计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(11):54−62 [doi: 10.15888/j.cnki.csa.007166]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

面向任务调度优化的分布式系统信息管理框架①

胡亚辉[1], 朱宗卫[2], 刘黄河[2], 王　超[1]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230027)
[2](中国科学技术大学 软件学院, 苏州 215123)
通讯作者: 胡亚辉, E-mail: huyhcs@mail.ustc.edu.cn

摘　要: 近年来深度学习作为学术界与工业界共同关注的热点, 取得了飞跃式的发展, 在计算机视觉、语音识别等领域取得了令人瞩目的成果. 深度学习分训练与推理两个阶段, 在实际应用中主要关注的是推理阶段. 深度学习推理过程中伴随着巨大的计算量, 通过分布式系统提高其计算速度也得到了越来越多的关注. 然而, 构建分布式深度学习推理系统面临着深度学习加速设备更新迭代快速、上层应用及计算任务复杂多样等挑战. 本文设计并实现的系统信息管理框架, 用于收集并处理系统中的各类信息, 收集及处理的规则具有高度的可扩展性和灵活性, 并提供通用的 RESTful API 数据访问接口, 以支持分布式深度学习推理系统对各类硬件加速器的灵活兼容性以及对任务调度策略的动态调整能力. 最后, 本文通过一个应用实例对该框架的功能进行验证并对实验结果进行分析.

关键词: 分布式系统; 深度学习推理; 任务调度; 系统信息管理

引用格式: 胡亚辉,朱宗卫,刘黄河,王超.面向任务调度优化的分布式系统信息管理框架.计算机系统应用,2019,28(11):54–62. http://www.c-s-a.org.cn/1003-3254/7166.html

System Information Management Framework of Distributed System for Task Scheduling Optimization

HU Ya-Hui[1], ZHU Zong-Wei[2], LIU Huang-He[2], WANG Chao[1]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
[2](School of Software Engineering, University of Science and Technology of China, Suzhou 215123, China)

Abstract: In recent years, deep learning, as a hotspot of common concern in academia and industry, has made great progress and achieved remarkable achievements in computer vision, speech recognition and other fields. It is divided into two stages: training and inferencing. In practical application, the main concern is the inferencing stage. The process of deep learning inferecing is accompanied by a huge amount of computation, and more and more attention has been paid to using distributed system to improve its computing speed. However, the construction of distributed deep learning inferencing system is faced with the challenges such as rapid updating and iteration of deep learning accelerators, complex of applications and computing tasks. The information management mechanism proposed in this study is used to collect and process all kinds of information in the distributed system, and the rules of collection and processing are highly customizable and flexible. It also provides a universal RESTful API data access interface to support the flexible compatibility of various hardware and the dynamic adjustment ability of task scheduling strategy in the deep learning inferencing system. Finally, we verified the function of the mechanism through an example and analysed the experimental results.

Key words: distributed system; deep learning inference; task scheduling; system information managing

① 收稿时间: 2019-04-26; 修改时间: 2019-05-23; 采用时间: 2019-05-27; csa 在线出版时间: 2019-11-06

1 引言

深度学习作为机器学习的一个分支, 近年来取得了飞跃式的发展, 在众多领域如计算机视觉、语音识别等都取得了令人瞩目的成果. 以 AlexNet 将 ImageNet 数据集上的图像识别 top-5 准确率由 73.8% 提高至 84.7% 为标志, 深度学习中所使用的神经网络模型开始朝着更深、更复杂的方向发展, 随后又出现了 VGG 和 ResNet 等更加复杂的网络模型[1]. 然而, 伴随着使用更加复杂的网络来实现准确率的提高, 传统的 CPU 已经远远满足不了其性能需求, 因此, 以 GPU 为代表的具有高度数据并行性的计算设备逐渐被用于深度学习计算的加速. 除 GPU 以外, 还有众多基于 FPGA[2–5]或基于 ASIC[6,7]的深度学习加速器. 这些工作都大大提升了单节点上深度学习推理计算的性能, 然而在面临海量数据处理的应用场景, 单个节点的性能仍显不足. 一直以来分布式系统都是提供计算力的重要途径, 因此也被用作深度学习推理的加速平台, 并与深度学习加速器相结合使用. 例如, 谷歌早在 2015 年就部署了 TPU 集群专用于加速神经网络的推理过程[8]. 然而, 构建分布式深度学习推理系统依然面临着如下挑战. 第一, 以 GPU 为代表的深度学习加速器如今正处于快速发展阶段, 不断涌现的新型硬件使得系统必须具有高度灵活的硬件兼容性以适应其快速的更新迭代; 第二, 任务调度对系统整体性能影响显著, 而不同深度学习应用之间所呈现的计算和访存特征差别巨大, 因此系统必须具有灵活调整任务调度策略的能力. 为应对分布式系统软硬件环境的动态性以及各类深度学习应用特点的多样性, 本文设计并实现了可扩展的系统信息管理框架, 支持对系统信息收集策略和处理策略的定制化, 一方面提高系统对各类深度学习加速器的兼容性, 另一方面使得分布式深度学习推理系统具有根据软硬件环境以及实际应用的特点动态调整调度策略的能力.

