计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.210900117

减少 Hadoop 集群中网络队头阻塞的调度算法

田冰川　田　臣　周宇航　陈贵海　窦万春
南京大学计算机科学与技术系　南京 210023
(bctian@smail.nju.edu.cn)

摘　要　大数据分析系统的用户希望任务的执行时间尽可能短。然而,在任务执行期间,网络与计算时刻都可能成为阻碍任务执行的资源瓶颈。通过对大数据分析系统的观察与分析,得出如下结论:1)根据当前资源瓶颈的不同,数据并行框架应当在多种工作模式之间切换;2)子任务的调度应当充分考虑将来可能到达的新任务,而不能仅考虑当前已经提交的任务。基于上述观察,设计并实现了全新的任务调度系统 Duopoly,其由感知计算资源的网络调度器 cans 与感知网络资源的子任务调度器 nats 两部分组成。通过小规模物理集群与大规模仿真实验对 Duopoly 的效果进行评估,实验结果表明,与现有工作相比,Duopoly 可以将平均任务完成时间缩短 37.30%～76.16%。

关键词:Hadoop 集群;队头阻塞;网络调度;任务调度
中图法分类号　TP393

Reducing Head-of-Line Blocking on Network in Hadoop Clusters

TIAN Bing-chuan,TIAN Chen,ZHOU Yu-hang,CHEN Gui-hai and DOU Wan-chun
Department of Computer Science and Technology,Nanjing University,Nanjing 210023,China

Abstract　Users of big data analytics systems want the execution time of tasks to be as short as possible. However,during task execution,both network and computational moments may become resource bottlenecks that hinder task execution. Through the observation and analysis of the big data analysis system,the following conclusions are drawn:1) the data-parallel framework should switch between multiple working modes depending on the current resource bottlenecks;2) the scheduling of subtasks should fully consider the new tasks that may arrive in the future,not only the currently submitted tasks. Based on the above observations,a new task scheduling system Duopoly is designed and implemented,which consists of two parts:cans,a network scheduler that senses computational resources,and nats,a sub-task scheduler that senses network resources. The effectiveness of Duopoly is evaluated by small-scale physical clusters and large-scale simulation experiments,and the experimental results show that Duopoly can reduce the average task completion time by 37.30%～76.16% compared with existing work.

Keywords　Hadoop cluster,Head-of-line blocking,Network scheduling,Job scheduling

1　引言

大数据分析系统的用户希望分析任务的执行时间尽可能短[1]。为了保障交互式数据查询的秒级甚至次秒级响应时间,数据并行框架(如基于内存计算的 MapReduce[2] 与 Spark[3]需要依赖于大规模内存并行处理。通常,数据并行框架将一个任务划分为若干阶段,并将每个阶段进一步分割为若干子任务,然后将这些子任务分配至某个集群的若干服务器。每个服务器将其可用的计算资源(CPU、内存等)划分为若干槽位(指一个 CPU 核心、一个 Docker 容器、一个线程等),每个子任务在执行期间占用一个槽位。连接服务器的下层网络,负责完成相邻阶段间的数据传递,将上一阶段子任务输出的执行结果传递至下一阶段作为其子任务的输入。我们将一个任务的完成时间定义为自其提交到全部子任务完成所经历的时间。

在任务执行期间,网络资源与计算资源均可能成为阻碍任务执行的瓶颈。对于给定的任务而言,其完成时间由任务执行期间可用的资源数量决定。我们将一个子任务的槽位占用时间定义为自其开始占用槽位并接收数据到完成计算并释放槽位所经历的时间,从而可以利用槽位占用时间来描述一台服务器中可用资源的总数目。一个子任务的生命周期可以分为 3 个阶段:第一阶段从子任务准备就绪(即其依赖的任务均已执行完毕)起,至其被实际分配槽位并开始接收数据为止,当集群计算资源处于高负载状态时,相对不充足的可用

到稿日期:2021-09-14　返修日期:2021-12-12
基金项目:广东省重点研发计划(2020B0101390001);国家自然科学基金(61772265,61802172,62072228)
This work was supported by the Key-Area Research and Development Program of Guangdong Province(2020B0101390001) and National Natural Science Foundation of China(61772265,61802172,62072228).
通信作者:田臣(tianchen@nju.edu.cn)

槽位将导致此阶段的耗时明显增长;第二阶段从子任务开始接收数据起,至其完成数据接收并开始计算为止,当集群网络资源处于高负载状态时,网络拥塞将阻塞数据输入,从而导致此阶段耗时明显增长[4-6];第三阶段从子任务开始计算起,至其完成计算并释放槽位为止,对于给定的槽位,该阶段的持续时间可以看作常量。

