点. $A_{\min}^{j}$代表作业j可以运行的最小任务数量, $A_{\max}^{j}$代表作业j可以运行的最大任务数量, 每个作业提交N_j个任务, 那么$\sum_j N_j = N$.

(1) 可以抢夺的资源公平分配(FSP)

在基于队列的资源分配中, 我们计算每个作业可以分配到的资源份额A_j. 如果$A_{\min}^{j} = A_{\max}^{j} = A_j$, 那么作业$j$只能使用集群资源的基本份额.

如果某个作业中包含的任务数量小于其基本资源份额, 即$A^*<N_j$, 那么其他作业的任务就可能超过其基本资源份额, 当该作业有新的任务提交时, 就需要收回自己的资源, 即强制停止超过基本资源份额的作业中的任务, 一般选择执行时间最小的任务来进行强制停止.

(2) 不抢夺的资源公平分配(FS)

如果一个作业已经分配的计算资源份额超过了其基本资源份额, 那么必然导致其他作业分配到的资源份额低于其基本资源份额. 为了阻止该作业继续获得资源, 对该作业中未分配资源的任务, 调整任务顶点到不分配顶点之间边的容量以及边的代价, 降低其本次调度中获得 GPU 设备的机会, 最终实现资源的公平分配. 这种情况下, 正在运行中的任务不会被强制停止.

(3) 不考虑公平性的最大任务数分配(FSU)

对于不考虑公平性的资源分配算法, 其含义就是保证最小代价下的最大任务数. 这种情况下, 我们只需要将流网络中的每个作业的不分配顶点删除. 通过计算这种流网络的最小代价最大流, 就可以得到只考虑最小代价时的最大任务数分配方法.

对于不同的策略, 仅仅通过重新构筑流网络的顶点、边的容量以及边的代价, 就可以实现不同调度策略的要求. 基于 GPU 数量的共享调度在概念上非常简单, 但是比起简单修改流网络拓扑结构而达到效果, 基于 GPU 数量的共享调度机制的实现是复杂的.

4.4 算法实现

(1) 局部流网络拓扑结构建立算法

算法 2 中, 主要过程是创建流网络的顶点和边, 设置边的容量和计算数据传输代价. S3–S5 创建作业顶点和不提交任务顶点, 同时对每个作业建立任务顶点. S7, S8 对于每个可用计算资源(GPU 设备), 建立 GPU 资源顶点. S9–S18 用于建立作业的任务到 GPU 资源顶点之间的边, 边的容量为 1, 边的代价根据任务的数据大小和位置来计算. S19~S22 建立作业到不分配顶点之间的边, 这里的代价和容量决定了资源的公平性, 需要计算每个作业不被调度的任务数量. S23–S26 分别建立作业顶点到任务顶点之间的边. S27–S30 建立不分配顶点到汇点之间的边. 这里的边的代价使用α, 通过α值的调节, 影响资源的分配.

算法 2. 局部流网络创建.

输入: 作业集合 J, 计算节点 M 的 GPU 队列集合 G;

输出: 流网络.

S1 new vertex S, E

S2 **if** ($j_i \in J$){ /*创建作业和作业顶点*/

S3 new vertex u_i; insert (u_i,V)

S4 new vertex j_i; insert (j_i,V)

S5 **for** ($t_k^i \in J_i$) new vertex t_k^i; insert (t_k^i,V)

S6 }

S7 **for** ($m_i \in M$) /*创建 GPU 顶点*/

```
S8      for ( g_k^i ∈ m_i )   new vertex  g_k^i ; insert ( g_k^i ,V )
S9    for (v_i∈V) {   /*创建任务到GPU的边*/
S10     for (v_j∈V) {
S11       if (v_i=v_j) continue;
S12       if (v_i∈J∧v_j∈M) {   /*任务需求的显存不超过GPU可用显存*/
S13         if (v_i.gmem≤v_j.gmem) {
S14           new edge  e_i^j
S15           e_i^j.c = 1
S16           e_i^j.w = ε   /*按照第4.2节中的情形(2)计算代数据传输价*/
S17         }
S18       }
          /*建立作业到不分配顶点之间的边*/
S19       if (v_i.type=job∧v_j.type=unscheduler) {
S20         new edge  e_i^j
S21         e_i^j.c = N_j − A^j; e_i^j.w = α
S22       }
          /*建立作业顶点到与作业中任务顶点之间的边*/
S23       if (v_i.type=job∧v_j.type=task∧v_j.id=v_i.id) {
S24         new edge  e_i^j
S25         e_i^j.c = 1; e_i^j.w = ε
S26       }
          /*建立不分配顶点到顶点E之间的边*/
S27       if (v_i.type=unscheduler) {
S28         new edge  e_ε^i
S29         e_ε^i.c = N_i − A^i; e_ε^i.w = α
S30       }
S31     }
S32 }
```

(2) 最小代价最大任务分配算法

各个作业按照算法2建立局部的流网络后，将信息发送给资源分配器．接下来，资源分配器根据各个局部流网络合并成全局的流网络，然后求出最小代价下的最大任务数．合并全局的流网络相对简单，在此不再叙述．求解全局流网络的过程，本文使用对偶法解最小代价最大流问题．