2 背景与动机

2.1 分布式系统下任务调度的研究现状

分布式系统中任务调度起着至关重要的作用, 选择合适的调度算法有利于提高系统整体的资源利用率和吞吐率, 从而提升系统性能. 因其重要性, 分布式系统调度算法一直是分布式系统研究领域的一个热点问题. 随着异构计算平台的产生和发展, 集群中硬件资源越来越丰富多样, 各类芯片对于不同类型的计算在性能、功耗上表现各不相同, 尤其是对于深度学习类应用, 不同计算部件的处理能力相差巨大, 因此在面向大规模数据集的深度学习推理的场景下, 考虑针对具体的应用特点进行任务负载划分以及任务调度策略的设计是非常有必要的. 此外, 从用于加速深度学习推理的硬件的发展角度来看, 此次人工智能热潮中, 涌现了大批的深度学习加速硬件及平台, 其中有代表性的有谷歌的 TPU、寒武纪公司的智能芯片系列、微软公司的基于 FPGA 的深度学习加速平台 BrainWave[9]等. 深度学习加速硬件的快速迭代迫使分布式系统应该对于新型计算资源具有更好的兼容性, 然而现今的分布式计算系统中一般对这些新型资源的支持都不够友好, 如 Hadoop 的资源管理器 Yarn 在默认情况下仅支持对 CPU、内存、硬盘等资源的管理[10]. 为了适应硬件的快速迭代, 用于深度学习的分布式推理系统应该支持对资源种类的易扩展性, 并且在任务调度时应根据系统资源以及作业特点的变化动态调整任务调度策略.

近年来, 每年有上百篇与分布式系统下的任务调度问题相关的论文发表, 然而据统计, 2005 年至 2015 年期间发表的 1050 篇论文中, 22% 从未被引用过, 在所有引用中, 超过 60% 仅来自其中 12% 的文章[11]. 这足以说明目前大部分的研究成果属于一次性工作. 并且如此大量的研究成果的发表也增加了后来研究者的困难, 为此, 目前有大量的关于分布式系统任务调度问题的综述性文章, 对该领域的研究脉络进行梳理, 以期为研究人员提供理论基础. Lopes 等人中从工作负载、资源、调度需求三个维度出发, 并进一步对每个维度进行细分, 对分布式系统中的调度问题和解决方案进行分类以及形式化描述[11], 来帮助研究人员方便地对之前的研究成果加以利用. Gautam 等人针对 Hadoop 集群中的任务调度常见算法 (FIFO、Fari、Capacity、Delay 等) 从多个方面进行了归类总结, 包括是否支持任务优先级、资源共享是否公平、适用环境为同构还是异构、任务分配策略为动态还是静态等, 分析各算法的优势和劣势[12].

诸如此类的综述性文章为以后的研究工作提供了一定的理论基础, 但是对于最大规模的重复利用已有成果, 仍远远不够. 究其原因, 系统规模扩大、软硬件资源复杂、应用负载多样性等因素都为分布式系统下的任务调度带来了更大的挑战, 设计合理的调度算法必须将大量的因素考虑在内. 然而, 考虑大量因素所带

来的大量重复的细节工作使得调度算法的设计难以进行, 目前大多数算法都是针对少量具体的影响因素进行设计或优化. 例如, Hammoud 等人考虑了数据局部性以及网络传输对任务的影响, 对 MapReduce 框架中的 Reduce 任务调度进行优化, 将任务放置在更靠近数据所在节点进行处理[13]. Arslan 等人综合考虑了文件访问代价以及 CPU 负载等因素, 做为 MapReduce 框架中 Reduce 任务调度优化的依据[14]. 这类工作由于出发点本身的局限性, 在设计时仅考虑具体的某一个或两个因素, 因而难以适应集群软硬件环境以及工作负载的变化.

对现有的分布式系统下任务调度的相关工作进行总结可以发现, 目前有大量工作都是面向特定场景对调度算法进行优化; 另外有大量综述性的工作, 对调度问题进行抽象及形式化表述, 以更好地为研究人员提供理论上的依据; 但尚缺乏从如何简化调度器设计的角度出发的工作, 考虑将大量与算法核心设计无关的系统信息收集与处理的工作独立出来, 以提升分布式系统中调度算法研究工作的效率和质量.

2.2 分布式系统下任务调度关键步骤描述

分布式系统中任务调度可以看作是在满足某些约束条件的前提下, 将一个作业分解成为若干任务, 并将这些分解后的任务分配给一组处理单元进行处理的过程, 处理单元一般对应于一个能够完成任务处理的资源组合, 例如 CPU、内存和网卡的组合, 一个物理机、虚拟机、容器都可以看作是一个处理单元. 而调度策略则决定了如何划分作业以及各个划分后的任务在哪个处理单元于何时开始处理, 因此, 任务调度策略是整个调度器的最关键部分. 在实际设计中, 调度器通常会对多种信息进行综合分析, 包括系统中的软硬件资源、作业执行需要满足的指标、任务执行的历史信息等, 最终得到一个合理的调度策略. 分析的方法可以是某种启发式算法, 如蚁群算法、遗传算法, 也可以是神经网络等机器学习类算法. 上述过程如图 1 所示, 为了实现调度器的可用性和高效性, 需要解决两方面的挑战.