为了缩短数据并行框架下平均任务完成时间,我们需要从网络资源与计算资源两方面入手。如前文所述,一个子任务会因计算资源不足而在第一个阶段被阻塞,也会因网络资源不足而在第二个阶段被阻塞。一个子任务被阻塞的时间越长,其完成时间也就越长,进而导致任务完成时间的延长。当数据并行框架内同时存在多个任务时,该问题更加显著,这是因为任务间的资源竞争将加剧子任务阻塞对任务完成时间的影响。对于离线任务调度,即使仅存在单一资源瓶颈,该问题也为 NP 难问题。

已有工作通过减少网络拥塞引起的子任务阻塞来缩短任务完成时间,具体可以分为网络调度和任务放置两大类。基于网络调度的工作利用 Coflow 等应用层语义减少数据传输阶段的时间消耗[7-10];基于任务放置的工作利用在槽位分配期间将数据的局部性纳入考虑,从而通过减少网络传输数据量的方法来缓解网络拥塞,减少数据传输阶段的时间消耗[11-15]。这些工作均可以有效缩短任务的平均完成时间。

然而,已有工作均不能解决队头阻塞的问题。这些工作过于激进地将资源分配给已到达的任务,从而导致即将到达的任务因资源不足而长时间处于阻塞状态。基于网络调度的工作在优化网络指标的同时,导致计算槽位得不到充分利用;而基于任务放置的工作在分配槽位时忽略了网络因素,导致占用了槽位的子任务被网络阻塞而陷入空等。基于这些问题,我们发现任务的平均完成时间仍有进一步优化的空间。

通过对数据并行框架的观察与分析,我们得出如下结论。1)根据当前资源瓶颈的不同,数据并行框架应当在多种工作模式之间切换。当计算槽位充足时,调度系统应当致力于缓解网络拥塞;否则,调度系统应当致力于缩短子任务占用的槽位时间。子任务占用的槽位时间越短,因槽位不足而被阻塞在第一阶段的子任务就可以越快得到槽位,被阻塞的任务就可以越快完成。2)子任务的调度应当充分考虑将来可能到达的新任务,而不能仅考虑当前已经提交的任务。调度系统不为不能立即产生收益的子任务(如其数据传输链路目前处于拥塞状态)分配槽位,而是将槽位预留给未来可能到达的子任务,从而使调度系统的队头阻塞问题得到缓解。基于上述分析,我们认为,对任务平均完成时间的进一步优化是可行的。

在上述结论的指引下,本文重新探讨了数据并行框架下的任务调度这一经典问题,设计并实现了全新的任务调度系统 Duopoly,其由感知计算资源的网络调度器 cans 与感知网络资源的子任务调度器 nats 两部分组成。

当计算槽位不足时,cans 根据当前计算资源占用情况进行网络调度,从而减少子任务消耗在网络传输上的槽位时间。我们发现,已有的网络抽象不能满足网络调度的要求,如流级调度缺乏应用层语义,而 Coflow 级调度因粒度过大而不能反映子任务占用的槽位时间。对于一个子任务而言,被分配槽位后,其执行时间仅仅取决于接收数据所耗费的时间。因此,cans 将发往同一个子任务的所有数据流视为整体,作为基本的网络调度单位,并利用交换机物理队列对其实现网络内部的优先级进行排序。

与此同时,nats 根据 cans 的网络调度策略,对每一个等待分配槽位的子任务进行实时进度预测,以判断其被分配槽位后是否会因输入数据被网络阻塞而导致空等。对于短时间内无法产生显著执行进度的子任务,nats 将不会为其分配槽位,而是将槽位预留给未来可能到达的子任务,从而避免原本能够产生显著执行进度的子任务因槽位不足而被阻塞。此外,在网络调度信息的辅助下,nats 也可以在多个候选子任务间进行选择,从而在减少空等的同时,避免网络资源的浪费。

我们通过小规模物理集群与大规模仿真实验对 Duopoly 的效果进行评估,实验结果表明,与现有工作 Aalo 和 NEAT 相比,Duopoly 分别将平均任务完成时间缩短 37.65%~70.30%与 37.30%~76.16%,尾延迟与二者相比没有显著差异。与此同时,我们也在多种不同的场景下对 Duopoly 进行了评估,包括集群规模、槽位数目、系统负载等。实验结果表明,Duopoly 可以在绝大多数情况下取得良好的性能提升。