基本思路是如下．

① 初始化可行流 $f=\{0\}$;

② 寻找从源顶点 S 到汇顶点 E 的一条最小费用可增广路径 p: 若不存在 p，则为 N 中的最小费用最大流，算法结束；若存在 p，则用求最大流的方法将 f 调整成 f'，使 $v(f')=v(f)+Q$,并将 f'赋值给 f，转步骤②．

具体的详细算法这里不再叙述．

(3) 算法复杂度和可扩展性分析

每一次资源调度过程中包括两个步骤：第1步是建立包含代价的流网络，第2步针对流网络计算最小代价最大流．假如当前需要调度的作业数量为 $i=|J|$，每个作业中的任务数分别为 $k_i=|T_i|$，集群中包含的GPU数量

© 中国科学院软件研究所 http://www.jos.org.cn

为 $m=|G|$, 那么流网络的顶点数量为 $|V|=2+|J|+\sum_{i=1}^{|J|}(k_i+1)+m$, 其中, 2 代表开始和结束顶点, $|J|$代表作业数量, $\sum_{i=1}^{|J|}(k_i+1)$ 代表作业中的所有任务数量以及每个作业中的不调度顶点, m 代表 GPU 设备数量. 边的数量分为 3 个部分: 第 1 部分是开始顶点到作业顶点的边为$|J|$、GPU 设备到结束顶点的边为 m 和作业顶点到任务顶点的边的数量为 $\sum_{i=1}^{|J|}k_i$; 第 2 部分是任务到 GPU 设备顶点之间的边, 最大可能的边的数量为 $m\times\sum_{i=1}^{|J|}k_i$; 第 3 部分是不调度顶点与作业之间的边和不调度顶点与结束顶点的边与为 $2\times|J|$. 所以总的边的数量为 $|E|=3\times|J|+m+(m+1)\times\sum_{i=1}^{|J|}k_i$. 所以, 构造流网络的代价与 GPU 设备数量和任务数量相关.

流网络的最小代价最大流计算中, 假如顶点数量为$|V|$, 边的数量为$|E|$, 可用 GPU 数量为 m, 即流网络的可行流, 那么按照最短路径计算方法, 其复杂度为 $O(m|E|\log(|V|))$.

从上述的分析看, m 个 GPU 设备, t 个任务, 一次调度的规模大约是 $O(t^3m^4+mt)$. 当任务数量较大, GPU 集群规模较大时, 调度时间就变得较大, 所以该调度方法的大规模扩展性较差.

5 实验设计

系统在一个集群上进行实验, 集群中包含 15 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 颗 Xeon2.1 处理器, 每个处理器包含 8 个核, 32GB DDR4 内存, 2 块 RTX2080TI GPU 卡, 每个 GPU 卡 10GB 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, Mesos 的基础版本为 1.6. 这些计算机被组织在 4 个机架内, 每个机架包含 4 台计算机. 机架内的计算机通过机架换机连接, 各个机架交换机通过级联方式与汇聚交换机连接.

本文中的软件采用 CUDA 架构和 Mesos 资源管理框架.

(1) 对于作业框架, 在 Mesos 提供的样例基础上重新设计了针对数据处理的作业管理框架. 当 Mesos 向作业提供资源邀约后, 作业按照数据存储位置建立局部的流网络, 并计算各种可行方案对应的代价和容量. 当资源分配发送确认后的分配结果后, 各个作业向计算节点提交自己的任务;

(2) 资源管理方面, 改进了 Mesos 集群资源管理框架, 在 Master 节点上扩展了资源分配的策略以及 GPU 使用状态的管理. 当各个作业的局部流网络发送到资源分配器后, 资源分配器合并这些局部流网络, 形成全局流网络, 然后使用本文中的调度算法, 计算出最小代价最大任务数的调度结果, 最后向作业通知分配结果;

(3) 在计算节点上添加了 GPU 设备状态的监视和汇报机制, 同时还添加了数据远程访问服务, 用于远程节点对数据的存取操作.

每个作业包含两个主要的部分: 作业引擎和作业执行器. 作业引擎负责数据分片和任务描述的生成, 然后将任务描述发送到资源管理器并请求资源. 一旦获得计算资源, 资源管理器启动执行器, 执行器启动后从作业引擎获得任务描述信息并启动任务. 任务结束后, 执行器返回任务结果文件位置, 等待作业引擎的后续调度. 作业引擎与任务执行器之间通过网络通信交换任务状态和结果文件的存储位置.

针对数据分片部分, 实验过程中需要不断变换数据分片的大小, 本文中采用在每个计算节点启动一个文件服务的方式, 执行器的任务需要数据时, 连接计算节点的文件服务, 通过网络获得任务需要的数据. 作业启动前, 采用随机方式在不同的计算节点上复制各个数据文件的副本. 作业启动后, 作业引擎计算出每个任务所需的数据分片的大小和分片的偏移量, 任务根据这些信息, 按照代价最小原则从所有副本中选择一个最佳位置来获取数据. 全部任务执行结束后, 由作业引擎启动一个合并任务, 从各个任务的临时数据文件中获得数据, 然后由合并任务进行计算, 生成结果.