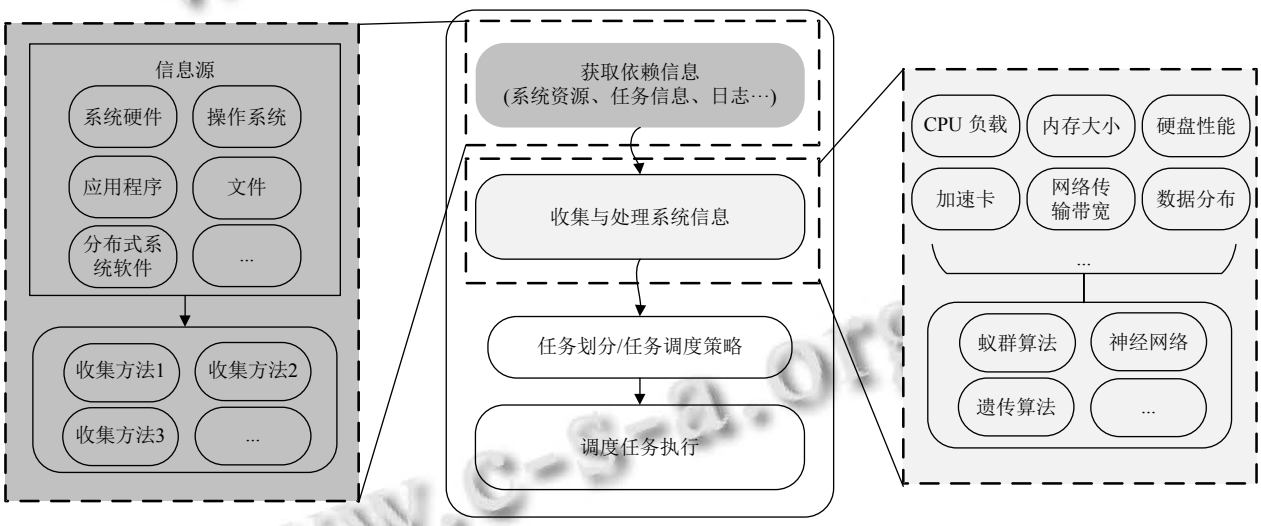


图 1 分布式系统任务调度主要过程示意图

(1) 生产信息的数据源以及信息本身的表示形式均丰富多样且容易随时间发生变化, 例如, 在集群的使用过程中, 随着机器的更新迭代, 系统中会加入各种各样的新型硬件或者不同类型的操作系统及基于此的各种软件系统, 调度器必须有获取并处理这些信息的能力. 即使这些信息的收集及处理在调度算法设计中并非核心问题, 但对调度器的实现以及实际的调度效果有重要的影响, 是否具有对某种信息的获取能力, 以及获取和处理的方法是否高效可靠, 则直接决定了所设计的调度算法是否具有可行性.

(2) 采用启发式或者机器学习算法对大量系统信息进行分析以寻求合理调度策略的过程具有极高的计算复杂度, 并且实现难度较高, 不仅使得此类算法在实际系统中的实现或者或者移植变得困难, 且可能成为系统性能的瓶颈, 这大大限制了各种复杂分析算法在调度器中的使用, 如何保证这部分计算逻辑的正确性以及健壮性值得深入探究.

3 框架设计与实现

本文所描述的系统信息管理框架的核心设计思想

是将系统信息收集与处理这两部功能的实现独立于调度器, 调度器只需通过 RESTful API 接口进行数据访问以获取所需信息, 信息的收集和分析处理分别由信息收集器和信息处理器负责, 信息收集器以及信息处理器均可独立优化且具有功能可扩展性, 从而保证系统可用性和高效性. 如图 2 所示, 收集器负责采集各种数据源生产的信息, 具体的数据源可以是操作系统、硬件设备、应用软件、日志文件或分布式系统软件等等, 对于不同数据源, 收集器可以根据数据源以及数据表示形式的不同进行设计和扩充, 以实现对不同类型信息的支持.

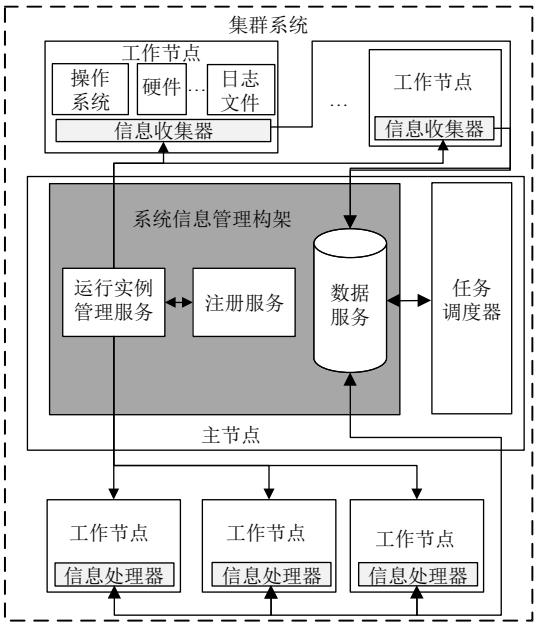


图 2 系统信息管理框架功能框图

信息处理器则是对收集的基础数据做进一步加工处理, 得到更具使用价值的信息. 例如, 将各节点 CPU 负载和节点间文件访问代价作为 MapReduce 框架中 renduce 任务调度的依据, 则可以使用信息收集器从操作系统获取各节点的网络带宽以及硬盘性能等相关信息, 从 MapReduce 框架中获取数据块的分布信息, 信息处理器使用这些信息计算出节点间数据访问的代价并通过数据管理服务提供的数据写入接口保存数据, 调度器仅需通过数据读取接口获取数据, 并依此完成任务调度, 使得大量复杂的信息获取和处理逻辑对调度器透明化.

系统信息管理框架的核心服务模块包括 3 个, 分别为:

(1) 注册服务, 负责完成信息收集器以及信息处理器的注册.