2 背景与动机

2.1 研究背景

在数据并行框架中,一个任务的执行过程可以被看作由若干阶段构成的有向无环图,相邻阶段间的有向边表示阶段间的数据传输。每个阶段可以被进一步分为没有依赖关系的若干子任务,每个子任务为一段程序,其通过网络读取远程数据或直接读取本地数据作为输入,执行程序定义的运算,并将执行结果存储在本地,然后结束运行。在执行期间,网络中存在多条数据流,将数据从源服务器传输至目的子任务。当一个任务的所有子任务均执行完毕后,该任务执行完毕。

相应地,每个服务器将其可用的计算资源(CPU、内存等)划分为若干槽位(指一个 CPU 核心、一个 Docker 容器、一个线程等),调度系统持续将空闲的槽位分配给各任务中等待执行的子任务,并收回已完成子任务所占用的槽位,以重新分配。对基于内存计算的数据并行框架而言,一个子任务所占用的槽位时间由网络传输和实际计算两部分共同决定;相比之下,其余开销(如磁盘读写)可以忽略不计。不失一般性,本文假设每个槽位的计算能力是均等的,即同一个任务在不同槽位中实际计算过程的耗时相同。

对于大数据分析系统而言,其负载在多个维度具有突发提交的特性,如任务数目、网络传输数据量、计算资源需求等,其峰均值比例可以大至为 31:1[16],这导致网络资源和计算资源均随时有可能成为整个系统的瓶颈。

为了简化问题,本文以 Hadoop 为例,对基于数据并行框架的大数据分析系统任务调度展开研究。典型的 Hadoop 任务由 Map 和 Reduce 两个阶段组成,两阶段间依靠网络进行数据传输。调度系统不对两阶段的子任务进行区分,一个计算槽位既可以被 Map 阶段的子任务占用,也可以被 Reduce 阶段的子任务占用。在多任务并发的场景中,我们可以认为一个任务的 Reduce 阶段仅在其 Map 阶段完全结束后才开始

执行，而不必考虑流水线对系统性能的影响[13]。一个机器槽可用于 Mapper 或 Reducer[14,17-18]。

类似地，我们认为一个子任务的实际计算仅在所有数据被完整接收后才开始；当子任务的计算结果完整写入本地缓存后，其占用的槽位方可被释放。Hadoop 的网络传输设计分为 3 个部分：Map 阶段的数据读入、Reduce 阶段的数据读入以及 Reduce 阶段的数据输出。已有研究工作表明，Map 阶段数据读入的本地化比例可以达到 99%[11]，而 Reduce 阶段的数据输出可以利用空闲网络完成[19]。因此，我们可以忽略 Map 阶段子任务对网络资源的占用，将其简化为仅具有计算过程的简单子任务；相比之下，Reduce 阶段的子任务则需要通过网络接收数据作为输入，然后执行计算过程。已有工作表明，Reduce 阶段的数据读入在集群网络流量中占据主导地位[11,14,20-21]，因此，本文将网络调度的重心置于 Reduce 阶段的数据读入上。

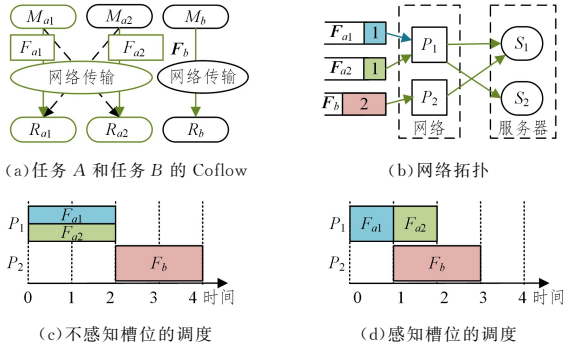
需要注意的是，我们以 Hadoop 为例的解决方案可以被扩展至其他的数据并行框架，如 Spark。Spark 的流水线机制允许一个子任务边接收数据边进行计算。当计算速率大于网络传输速率时，该子任务的计算时间可以视为 0；反之则可根据两速率的差值，对额外的计算时间进行近似估计，从而转化为与 Hadoop Reduce 阶段子任务类似的模式。

本文的系统设计方案面向以短作业优先（Short Job First，SJF）作为任务调度策略的私有集群，其调度系统不支持抢占，即当一个槽位被分配给具体的子任务后，调度系统既不能将子任务移动至其他槽位，也不能中断子任务的执行。此外，我们假设集群网络是非收缩的，即集群网络的核心层与接入层提供相同的总带宽。需要注意的是，虽然本文提出的方法并不针其他任务调度策略以及带有带宽收缩的网络，但实验结果表明，二者依然会因计算槽位的节省而受益。