5.1 测试作业

本文中选择了一些典型的应用程序作为测试负载, 测试程序虽然不多, 但是每个程序都使用不同的数据

进行测试. 即使同一个程序, 读取的数据集不同时就可能产生多个应用程序实例, 每个实例是一个作业. 一个作业在实验中使用相同的数据集.

每次实验都运行多个作业. 采用不同的调度策略时, 作业数量以及作业的运行次序保持一致. 每次实验都并行运行 k 个作业, 一旦一个作业运行结束, 立即启动一个新的作业, 参与实验的作业有以下几类.

(1) 矩阵乘法(MM). 矩阵数据存储在两个大型文件中, 计算过程中采用分块矩阵, 即$(A_1,A_2,\dots,A_m)(B_1,B_2,\dots,B_m)^T=A_1B_1+\dots+A_mB_m$, 矩阵 A 垂直分块, 矩阵 B 水平分块, 分块数量一样, 每个分块矩阵相乘的结果再按照水平分块, 最后, 相同水平分块进行加法运算, 得到最后的结果. 结果是一个文件, 存储在指定的位置. 这个过程包括乘法运算和加法运算, 乘法运算的任务数量和加法运算的任务数量一样, 总的任务数是分块的 2 倍, 第 1 组测试使用两个相同的方阵进行乘法操作, 矩阵 A 的每个分块大小为 1 GB, 矩阵 B 的分块大小为 1 GB, 分块数量分别为 6, 10, 15, 20, 作业称为 J1, J2, J3, J4. 另外一组测试使用两个不同的矩阵, 矩阵 A 的每个分块大小为 1 GB, 矩阵 B 的分块大小为 60 MB, 分块数量分别为 8, 10, 20, 30. 作业称为 J5, J6, J7, J8. 共计 8 个作业;

(2) 稀疏整数出现次数计算(SIO). 作业执行时读取文件, 计算每个整数出现的次数. 随机生成的整数都限制在一定的范围内. 作业使用 MapReduce 计算方式, Map 阶段按照分片数量生成任务, 每个任务计算整数出现的次数. Reduce 阶段只有一个任务, 用于统计不同数字出现的次数. 由于随机生成的整数在一定的范围内, 所以 Reduce 任务使用 CPU 执行. 每个分片的数据大小为 3 GB, 分片数量分别为 5, 6, 8, 10, 15, 20, 30, 60. 作业称为 J9, J10, J11, J12, J13, J14, J15, J16. 共计 8 个作业;

(3) 素数的判别(prime). 作业读取数据并检查每个素数, 在生成素数时, 将素数的大小限制在 4 位数. 该作业的特点是具有较高的 GPU 使用率, 分片数据大小为 2 GB, 分片数量为 4, 8, 10, 15, 20, 30, 40, 60. 作业称为 J17, J18, J19, J20, J21, J22, J23, J24. 共计 8 个作业;

(4) 字符串排序(StringSort). 对于长度一定的字符串, 使用 GPU 进行排序. 首先对数据文件分片, 使得数据大小满足 GPU 显存大小的要求, 每个分片在单个 GPU 上执行, 产生(K,V)键值对数据, K 代表一部分按照前 4 个字符计算的大小值, V 代表字符串真实值. 按照 K 将数据分割成不同的区段, 同一区段的数据由一个 CPU 任务完成输出. 测试时, 每个数据的长度为 20, 数据分片大小为 2 GB, 分片数量分别为 5, 6, 8, 10, 30, 60. 作业称为 J25, J26, J27, J28, J29, J30. 共计作业数量为 6;

(5) 聚类算法(kmeans). 聚类作业分为两个部分: 一个是参数服务, 一个是任务执行. 作业执行时, 首先将数据分片, 根据分片结果得到作业中任务的数量. 另外, 参数服务随机产生聚类中心. 作业启动后, 根据分配的 GPU 数量, 启动执行器并执行任务. 任务执行完后, 计算出新的局部聚类中心, 然后发送到参数服务器, 由参数服务器计算新的聚类中心. 再执行相同的计算过程, 直到设置的迭代次数满足要求. 测试时, 数据分片大小为 500 MB, 数据维度为 2 500, 聚类数为 64, 迭代次数为 10, 分片数量分别为 5, 8, 10. 作业称为 J31, J32, J33. 共计作业数量为 3;

(6) 图片检测(Yolo v3). 使用 DarkNet[31]提供的图片检测程序以及神经网络结构和权重参数, 对大规模的图片进行并行检测. 作业启动后, 根据分片数量将图片分片, 每个分片为一个任务. 实验中使用了 1 万张图片, 测试时分片数量分别为 5, 10, 20, 每个分片包含的图片数量为 2 000 张、1 000 张和 500 张. 作业称为 J34, J35, J36. 共计作业数量为 3.

5.2 测试标准

本文使用以下的指标来评价调度策略的效果.

(1) 作业运行时间

在实验中, 每次调度需要完成 m 个作业, 从第 1 个作业被调度到最后一个作业完成, 跨越的整个时间区间称为作业运行时间, 记为 DT. DT 用来描述作业并行运行中出现拖后的程度: 当 DT 越大, 说明存在作业拖后, 造成作业执行周期拉更大; 当 DT 越小, 说明并行执行的作业能够同时完成.