(2) 管理服务, 负责管理各信息收集器和信息处理器, 包括启动、停止、副本控制等.

(3) 数据服务, 负责完成数据的存储和读写, 收集器和信息处理器产生数据之后通过数据服务中的写入接口将数据写入存储系统, 信息处理器和调度器在需要使用数据时则通过数据服务的数据读取接口获取数据.

接下来分别对这三者的设计与实现进行详细说明.

3.1 信息收集器与处理器的注册

注册服务的核心功能是维护框架中所有的信息收集器与信息处理器的注册信息. 信息收集齐与处理器本质上都是采用特定方法获取所需的数据并生产供系统中其他模块所用的的数据, 可独立实现及优化, 具体实现可以是任意完成数据采集或处理功能的可执行文件. 例如, 一个信息收集器的功能是收集节点的内存使用情况, 则其实现方式之一是使用 shell 脚本读取 proc 文件系统中的数据来获取相关信息. 信息处理器与此类似, 不同的是其数据输入通常为某信息收集器或另一信息处理器的输出, 这些数据由数据服务维护并提供访问接口. 数据来源体现在具体数据处理器或收集器的程序设计中, 可以通过操作系统的接口、系统健康监测程序的输出文件或数据数据服务提供的数据访问接口等方式获取数据.

注册服务的功能如图 3 所示, 设计者通过命令行客户端或者在程序中使用注册接口向框架内添加一个收集器或处理器, 注册信息需包含数据对象, 以及获取该数据的信息处理器或者信息收集器的信息, 可以是某一个可执行文件的路径. 注册服务接收并处理注册信息, 并存入存储系统. 注册完成后, 任何程序可以通过注册服务提供的数据查询接口来获取当前系统中已注册的数据及其获取方式. 当某类数据的信息处理器或者信息收集器的设计者对其逻辑进行更改后, 通过更新接口对注册信息进行更新.

3.2 信息收集器与处理器的运行管理

信息收集器与处理器管理服务的功能是控制收集器与处理器的具体实力在各个节点的启动与停止, 决定了具体的收集器或者处理器的工作模式, 包括何时启动、何时停止以及收集或处理数据的时间间隔等, 这些信息在注册时已经指定.

运行管理服务的实现架构如图 4 所示, 管理服务

采用的是 Master-Slave 架构, Master 端程序负责通知运行与各个节点之上的 Slave 端程序进行运行实例的启动或停止, 二者之间通过 RESTful API 进行通信. Slave 端程序在接收到 Master 端程序所发送的信息收集器/处理器的启动命令后, 对启动命令进行解析, 获取该收集器/处理器所负责生产的数据的标识以及执行运行实例的命令. 之后, 为运行实例分配运行槽, 运行槽负责启动具体的运行实例并与其进行交互, 运行实例从数据源获取数据, 最后运行槽对所获取的数据进行包装并通过数据服务提供的数据写入接口对获取的数据进行保存.

3.3 数据管理

注册服务的核心功能在于维护了数据在节点上的获取方式, 在不同的节点上可能存在多个副本而产生多个具体数据, 对这些具体数据必须进行有效的组织与管理, 并提供相应的读写接口, 此即数据管理服务的功能. 如图 5 所示, 数据服务负责存储具体的数据, 提供数据的读写、更新等接口, 数据的使用者直接通过数据读取接口获取数据, 数据的使用者包括数据处理器和任务调度器等. 数据的生产者通过写入接口写入数据, 数据服务接收到写入请求后, 对请求进行解析后将获取的信息写入存储系统.