对于感知网络的任务调度系统而言，网络信息的缺乏与迟滞是其面临的主要问题之一，在一个任务实际执行之前，调度系统很难判断该任务对网络资源的占用情况，这使得许多系统将网络信息视作完全未知[10,22]。然而在许多情况下，网络信息并不是完全未知的，例如：1）在 Hadoop 中，一个任务在 Map 阶段与 Reduce 阶段的子任务数目在任务提交时即可确定，两阶段之间的流量模式在 Map 阶段执行一段时间后即可以接近 100%的准确率得到估计[23-24]；2）大数据分析系统中的任务大部分是周期性提交的，此类任务的资源占用也是可以精确预测的[15]；3）对于同一类别的应用，其数据的输入输出比相对稳定，这使得调度系统可以根据当前阶段输入的数据量估计其输出的数据量，从而预测其对网络资源的占用情况[16,20]。基于上述方法，我们可以在任务提交时，根据资源占用规模对其进行粗略排序，实验结果表明，规模估计误差对调度系统的影响相对较小。

2.2 网络调度与槽位浪费

考虑如图 1(a)中 Hadoop 任务的例子。任务 A 由已经完成的 Map 阶段子任务 M_{a1}，M_{a2} 以及即将开始的 Reduce 阶段子任务 R_{a1}，R_{a2} 这 4 个子任务组成；其 Coflow 包含 4 条数据流，其中数据流 F_{a1} 与 F_{a2} 各有 1 单位的数据需要传输，另外两条数据流的传输量可忽略不计。任务 B 由已经完成的 Map 阶段子任务 M_b 以及即将开始的 Reduce 阶段子任务 R_b 2 个子任务组成；其 Coflow 仅包含 1 条数据流 F_b，该数据流有 2 单位的数据需要传输。这里我们重点分析网络调度对槽位资源的影响，因此假设所有 Reduce 阶段子任务占用的槽位时间是由网络传输主导的，而实际计算阶段占用的槽位时间是可忽略的。图 1(b)描述了此时集群网络使用情况。其中 P_1 和 P_2 为 Map 阶段子任务所在服务器的网络出口，其中数据流 F_{a1} 与 F_{a2} 通过 P_1 进入网络，数据流 F_b 通过 P_2 进入网络，两出口可分别以速率 1 进行数据发送。此时服务器有 2 个可用的槽位 S_1 与 S_2，分别分配给子任务 R_{a1} 与 R_{a2}；由于没有其他可用的槽位，因此子任务 R_b 处于等待状态。



(a)任务 A 和任务 B 的 Coflow (b)网络拓扑 (c)不感知槽位的调度 (d)感知槽位的调度

图 1 网络调度会影响槽位的可用性

Fig. 1 Network scheduling affects slot availability

无论采用何种网络调度策略，出口 P_1 与 P_2 的速率限制导致任务 A 与任务 B 分别至少需要 2 单位的时间来完成数据传输。由于任务 A 的所有子任务均已分配槽位，因此其完成时间仅由其网络传输时间决定，与调度策略无关；相反，任务 B 的完成时间则由网络调度决定。通过数据流调速来保障同一 Coflow 中的所有数据流同时完成是一种被广泛接受的网络调度策略，因为计算过程的开始依赖于 Coflow 内最慢数据流的完成，所以对过快的数据流主动降速可以避免对网络资源的过度占用[7,9]。如图 1(c)所示，这种策略将导致数据流 F_{a1} 与 F_{a2} 平分出口 P_1 处的带宽，从而使子任务 R_{a1} 与 R_{a2} 同时在 2 时刻执行完毕；这进而导致子任务 R_b 最早在 2 时刻得到槽位，并在 4 时刻执行完毕。此时，两任务的平均完成时间为(2+4)/2=3。

然而，上述两任务的平均完成时间可以通过网络调度进行优化。如图 1(d)所示，如果调度系统给数据流 F_{a1} 更高的网络优先级，则子任务 R_{a1} 就可以在 1 时刻执行完毕，并释放对槽位 S_1 的占用，使槽位资源紧缺的情况得到缓解。接下来，子任务 R_b 可以被调度至槽位 S_1，从而使任务 B 可以在 3 时刻执行完毕。注意，这一过程对任务 A 的执行没有负面影响，这是因为任务 A 的流量仍然在 2 时刻才能全部通过出口 P_1，子任务 R_{a2} 在 2 时刻执行完毕，从而使任务 A 的完成时间仍为 2。至此，两任务的平均完成时间减少为(2+3)/2=2.5。