(2) 单个作业完成时间

© 中国科学院软件研究所 http://www.jos.org.cn

如果有多个作业共享集群资源，当作业 j 通过调度后，从第 1 个任务开始到最后一个任务结束，其花费的时间称为单个作业完成时间，记为 t_j^{sh}. 单个作业完成时间用来描述作业在共享资源时的延迟情况，时间越长，延迟越严重.

(3) 公平性偏差

当存在 m 个作业共享集群资源时，对于作业 j，当其独占 $1/m$ 的集群资源，从第 1 个任务开始到最后一个任务结束，其花费的时间称为理想作业完成时间，记为 t_j^{id}. 理想作业完成时间与单个作业完成时间的比值称为公平率，记为 s_j，$s_j = t_j^{id} / t_j^{sh}$. 由于调度开销，作业资源分配冲突等，$t_j^{sh}$ 一般花费更多的时间，故 $0<s_j<1$. 当 s_j 越接近 1 时，越能表明调度算法的公平性.

当 m 个作业共享集群资源时，公平率的算数平均值记为 $S=\sum_{i=1}^{m} s_i / m$，其均方差称为公平性偏差，记为 $\sigma=\sqrt{((s_1-S)^2+(s_2-S)^2+\ldots+(s_m-S)^2)/m}$. 公平性偏差用于衡量调度算法资源分配的公平性：方差越小，说明公平性越好.

(4) 网络数据传输大小

作业执行完成后，统计非本地化调度的任务的个数，再根据每个任务通过网络阐述的数据的大小，计算出该调度策略下网络数据传输的规模，从而估计出数据本地化带来的效果.

因为无法使用单个指标来评价算法，本文使用 4 个指标来进行对比. 当一个作业单独使用集群执行时，执行时间为 t; 而 m 个作业同时使用集群时，执行时间不超过 $m\times t$. 当一个作业运行时，需求的资源不超过全部资源的 $1/m$ 时，即使有 m 个作业共享集群计算资源，那么其运行时间仍然为 t. 最坏情况下，当一个作业执行时，需要全部的集群计算资源，那么 m 个作业同时运行时，其作业执行时间为 $m\times t$.

作业的公平性偏差是评价调度器的指标，用来评价调度器的公平性. 作业运行时间则用来评价系统的吞吐量，可以用来显示不公平的分配策略对吞吐量的影响. 单个作业完成时间则描述了作业的规模对于完成时间的影响. 网络数据传输大小描述作业运行中本地化调度的程度，不同的策略和调度算法数据传输代价的影响会不同.

5.3 实验评价

在本节，我们针对不同的网络条件运行一系列的作业，对比基于 GPU 数量和基于最小代价最大任务的调度算法在性能方面的不同：第一，针对测试的应用程序，使用抢占策略的共享调度时，其性能比其他策略的性能都好；第二，我们比较了基于 GPU 数量和基于最小代价最大流的调度算法，后者具有更好的性能以及较低的网络使用率；最后，大幅度地降低数据传输代价会得到较好的吞吐量.

5.3.1 作业运行时间的基准性能

为了对比调度的性能，需要计算每个作业独占资源时的基本性能. 但是调度策略不同时，其获得的数据也不一样，所以我们比较 3 种算法在作业独占资源时的运行时间，选择完成时间最小值作为基准.

为了获得基准数据，每个机架都使用其全部 1 GB 带宽链接到中央交换机，每个作业独立运行，在这种情况下，各个作业的运行时间以及网络数据传输大小，用作业运行时间最小的数据作为基本数据. 测试过程中，我们每次运行了 36 个作业，共测试 3 次，分别获得了基于 GPU 数量的共享分配算法(GS)、基于最小代价最大任务的共享分配算法(FS)以及采用抢占策略的最小代价最大任务数算法(FSP)的性能数据，其结果如表 1 所示.

表 1 理论运行时性能指标

算法	作业运行时间(单位: s)	数据传输大小(单位: TB)		
		合计	机架内	机架之间
GS	825.6	1.3	0.325 (25%)	0.195 (15%)
FS	808.2	1.3	0.299 (23%)	0.156 (12%)
FSP	782.3	1.3	0.208 (16%)	0.117 (9%)

从表 1 可以看出: FSP 算法的效果最好，而且偏差也较小，平均值大约为 1.8%. 所以，我们选择 FSP 算法

© 中国科学院软件研究所 http://www.jos.org.cn

的各个性能指标作为比较基准.

我们观察到：即使作业独占资源运行，FSP 也可以显著改善跨集群数据传输，从而缩短运行时间. 在依据 GPU 设备数量进行公平共享的方式中，一旦有空闲资源，即使数据传输代价较大，任务也会被提交到 GPU 上. 而 FSP 调度过程中，某些任务由于数据传输代价过大而会被延迟调度，一直到合适的计算资源出现，任务才开始调度.

5.3.2 网络受限时作业性能评价

实验中，各个机架交换机采用 1 GB 交换机，汇聚交换机也采用 1 GB 交换机. 在实验中，我们针对不同调度策略来实施. 实验设置并行运行的作业数量为 6，该并行度可以确保作业运行的整个期间有足够多的任务来保持集群资源的高利用率. 每次实验都至少进行 3 次，然后取得平均值作为最终的结果数据. 图 5 给出了每个作业在不同的调度策略下的单个作业完成时间，同时给出了作业的并行度 k 为 1 时，各个作业的理想完成时间，用来对比单个作业完成时间的差异.