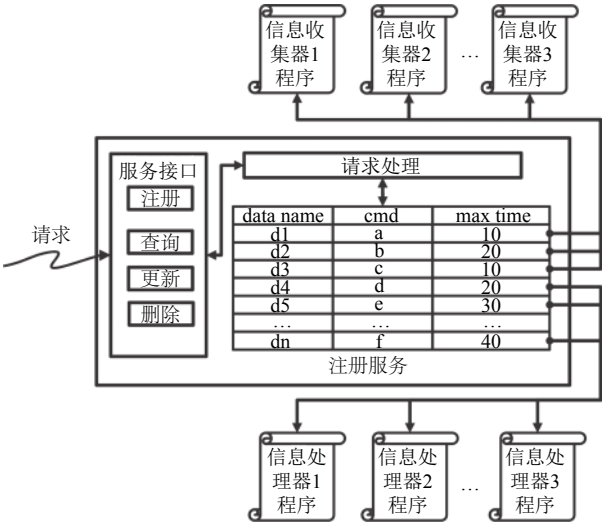


图 3 注册服务功能示意图

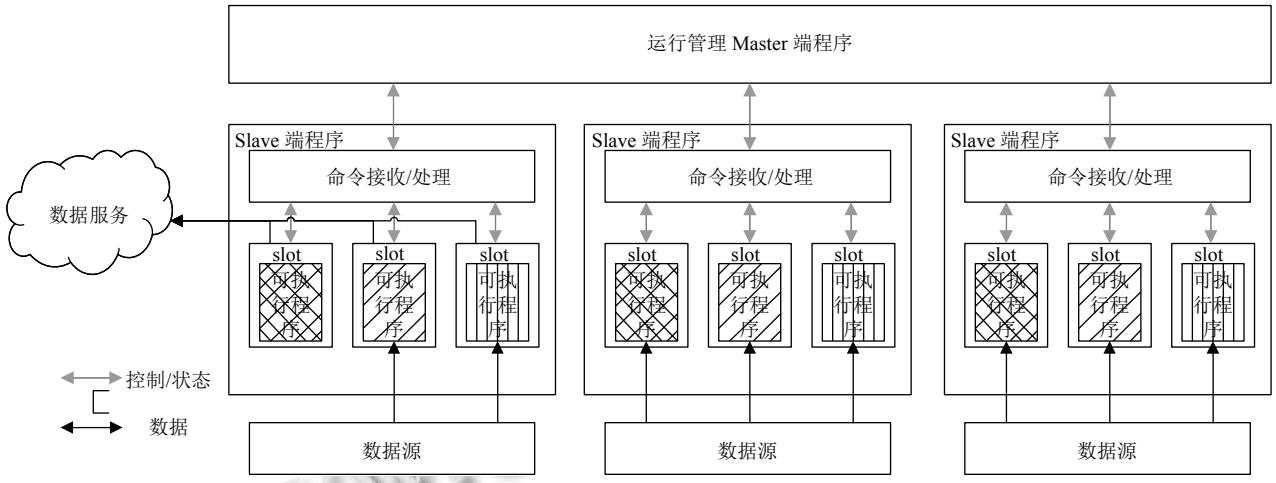


图 4 运行管理服务功能示意图

系统信息管理框架的核心功能及设计目标之一是支持调度器灵活高效的获取所需数据以灵活调整其调度策略. 而数据服务能否提供高效的数据访问机制则决定了整个系统信息管理框架的可用性. 在设计中我们采用了异步的数据访问机制来保证数据访问的高效性. 所谓异步的数据访问机制, 是指将数据的收集与处理和数据获取异步进行. 通常情况下, 当任务调度器需要某些数据时, 会通过一定方式临时性地从系统中获取基础信息, 再通过一系列的处理最终获取所需的数据. 这样做的好处是可以保证数据的准确性和有效性较高, 然而伴随着的是较长的数据获取时间, 尤其是在所需的信息量非常大的情况下, 并且在分布式系统的环境中, 还需要面临各种不确定性. 设计所采用的异步数据访问机制中, 各运行槽的数据写入过程和调度器的数据访问过程完全分离. 这虽然牺牲了一定的数据准确性, 但提升了调度器数据访问的效率和稳定性.

3.4 原型系统实现与接口设计

在构建原型系统的过程中, 主要使用了 SpringBoot 框架. Spring 框架是一个开源的用于企业级应用开发的编程框架, SpringBoot 是由 Pivotal 团队开发的用于

简化 Spring 应用的初始搭建以及开发过程. 依赖于 SpringBoot 我们可以较快地实现各个服务模块的功能并对外提供 RESTful API 接口, 接口设计如表 1 所示. 本次原型系统的设计中, 数据存储通过轻量级的关系型数据库 mysql 实现, 具体的, 包含表 t_Data、t_DP. 表 t_Data 用于存储数据, 包含 id、data_name、data_value 和 node_name 等列, 其中 data_name 为数据名称, data_value 为数据的值, node_name 为产生该条数据的节点名称, id 为 data_name 与 node_name 的组合作为主键; 表 t_DP 用于存储注册信息, 包含 data_name、cmd 和 time_max 等列, data_name 为所注册的信息收集器/处理器所生产的数据名称, cmd 为执行该信息收集器/处理器实例的命令, time_max 为两次运行之间的时间间隔.

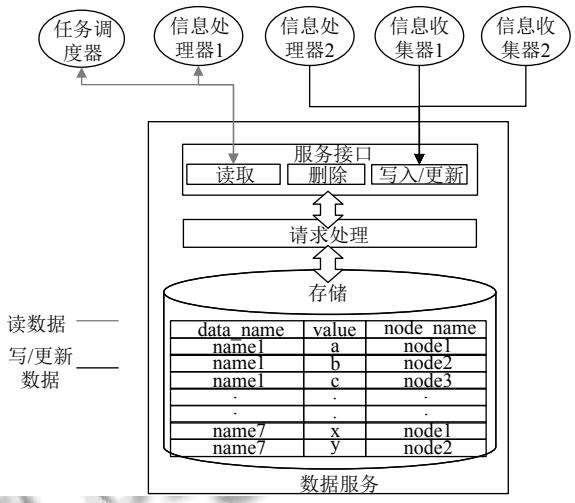


图 5 数据管理服务功能示意图

表 1 API 接口设计

服务模块	API 请求	请求主体	说明
注册服务	POST api/registry	{“dataName”: name, “cmd”: cmd, “max-Time”: maxTime}	创建一个信息收集器或者信息处理器, 产生的数据名称为 name, 通过cmd运行改信息收集器或者信息处理器, 运行间隔为 max-Time 秒
	GET/api/registr	-	获取当前所维护的所有信息收集器和信息处理器的信息
	GET/api/registry/{dataName}	-	获取负责生产名称为dataName的数据的信息收集器或者信息处理器
	PUT/api/registry	{“dataName”: name, “cmd”: cmd, “max-Time”: maxTime}	更新名称为name的数据的信息收集器或者信息处理器的运行命令以及运行间隔
	DELETE/api/registry/{dataName}	-	删除一个信息收集器或者信息处理器
数据服务	POST api/data	{“dataName”: name, “nodeName”: node, “dataValue”: ”data”}	新增 name 类型数据, 此数据来自于节点 node, 数据内容为 data
	GET api/data/?dataName={dataName}	-	获取 dataName 类型所有数据
	GET api/data/?dataName={dataName}&dataNode={node}	-	获取来自于 node 节点的 dataName 类型数据
	PUT api/data	{“dataName”: name, “nodeName”: node, “dataValue”: ”data”}	更新 name 类型下来自结点 node 的数据, 数据内容更新为 data
	DELET api/data/?dataName={dataName}&dataNode= {node}	-	删除来自结点 node 的 dataName 类型数据
	DELET api/data/?dataName={dataName}	-	删除所有 dataName 类型数据
运行管理	POST api/mng	{“dataName”: name, “cmd”: cmd, “max-Time”: maxTime}	通知运行管理服务注册模块的信息已经更新, 新添加了 dataName 型数据的收集器/处理器, 以及运行命令和运行间隔
	DELETE api/mng/{dataName}		停止并删除 dataName 数据的收集器/处理器的运行实例