基于上述例子，我们发现，现有网络调度策略在优化网络指标时会导致槽位被长期占用且低效利用的问题，从而引起槽位资源的浪费。

2.3 子任务调度与槽位浪费

考虑另一个 Hadoop 任务调度的例子。集群中有 2 个

服务器，每个服务器包含 2 个槽位，服务器收发数据的速率均为 1。任务 A 在 0 时刻提交，其包含 4 个 Map 阶段子任务 M_1-M_4 与 4 个 Reduce 阶段子任务 R_1-R_4。任务 B 在 2 时刻提交，其仅包含 1 个 Map 阶段子任务 M_5。我们将子任务 M_i 的输出数据记为 O_i，其输出数据保存在所在服务器的缓存中；每个子任务 M_i 均需要 1 单位的时间执行计算过程，计算过程执行完毕后槽位随即释放，缓存数据的进一步传输不再占用槽位时间。对于任务 A 而言，我们假设其两阶段间的每个子任务对 $\langle M_i, R_j \rangle$ 间均有 1 单位的数据需要传输，同时假设本地数据传输耗时相比网络传输可以忽略，其未在图 2 中标出。

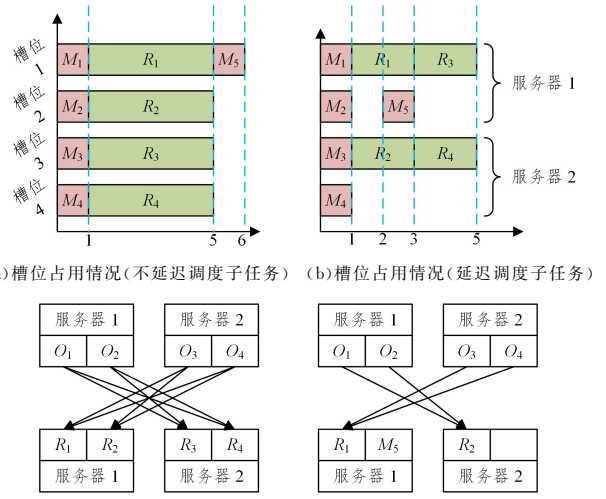


图 2 槽位预留可提升调度性能

Fig. 2 Slots reservation can improve scheduling performance

当计算槽位存在空余时，以 NEAT 为代表的多数子任务调度系统总会将其分配给某个等待调度的子任务，而这种策略有时反而会导致整体资源利用率低下。如图 2(a)所示，任务 A 在 0 时刻提交并进入 Map 阶段，其 4 个子任务 M_1-M_4 得到槽位；在 1 时刻，其 4 个子任务全部执行完成，任务 A 进入 Reduce 阶段，4 个子任务 R_1-R_4 得到槽位，并拉取 O_1-O_4 的数据。在 2 时刻，任务 B 提交，但此时所有的槽位都被占用，因此其无法立即执行，只能等待槽位释放；如图 2(c)所示，此阶段中，每个服务器的上下行网络带宽均被 4 条数据流平分，导致至少经过 4 单位的时间之后才有槽位被释放，因此任务 B 的子任务 M_5 最早在 5 时刻才能得到槽位，并在 6 时刻完成。此时，两任务的平均完成时间为 $(5+6)/2=5.5$。

上述策略导致任务 B 因计算资源缺乏而被长时间阻塞，而计算资源被任务 A 长期低效占用，其根本原因在于多流并行传输同时拖慢了所有 Reduce 子任务的执行进度。基于上述观察，我们发现，如果子任务调度期间能够提前感知网络状态，提前发现即将得到槽位的子任务会因网络阻塞而无法立即取得明显的执行进度，那么将槽位预留而不是立即分配将取得更好的结果。如图 2(b)所示，对于不能立即取得明显执行进度的子任务，为其分配槽位与否对当前任务的执行进度没有影响；此时，延迟调度将允许调度器等待未来可能到达的新任务，以获得更大的决策空间，从而增加得到更优解的机会。如图 2(d)所示，如果子任务调度器能够在 1 时刻发现仅调度 R_1 与 R_2 即可利用所有的网络资源，那么就可以延迟对 R_3 与 R_4 的调度，此时将有 2 个槽位被预留。这样，当任务 B 在 2 时刻到达时，M_5 可以立即被调度至预留的槽位中，使得该任务在 3 时刻完成。此时，任务 A 的完成时间并未受到任何影响，因此两任务的平均完成时间仅为 $(5+3)/2=4$。