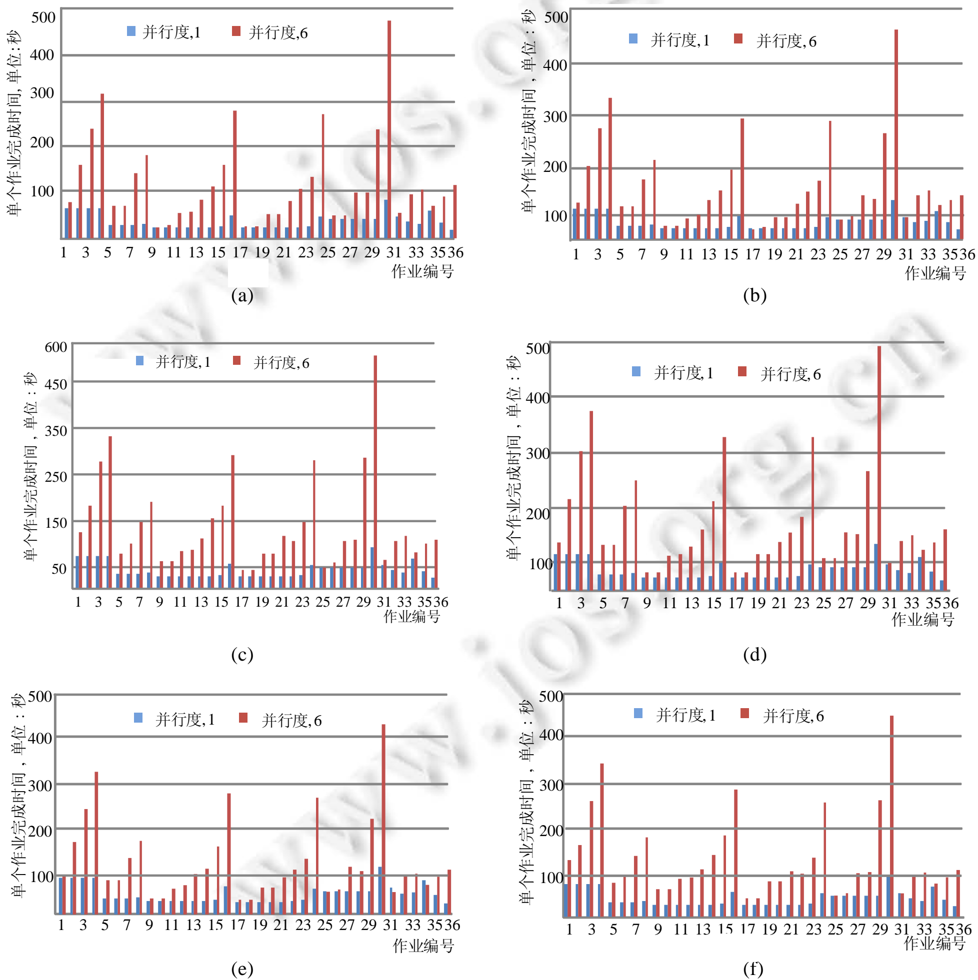

图 5 不同策略下单个作业完成时间

图 5(a)–图 5(c)是基于 GPU 设备数量的公平调度算法时，全部作业的单个作业完成时间以及理想作业完成时间. 测试单个作业完成时间时，也测试了同时存在 6 个作业并行运行的完成时间. 图 5(a)使用非抢夺策略，图 5(b)使用抢夺策略，图 5(c)使用延迟调度策略. 图 5(d)–图 5(f)是基于最小代价最大任务数的调度算法时，全部作业单个作业完成时间以及理想作业完成时间. 图 5(d)是不采用抢夺策略，图 5(e)是使用抢夺策略，图

© 中国科学院软件研究所 http://www.jos.org.cn

5(f)是不考虑公平策略. 从图中可以看出: 当作业的规模比较大时, 单个作业完成时间明显增大. 原因在于: 多个作业并行运行时, 每个作业只能得到一部分集群资源, 由于作业中包含的任务较多, 一些任务因为等待及资源被推迟. 从作业中第 1 个任务开始到最后一个任务完成, 与独占资源(k=1)相比, 时间区间明显加大. 说明作业之间存在资源竞争时, 调度系统按照基本资源配额来提供资源. 对于作业规模小的作业, 其基本资源配额可以满足任务的需求, 所以满足任务的执行需要, 故单个作业完成时间基本上与理想作业完成时间相同. 对于不同的调度策略, 单个作业完成时间也不相同. 抢夺方式单个作业完成时间最小, 延迟策略完成时间稍微长一点, 不考虑公平性的单个作业完成时间也较长, 但是差别不是太大. 其主要原因是: 由于任务计算过程中的数据规模比较大, 虽然作业中的任务被延迟调度, 但是尽量满足了数据本地化要求, 减少了数据的传输, 使得最终完成时间延迟不太大. 值得注意的是: 在抢夺策略中, 由于将一些非本地数据传输的任务强制终止, 使得其他作业的本地任务增大, 明显降低了单个作业完成时间; 而在非抢夺方式中, 虽然减少了部分任务被强制停止而造成的计算损失, 但是受数据的非本地化因素影响, 导致最终单个作业完成时间并未减少.