4 应用实例

本节将通过一个具体实例介绍如何通过系统信息管理框架来辅助调度器完成更合理的调度. 分两个主要步骤: (1) 系统信息收集的可执行程序设计完成后, 通过注册模块提供的注册接口进行注册并生成数据访问接口. (2) 通过设计不同的信息处理逻辑来对所获取的数据进行加工, 并生成相应数据访问接口.

4.1 实验设置

为了说明本文所描述的系统信息管理框架对任务调度器的支持, 本次实验中, 针对分布式系统中使用深

度神经网络模型对含量图片进行分类处理的应用场景, 设计并实现了一个任务调度器, 其主要功能是对作业负载进行静态划分, 即将待处理的数据集划分为多个子集, 指定各个节点所需处理的数据子集. 实验中选取的实际任务为, 在具有 1 个主节点和 4 个工作节点的集群中使用基于 AlexNet 的物体分类程序对大批量图片进行分类处理, 数据集为从 ImageNet 中选取的包含 400 张图片的子集. 主节点及各个工作节点的配置为, 英特尔至强 W1505 型 CPU, 主频 2.53 GHz, 4 GB 内存, 各个节点配有千兆网卡, 节点之间通过万兆交换机互连. 图片存储于分布式存储系统 HDFS 中.

调度器采用的负载划分算法的主要步骤为:

算法 1. 调度器负载划分算法

输入: 数据集合 D, 节点集合 N ($Node_1$, ⋯, $Node_n$).
输出: D 的一个划分 $\{D_1, \cdots, D_n\}$, D_i 为节点 $Node_i$ 负责处理的数据子集, $D=D_1 \cup D_2 \cup \cdots \cup D_n$.

1. 得出数据在节点上的存储分布作为初始划分 $\{D_1, \cdots, D_n\}$;
2. 获取各个节点当前负载量 $\{L_1, \cdots, L_n\}$;
3. 获取每个节点的处理能力 $\{C_1, \cdots, C_n\}$;
4. 按节点 i 的处理能得出节点 i 的目标负载量
$L_i^o=\frac{C_i}{\sum_{k=1}^{n} C_k} \times \sum_{j=1}^{n} L_j$
5. 将 N 划分为 3 个子集
$H=\{N_i|L_i^o<L_i\}, E=\{N_i|L_i^o=L_i\}, L=\{N_i|L_i^o>L_i\}$
6. 若 $H=\varnothing || L=\varnothing$, 则算法结束, $\{D_1, \cdots, D_n\}$为划分结果, 否则进入步骤 7;
7. 从 H 中选取一个节点 N_h, 从 L 中选取一个节点 N_l
8. 分别计算 N_h 与 N_l 载的差值 Δ_h 与 Δ_l, 并将 N_h 的部分负载转移至 N_l 转移量为 $\min\{\Delta_h, \Delta_l\}$; 将达到理想负载的节点转移至集合 E; 跳转至步骤 6.

算法核心思想为, 步骤 1–2 先按照数据在节点上的分布对数据做初始划分, 得到节点初始负载量, 各节点负责所持有的本地数据进行处理; 由于各节点的数据量以及处理能力的不匹配, 需要对数据进行重新划分. 步骤 3–4 首先按照节点的计算能力占总计算能力的比值得出各节点的理想负载量, 步骤 5 按照节点的处理能力与理想负载量是否匹配将所有节点划分为 3 个子集, H 集合内节点负载过高、L 集合内节点负载过低、E 集合内结点负载程度较为理想. 步骤 6–8 循环多次, 将高负载节点的负载划分至低负载节点, 直到各节点之间达到负载均衡.

算法 1 的关键之处在于对各个节点的处理能力以及计算负载的评估, 直接决定了任务划分结果, 而节点处理能力的评估方法应是随着系统的实际情况变化的, 例如向系统中添加具有加速器的节点则会影响性能评估的方式, 原先的算法可能会丧失其有效性. 而对计算负载的评估则应结合具体任务而定. 通过系统信息管理框架中的信息收集器以及信息处理器的修改来完成节点性能评估以及负载评估, 算法步骤 2 中仅需通过固定的 API 接口获取评估结果, 从而在避免修改调度算法的前提下完成调度策略的动态调整.

4.2 信息管理框架的应用

在不对调度算法做任何修改的前提下, 通过采用不同的信息收集器及信息处理器来实现不同的节点性能评估方法, 以适应实际需求. 实验中设计了 3 种评估方法, 分别如表 2 所示.

对应于 3 种评估方法, 需分别实现相应的数据收集器或者处理器, 并向系统完成注册. 在此, 我们以 CPU 使用率作为评估指标为例进行说明.