基于上述例子，我们发现，现有子任务调度策略在实现槽位分配时，存在无效子任务挤占计算资源的问题，这同样会引起计算槽位的浪费。

3 系统设计概览

3.1 设计准则

在数据并行框架中，用户随时可能提交任务，而调度系统也会随时对子任务进行调度。基于上文提到的例子，我们可以得到调度系统设计的两个准则。

准则 1 根据当前资源瓶颈的不同，数据并行框架应当在多种工作模式之间切换。在图 1 中的 0 时刻，槽位已经被耗尽，新的子任务无法执行。在这种情况下，调度系统应当尽力加快正在执行的子任务的进度，从而尽早释放被占用的槽位，将部分计算资源留给正在等待的高优先级子任务。

准则 2 子任务的调度应当充分考虑将来可能到达的新任务，而不能仅考虑当前已经提交的任务。在图 2 中的 1 时刻，槽位虽然相对缺乏，但尚能满足当前任务的需要。此时，调度系统应当意识到新的任务是随时可能到达的，从而采取相对保守的调度策略，尽量避免槽位被浪费。只有减少槽位的低效利用情况，才有可能缓解新到达任务的队头阻塞现象。

在上述准则的指导下，本文对数据并行框架下的任务调度系统进行了重新设计。除上述准则外，实际可用的调度系统还应满足如下 3 个特性：1)分布式网络调度。由于集中式网络调度存在实时性低与可扩展性差的问题，因此改用分布式网络调度。2)快速子任务调度。子任务调度算法的计算复杂度应当尽可能低，否则其自身有可能成为新的系统性能瓶颈。3)对不准确的测量具有鲁棒性。与传统调度系统相比，本文所提方案需要利用网络信息与任务特性，二者的测量与估计可能存在误差或数据缺失的情况。调度系统的性能不应因数据不准确而大幅下滑。

3.2 系统架构

基于上述准则，本文设计并实现了全新的任务调度系统 Duopoly，其系统架构如图 3 所示。主节点维护着每一个任务的全局优先级，当一个任务被提交至系统时，主节点首先计算其优先级，并按照优先级将其插入至有一个序列表中。

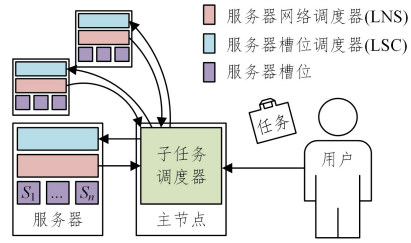


图 3 Duopoly 系统架构

Fig. 3 System architecture of Duopoly

Duopoly 系统的子任务调度器是对 Hadoop 调度器的

增强。子任务调度器负责监控槽位与网络使用情况以及当前进入系统的全部任务，以决定子任务调度时机与调度方案。当一个子任务得到槽位后，调度器将该子任务发送至对应的服务器，并有服务器负责在其槽位中执行该子任务。每个服务器负责监控其槽位内子任务的网络使用情况，当检测到子任务通过网络拉取外部数据时，服务器将根据数据流的优先级执行本地网络调度。此外，服务器将周期性监测本地网络利用率，并将监测数据发送至主节点。

根据槽位利用率的变化情况，主节点会向所有服务器发送模式切换信号；收到信号后，服务器将基于 cans 算法更新本地网络策略（详见第 4 节）。与此同时，主节点中的子任务调度器也会收到模式切换信号，并根据 nats 算法更新子任务调度策略（详见第 5 节）。

3.3 调度模式切换

Duopoly 系统通过缓解网络拥塞与节约计算槽位两种方法来提升调度器性能，分别对应于网络拥塞与计算资源不足两种情形下的调度模式。需要注意的是，上述两个优化目标并非是完全兼容的，因此调度时会出现二者无法兼顾的情况。为了尽可能缩短任务的平均完成时间，我们需要选择一种恰当的方式来实现两种模式的切换。

一个简单的方法是根据目前等待分配槽位的任务进行决策，如果目前空余的槽位不能满足所有就绪子任务的调度需求，则认为系统正处于槽位资源不足的状态。该策略具有一定的合理性，然而其与小任务优先的任务调度策略是不兼容的。如果当前正在运行的高优先级子任务已经占满了槽位，那么冒着拖慢高优先级任务的风险为低优先级任务预留槽位是不合理的。