图 6 是各种调度策略下, 作业针对不同的调度策略的性能对比.

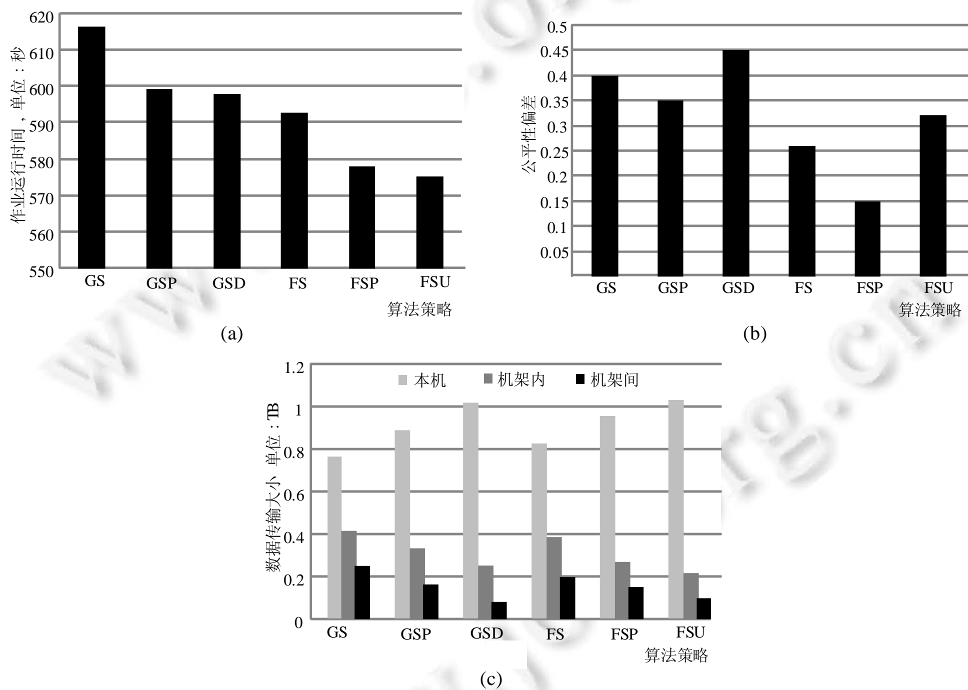

图 6 网络受限时性能指标

图 6(a)–图 6(c)分别比较了 6 种调度策略在作业并行运行时的作业运行时间、公平性偏差以及数据在网络间传输规模. 从图 6(a)中可以看出: 在最小代价最大任务数的算法中, FSU 效果较好. 这是因为该算法只考虑数据本地化, 减少了数据在网络间的传输, 所以全部作业运行结束花费的时间相对较少. 也就是说, 单位时间内能够执行更多的作业, 提高了资源吞吐率. 从图 6(c)也看到: 该策略使得机架间与机架内数据传输相对较少, 但是它的公平性偏差较大, 达到 0.32, 与抢夺方式比较, 差别有 25%左右. 同样, 基于 GPU 设备数量的调度方法中, 延迟调度算法带来作业并行运行时间减少的同时, 使得公平性有所降低.

结合图 5 和图 6 的比较我们可以看出: 当作业并行运行时, 只考虑数据本地化, 必然影响单个作业的完成时间, 但是全部作业的运行时间会有一定程度的降低. 从这些数据也可以得出结论, 数据的传输代价对于作业的运行时间影响还是比较大. 通过对公平性与数据本地化读取的折中处理, 能够得到较好的调度效果.

© 中国科学院软件研究所 http://www.jos.org.cn

5.3.3 网络不受限时的性能评价

为了测试不同的网络带宽对性能数据的影响, 实验时将汇聚交换机替换为 10 GB 交换机, 而各个机架交换机保持原来的配置. 在实验中, 使用第 5.3.2 节同样的测试作业, 进行了同样步骤的测试. 图 7(a) –图 7(c)分别比较了 6 种调度策略在作业并行运行时的作业运行时间、公平性偏差以及数据在网络间传输规模. 从图 7(a)中可以看出: 网络带宽的提高, 使得整体作业的运行时间都得到一定程度的减少. 所以, 网络带宽的增加有助于提高吞吐率. 从图 7(b)看到, 公平性偏差变化不大. 说明网络带宽的扩大, 对于资源分配的公平性影响较小. 从图 7(c)看到: 机架之间的数据传输量变大, 机架内的数据传输量变化不大, 而本地读取数据的大小有一定的减少. 这说明网络带宽的增加, 对数据传输代价产生影响.