4.2.1 信息收集器实现

信息收集器可以通过任意编程语言或工具进行实现, 这里使用 Linux Shell 脚本进行实现, 关键代码如图 6 所示, 通过读取/proc 文件系统的信息获取 CPU 状态, 进一步计算并输出 CPU 总体使用率. 脚本设计完成并进行功能正确性验证后, 分发至各个节点的统一路径下, 这里假设在/usr/Apps/bin 路径下, 则运行该脚本的命令为/usr/Apps/bin/getCpuUsage.sh. 用于收集数据的脚本部署完毕之后, 通过注册服务的注册接口完成注册. 注册完成后, 管理模块运行各个节点之上的脚本程序, 并将获取的数据通过数据服务的写入接口写入数据表, 而其他任意程序则可通过该数据的访问接口获取数据.

4.2.2 信息处理器实现

针对使用 CPU 负载为评估指标的节点性能评估方法, 我们在本次实验依据式 (1) 对节点的处理能力进行计算. $CpuFree$ 为处理器当前的总体空闲比率, $CpuReq$ 为程序对 CPU 的需求占 CPU 总体性能的比率, 得出的性能结果为相对值, 1 为最好.

$$Perf=\begin{cases}1, & CpuFree \geq CpuReq \\ \dfrac{CpuFree}{CpuReq}, & CpuFree < CpuReq\end{cases} \quad (1)$$

在具体实现上, 流程如图 7 所示, 首先通过数据管理服务提供的 RESTful API 风格的数据访问接口获取当前各个节点的 CPU 负载, 再根据式 (1) 对节点的处理性能进行评估, 最后将评估结果输出.

表 2 评估方法

评估指标	说明
处理器配置	按照处理器配置进行节点性能评估, 本次实验中所使用的 4 个工作节点的 CPU 配置相同, 如表 2 所示.
CPU 负载	使用 CPU 当前负载作为节点性能评估的指标, 具体的性能评估使用式 (1).
历史运行记录	各个节点在进行完一次处理之后, 向日志文件写入当次处理的平均 fps, 使用最近一次运行的 fps 作为节点性能评估指标.

```
#! /bin/bash

time=$(date "+%Y-%m-%d %H:%M:%S")

LAST_CPU_INFO=$(cat /proc/stat | grep -w cpu | awk '{print
$2,$3,$4,$5,$6,$7,$8}')

LAST_SYS_IDLE=$(echo $LAST_CPU_INFO | awk '{print $4}')

LAST_TOTAL_CPU_T=$(echo $LAST_CPU_INFO | awk '{print
$1+$2+$3+$4+$5+$6+$7}')

NEXT_CPU_INFO=$(cat /proc/stat | grep -w cpu | awk '{print
$2,$3,$4,$5,$6,$7,$8}')

NEXT_SYS_IDLE=$(echo $NEXT_CPU_INFO | awk '{print $4}')

NEXT_TOTAL_CPU_T=$(echo $NEXT_CPU_INFO | awk '{print
$1+$2+$3+$4+$5+$6+$7}')

SYSTEM_IDLE=`echo ${NEXT_SYS_IDLE}
${LAST_SYS_IDLE} | awk '{print $1-$2}'`

TOTAL_TIME=`echo ${NEXT_TOTAL_CPU_T}
${LAST_TOTAL_CPU_T} | awk '{print $1-$2}'`

CPU_USAGE=`echo ${SYSTEM_IDLE} ${TOTAL_TIME} | awk
'{printf "%.2f", 100-$1/$2*100}'`


echo {CPU_USAGE}
```

图 6 获取 CPU 使用率关键代码

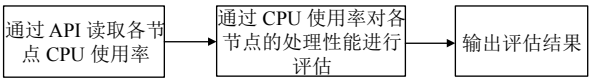


图 7 根据 CPU 使用率进行性能评估主要步骤

需要指出的是, 评估方法的改变, 对应的只是相关信息收集器与处理器的变化, 并不对算法的计算过程造成任何影响, 算法实现中仅是通过数据访问的 API 直接获取节点处理性能.

4.3 实验结果及分析

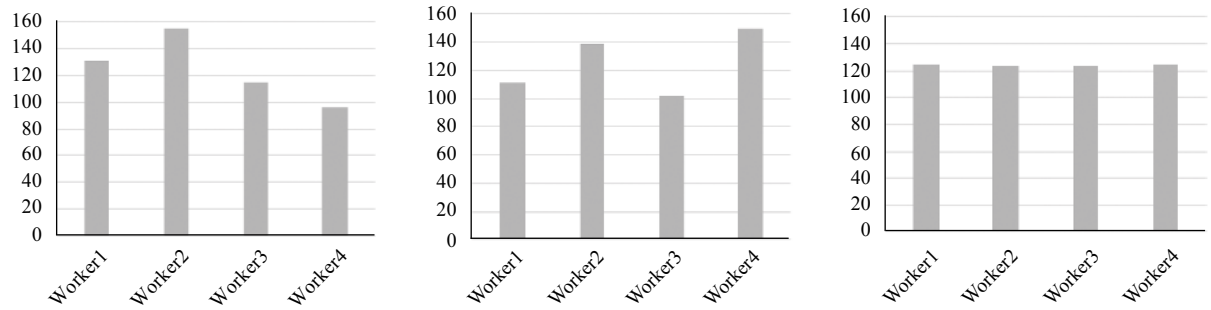
在采用 3 种不同节点处理能力评估方法的情况下, 对作业负载进行不同的划分, 各个节点的任务处理时间完成时间如图 8 所示. 可以看出, 在不修改调度算法的前提下, 通过改进节点性能评估方法而使得负载均衡的程度发生了改变. 在本次实验中, 各个节点的处理器配置相同, 但集群中运行的其他各类应用导致了各个节点的负载情况不同, 因此如图 8(a) 所示, 在采用按照处理器配置也即在各个节点间平均分配负载时, 效果并不理想.