基于上述考虑，本文以是否有优先级更高的小任务因槽位不足而被阻塞作为调度模式切换的信号，示例如下。假设任务 J_1 为目前系统中优先级最高的任务，其所有处于就绪状态的子任务均已被调度，此时系统处于槽位充足的状态。接着，优先级更低的任务 J_2 到达，即使系统中剩余的槽位无法容纳 J_2 中处于就绪状态的全部子任务，但此时系统仍然可以保持槽位充足状态，而没有必要切换至槽位不足的状态；否则，J_2 将可能体现出比 J_1 更高的优先级。随后，优先级最高的任务 J_3 到达，如果剩余的槽位无法满足 J_3 的要求，那么此时系统应当切换至槽位不足的状态，从而使 J_3 得到比 J_1 更高的优先级。这里的状态切换可以看作在不支持抢占的系统中模拟出任务抢占效果。文末的实验部分可以证明这种切换机制的有效性。

4 感知计算资源的网络调度

感知计算资源的网络调度（cans）与准则 1 相对应。当槽位充足时，cans 通过降低网络的拥塞程度来优化平均 Coflow 完成时间，此时优化 Coflow 完成时间与优化任务完成时间等价；当槽位不足时，cans 转而通过加速槽位释放为高优先级任务节省计算资源，以实现对平均任务完成时间的优化。本节首先提出供槽位不足模式使用的中等粒度网络抽象，接着介绍两种模式下基于 DSCP（Differentiated Services Code Point）的网络调度策略。

4.1 子任务调度与槽位浪费

在 Hadoop 中，我们可以认为一个子任务的实际计算仅在所有数据被完整接收后开始，且该子任务不依赖于同阶段的其他子任务。因此，我们应当将发往同一个子任务的所有数据流视为整体，作为基本的网络调度单位。该种网络抽象使得网络调度的粒度介于流级调度与 Coflow 级调度之间，我们称之为中等粒度网络抽象。使用中等粒度网络抽象意味着：1）对于发往同一个子任务的所有数据流，完成其中一部分是没有意义的，只有全部完成才能让子任务的执行进入下一阶段；2）像 Coflow 一样将发往一个阶段的所有数据流视为一个整体是没有必要的，即调度的目的不在于让一个阶段内的最慢数据流尽快完成。

我们将发往同一个子任务的所有数据流称为一个 Macroflow。一个 Macroflow 中的所有数据两两无关，即 Macroflow 内部不存在具有依赖关系的数据流。Macroflow 的完成意味着其所有数据流均已完成，完成时间即为最后一条数据流的完成时间。基于上述定义，我们可以将一个 Coflow 拆分为若干 Macroflow，各个 Macroflow 的目的地为互不相同的子任务。例如图 1(a)中任务 A 的全部流量可以视为一个 Coflow，也可以视为发往 R_{a1} 与 R_{a2} 的两个 Macroflow。

基于对 Macroflow 的观察，我们发现，对于一组给定的任务，最小化其累计消耗的槽位时间与最小化平均 Macroflow 完成时间是等价的。考虑 N 个 Hadoop 任务，其中第 i 个任务的 Reduce 阶段有 N_i 个子任务。考虑第 i 个任务的第 j 个子任务，其消耗在网络传输与实际计算上的时间分别为 $P_{i,j}$ 与 $Q_{i,j}$。此时，所有子任务累计消耗的槽位时间为：

$$\begin{aligned} T_{\text{slot}} &= \sum_{i=1}^{N}\sum_{j=1}^{N_i}(P_{i,j}+Q_{i,j}) \\ &= (\sum_{i=1}^{N}N_i)\left(\frac{1}{\sum_{i=1}^{N}N_i}\sum_{i=1}^{N}\sum_{j=1}^{N_i}P_{i,j}\right)+\sum_{i=1}^{N}\sum_{j=1}^{N_i}Q_{i,j} \\ &= c_1 \times T_{\text{Macroflow}} + c_2 \end{aligned}$$

其中，c_1 与 c_2 均为正数，分别表示累计子任务数目与累计消耗在实际计算上的时间。对于给定的任务而言，二者均可视为常量。因此我们可以把对槽位占用时间的优化转化为对平均 Macroflow 完成时间的优化，而后者可以通过网络内的优先级调度实现。

4.2 网络调度策略

当每个子任务被调度时，Duopoly 为其分配两个网络优先级，分别表示其在 Coflow 与 Macroflow 两种调度粒度下的优先程度。Duopoly 使用基于短作业优先（SJF）的调度策略对各子任务的数据传输进行排序，根据当前系统状态的不同，较小的 Coflow 或 Macroflow 具有较高的优先级。