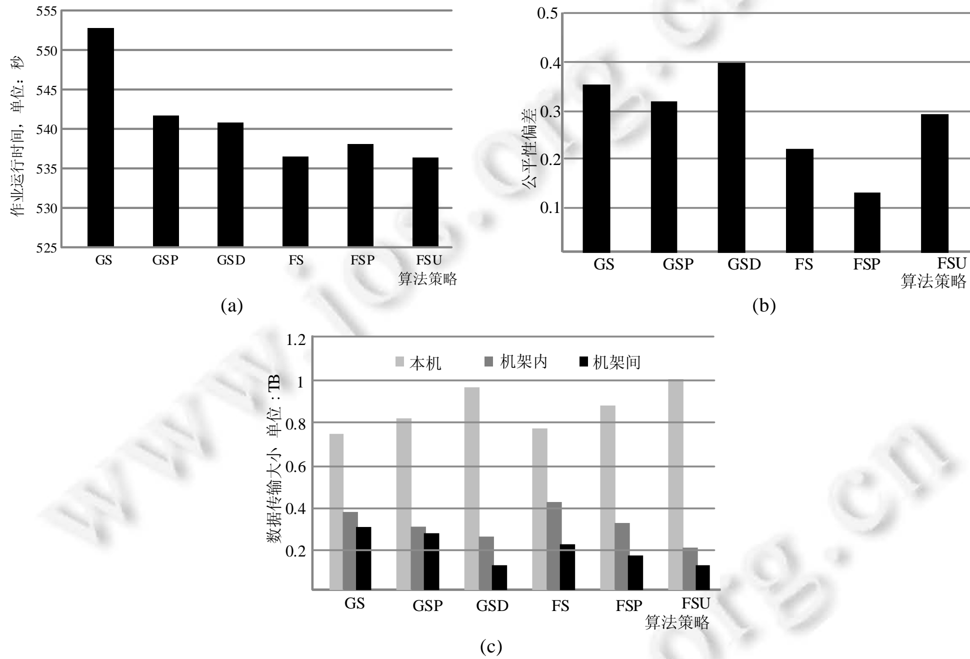

图 7 网络不受限时性能指标

5.3.4 补充测试

考虑到 J1–J36 为数据密集型作业, 现随机选取一个稀疏矩阵作业, 验证当作业所需数据量较小时, 最小代价最大任务数算法的调度效果. 该作业一共包含 3 个任务, 每个任务所需数据大小为 28 M, 任务所需数据相互独立, 互不干扰. 作业称为 J37. 测试时, 将 J1–J37 分别投交到使用 FSP, FS 和 FSU 算法的集群中, 验证作业并行度为 6、网络带宽分别为受限和不受限时算法的调度效果. 测试发现, 上述算法测试用时与图 6、图 7 测试用时相差不大. 这是由于当作业所需数据量较小时, 网络带宽对作业数据传输代价的影响较小, 进而导致该作业对集群整体运行时间的影响较小. 由此可以得出结论: 当使用上述算法运行数据量较小的作业时, 调度效果不会显著改变.

5.4 一体化调度与传统调度的对比

为了对比用户级别的公平与任务级别公平的不同, 本文模拟了 Yarn 中的容量调度、公平调度和本文中的一体化调度. 容量调度中使用先来先服务策略, 容量调度和公平调度的数据访问的优先级别设置 3 类: 本地、机架内、机架间. 本文设置了 10 个用户, 针对每个用户, 随机从 36 个作业中选择 6 个作业, 作为当前用户需要执行的作业, 每个用户模拟一个队列, 分配等量的 GPU 资源. 当所有用户的作业都完成后, 计算花费的时间. 进行多次测试后, 计算花费时间的平均值, 一体化调度花费的时间比公平调度少 9.52%, 比容量调度少 11.27%. 其原因是: 本文的调度是从全局的角度考虑数据访问情况, 而容量调度和公平调度则是从局部的角

© 中国科学院软件研究所 http://www.jos.org.cn

度考虑数据访问情况; 另外一个原因是, 各个用户中作业的执行时间和资源需求也存在差异. 所以, 基于最小代价最大任务数的调度方法优于公平调度和容量调度.

5.5 调度开销的估计

基于最小代价最大任务数的调度算法中, 建立顶点和边的算法复杂度为 O(V+E), 其中, V 代表流网络顶点集合, E 代表流网络边的集合. 流网络的最小代价最大流计算是经典的过程, 代价相对较大. 本文对调度的额外开销进行了估计, 实验评价过程中的平均调度开销为 5.04 ms, 最大情况下不超过 10.23 ms. 这些数据说明, 调度开销不是算法的瓶颈. 另外, 为了估计更大规模的调度开销情况, 我们模拟了 2 000 个 GPU 设备的集群, 包含 100 个并行运行的作业, 调度时间大约在 1–2 s 之间. 这对于数据密集型计算作业的运行时间来说, 是用户可以接受的范围. 调度开销的数据说明, 使用最小代价最大任务数调度方法是可行的.

6 结 论

GPU 计算资源在数据密集型计算中越来越得到重视和使用, GPU 集群资源共享也是目前发展的趋势. 随着越来越多的作业需要 GPU 计算, GPU 资源在不同的作业之间的共享显得越来越重要. 但是, GPU 的共享带来数据在计算节点之间大量传输的问题. 本文通过最小代价最大任务数量调度算法, 解决了现有算法中在 GPU 资源公平分配与数据传输代价的矛盾, 通过对数据密集型作业进行并行执行测试, 验证算法能够保证 GPU 资源的公平性达到 92%左右. 但是, 算法在减少数据传输规模上还有待进一步的提高.