如图 8(b) 所示是按照 CPU 负载来评估节点性能, 这种方法依赖于具体采用的计算公式, 需要合理分析应用对处理器的需求, 例如应用中是否针对多处理器进行了优化, 是否是计算密集型等. 通常采用这种方法, 需要综合考虑应用对处理器、内存、IO 的敏感程度而得出一个合理的性能评估公式. 在这次实验中, 我们的目的并不是为了寻求一个非常合理的公式, 而是为了说明通过我们提出的框架可以方便地对评估逻辑进行修改, 从而使得设计、开发和验证的效率更高.

从如图 8(b) 所示的实验结果来看, 该评估方法采用式 (1) 进行节点性能评估的效果并不理想, 但如前所述, 可通过重新设计该处理器的评估逻辑进行优化, 或启用另一个数据处理器, 直接从各个节点的历史运行信息中获取节点在运行该程序时的处理速度来评估节点性能, 这种方法的运行结果如图 8(c) 所示, 其较好的效果得益于在本次实验中, 即使各个节点之间的运行性能差异较大, 但节点各自的运行状态均比较稳定, 因此程序历史运行信息对下一次运行的处理速率有较高的指导意义. 这种方法在运行应用环境发生变化时很可能会变得不再适用, 这种情况下我们仍旧可以通过调整信息处理器的逻辑来适应新的变化.

5 结论与展望

本文介绍了面向分布式深度学习推理系统优化而设计的系统信息管理子系统, 该子系统的设计目的是为了将任务调度时需要的各类系统信息的收集工作从调度器设计中独立出来, 一方面是为了简化任务调度器的设计复杂性, 另一方面是为了提高调度器的灵活性. 当前的主流分布式系统能够提供的系统信息有限, 留给任务调度器的发挥空间不足, 如果设计者希望在调度中考虑复杂多变的系统信息, 这些信息的收集工作本身就会制约设计者的工作. 本文所描述的系统系统信息管理子系统支持灵活的功能扩展, 设计者可以通过对信息收集器与处理器的定制化来获取所需的系统信息. 同时在设计时, 通过 Restful API 接口对外提供服务, 保证了平台及语言无关性.



(a) 按 CPU 配置 (b) 按 CPU 负载 (c) 按历史运行记录

图 8 不同节点性能评估方法下各节点的处理时间 (s)

参考文献

1 Wang T, Wang C, Zhou XH, *et al*. A survey of FPGA based deep learning accelerators: Challenges and opportunities. arXiv preprint arXiv: 1901.04988, 2018.

2 Wang C, Gong L, Yu Q, *et al*. DLAU: A scalable deep learning accelerator unit on FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 36(3): 513–517.

3 Zhang YW, Wang C, Gong L, *et al*. A power-efficient accelerator based on FPGAs for LSTM network. Proceedings of 2017 IEEE International Conference on Cluster Computing. Honolulu, HI, USA. 2017. 629–630.

4 Sun F, Wang C, Gong L, *et al*. A power-efficient accelerator for convolutional neural networks. Proceedings of 2017 IEEE International Conference on Cluster Computing. Honolulu, HI, USA. 2017. 631–632.

5 Lu YT, Gong L, Xu CC, *et al*. Work-in-progress: A high-performance FPGA accelerator for sparse neural networks. Proceedings of 2017 International Conference on Compilers, Architectures and Synthesis For Embedded Systems. Seoul, South Korea. 2017. 1–2.

6 Chen YJ, Luo T, Liu SL, *et al*. DaDianNao: A machine-learning supercomputer. Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. Cambridge, UK. 2014. 609–622.

7 Chen TS, Du ZD, Sun NH, *et al*. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. Salt Lake City, UT, USA. 2014. 269–284.

8 Jouppi NP, Young C, Patil N, *et al*. In-datacenter performance analysis of a tensor processing unit. Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture. Toronto, ON, Canada. 2017. 1–12.

9 Burger D. Microsoft unveils project brainwave for real-time AI. Microsoft Research Blog, Microsoft 2017. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

10 Extend the YARN resource model for easier resource-type management and profiles. https://issues.apache.org/jira/browse/YARN-3926.

11 Lopes RV, Menascé D. A taxonomy of job scheduling on distributed computing systems. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(12): 3412–3428. [doi: 10.1109/TPDS.2016.2537821]

12 Gautam JV, Prajapati HB, Dabhi VK, *et al*. A survey on job scheduling algorithms in big data processing. Proceedings of 2015 IEEE International Conference on Electrical, Computer and Communication Technologies. Coimbatore, India. 2015. 1–11.

13 Hammoud M, Sakr MF. Locality-aware reduce task scheduling for MapReduce. Proceedings of the IEEE 3rd International Conference on Cloud Computing Technology and Science. Athens, Greece. 2011. 570–576.

14 Arslan E, Shekhar M, Kosar T. Locality and network-aware reduce task scheduling for data-intensive applications. Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds. New Orleans, LA, USA. 2014. 17–24.