References:

[1] Kindratenko VV, Enos JJ, Shi GC, *et al*. GPU clusters for high-performance computing. In: Proc. of the 2009 IEEE Int'l Conf. on Cluster Computing and Workshops. 2009. 1–8.

[2] Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal. ACM Sigplan Notices, 2012, 47(8): 117–128. [doi: 10.1145/2370036.2145832]

[3] Zhao B, Zhong J, He B, *et al*. GPU-accelerated cloud computing for data-intensive applications. In: Proc. of the Cloud Computing for Data-Intensive Applications. New York: Springer, 2014. 105–129.

[4] Che S, Boyer M, Meng J, *et al*. Rodinia: A benchmark suite for heterogeneous computing. In: Proc. of the 2009 IEEE Int'l Symp. on Workload Characterization (IISWC). 2009. 44–54. [doi: 10.1109/IISWC.2009.5306797]

[5] 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, 2011. 295–308.

[6] Hadoop capacity scheduler. 2021. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html

[7] Acuña P. Deploying Rails with Docker, Kubernetes and ECS. New York: Apress, 2016. 27–68. [doi: 10.1007/978-1-4842-2415-1]

[8] 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]

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

[10] Karanasos K, Rao S, Curino C, *et al*. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In: Proc. of the 18th USENIX Annual Technical Conf. Berkeley: USENIX, 2015. 485–497.

[11] 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]

[12] 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. New York: Association for Computing Machiney, 2018. 127–136. [doi: 10.1145/3149457.3155313]

[13] The apache software foundation. 2016. https://issues.apache.org/jira/browse/YARN-5517

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

[15] Garg S, Kothapalli K, Purini S. Share-a-GPU: Providing simple and effective time-sharing on GPUs. In: Proc. of the 25th Int'l Conf. on High Performance Computing. IEEE, 2018. 294–303. [doi: 10.1109/HiPC.2018.00041]

© 中国科学院软件研究所 http://www.jos.org.cn

[16] Xu Q, Jeon H, Kim K, *et al*. WarpedSlicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In: Proc. of the 43rd Annual Int'l Symp. on Computer Architecture (ISCA). IEEE, 2016. 483−496. [doi: 10.1109/ISCA.2016.29]
[17] Rhu M, Gimelshein N, Clemons J, *et al*. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In: Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO). IEEE, 2016. [doi: 10.1109/MICRO.2016.7783721]
[18] Yu H, Rossbach C. Full virtualization for GPUs reconsidered. In: Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking, 2017.
[19] Gu J, Chowdhury M, Shin K, *et al*. Tiresias: A GPU cluster manager for distributed deep learning. In: Proc. of the 16th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX, 2019. 485−500.
[20] Mahajan K, Balasubramanian A, Singhvi A, *et al*. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In: Proc. of the 17th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX, 2020. 289−304.
[21] Nvidia cuda C programming guide. 2021. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[22] Zhao X, Yao J, Gao P, *et al*. Efficient sharing and fine-grained scheduling of virtualized GPU resources. In: Proc. of the 38th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS). IEEE, 2018. 742−752. [doi: 10.1109/ICDCS.2018.00077]
[23] Wu B, Chen G, Li D, *et al*. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In: Proc. of the 29th ACM on Int'l Conf. on Supercomputing. New York: ACM, 2015. 119−130. [doi: 10.1145/ 2751205.2751213]
[24] Park JJK, Park Y, Mahlke S. Chimera: Collaborative preemption for multitasking on a shared GPU. ACM SIGARCH Computer Architecture News, 2015, 43(1): 593−606. [doi: 10.1145/2786763.2694346]
[25] Menychtas K, Shen K, Scott ML. Enabling OS research by inferring interactions in the black-box GPU stack. In: Proc. of the 2013 USENIX Annual Technical Conf. Berkeley: USENIX, 2013. 291−296.
[26] Fan Z, Qiu F, Kaufman A, *et al*. GPU cluster for high performance computing. In: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing. IEEE, 2004. 47−58. [doi: 10.1109/SC.2004.26]
[27] Cuda_Wrapper project. 2015. https://sourceforge.net/projects/cudawrapper/
[28] Stuart J A, Owens JD. Multi-GPU MapReduce on GPU clusters. In: Proc. of the 2011 IEEE Int'l Parallel & Distributed Processing Symp. IEEE, 2011. 1068−1079. [doi: 10.1109/IPDPS.2011.102]
[29] Liu J, Hegde N, Kulkarni M. Hybrid CPU-GPU scheduling and execution of tree traversals. In: Proc. of the 2016 Int'l Conf. on Supercomputing. New York: ACM, 2016. 1−12. [doi: 10.1145/2925426.2926261]
[30] Zaharia M, Chowdhury M, Das T, *et al*. Resilient distributed datasets: A faulttolerant abstraction for in-memory cluster computing. In: Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX, 2012. 15−28.
[31] Darknet: Open source neural networks. 2020. https://pjreddie.com/darknet/

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.

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.

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.

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

© 中国科学院软件研究所 http://www.jos.org.cn