基于短作业优先的网络调度有两种实现方式，即发送端调度或网内调度。发送端调度通过在应用层添加队列的方式，对等待发出的数据流按照优先级进行排序，其优点在于可以支持任意数目的优先级，而缺点在于无法感知网络内部的拥塞点。考虑两条数据流，其发送方不同而接受方相同，即使二者事实上具有不同的全局优先级，但只要二者在各自的发送方内均具有最高的本地优先级，其就会在网络中的某个拥塞点（或接收方）相遇并平分带宽，从而使优先级失效。

基于上述考虑，Duopoly 使用网内调度的方法来实现网络调度。基于网内调度的方法仅需要发送方在每条数据流的 IP 包头内指明其网络优先级，网络内每个交换机即可根据该优先级为数据包分配不同的硬件队列，并根据交换机配置来决定硬件队列间的优先级关系。与基于发送端调度的方法相比，该方法的优点在于可以在网络内的每个拥塞点上保证数据流的优先级顺序被满足，而其不足之处在于交换机的硬件队列数目是有限的。因此，我们需要将 Coflow 与 Macroflow 的大小分别划分为若干区间，落在同一区间内的两数据流具有相同的优先级。具体而言，我们使用一组指数递增的阈值来完成区间划分，该方法在大多数情况下具有与任意数目优先级类似的表现[10]。

需要注意的是，对于一条数据流而言，Duopoly 所使用的网络优先级是在发送前决定的，而不允许在发送过程中根据剩余大小实时调整。这一策略的选择与下文中的子任务调度策略相关。在一个子任务被调度时，调度器需要判断该子任务是否会被网络阻塞，并以此决定是否放弃调度。如果我们允许在子任务执行期间动态调整优先级，则网络行为将会变得难以预测。增加服务器的网络状态上报频率会加剧网络拥塞，而保持原有上报频率又会导致主节点信息过时，二者都会使系统性能产生极大的波动，其带来的性能损失大于潜在获益。因此，从稳定性的角度考虑，Duopoly 不允许网络优先级的动态调整。

5 感知网络资源的子任务调度

感知网络资源的子任务调度(nats)需要同时考虑准则 1 与准则 2。子任务调度器位于主节点内，其根据预估的规模维护全部活跃任务的优先级顺序，对网络资源与计算资源的利用情况进行实时监测，并结合当前任务与资源使用情况实现调度模式的切换。

5.1 调度框架

子任务调度的框架如算法 1 所示，根据系统状态的不同，调度器执行的具体策略也有所不同。如图 4 所示，根据槽位使用情况、网络占用情况、任务优先级、子任务阶段的不同，调度策略可分为 6 种情况。我们将在下文中对每一类调度策略展开具体的讨论。

算法 1 调度框架

```
1. L←按照长度升序排序后的活跃任务列表
2. Congestion←网络拥塞标识
3. Mode←当前调度器工作模式
4. while true do
5.   if Mode 为槽位不足模式 then
6.     Task,Server← 以 L 为参数调用算法 3
7.     else
8.       Task,Server← 以 L 和 Congestion 为参数调用算法 4
9.     end if
10.      if Task,Server 非空 then
11.      为 Task 分配网络优先级
12.      调度 Task 至 Server
13.      end if
14.      end while
```

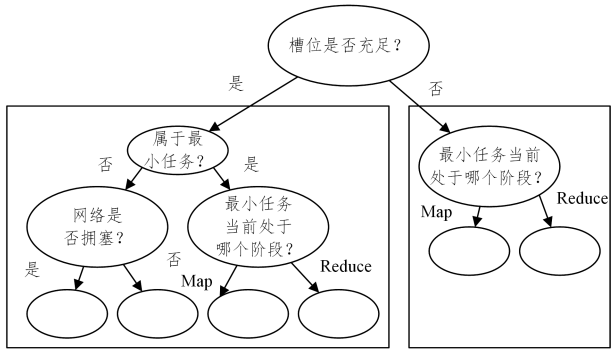


图 4 子任务调度框架

Fig. 4 Task placement framework

在下列两种情况下，子任务调度器将执行算法 1：1)当系统队列中存在就绪的子任务时，一个槽位被释放；2)当集群中存在空闲槽位时，一组子任务就绪。子任务调度器通过寻找一组空闲槽位与就绪子任务的匹配来完成一次调度。

如算法 1 所示，在一次调度过程中，调度器首先根据任务大小对所有活跃的任务进行排序，得到其优先级；然后读取网络拥塞标识与当前工作模式，二者均根据监控数据进行周期性更新；最后调度器根据当前工作模式(即槽位充足模式或槽位不足模式)执行不同的策略。当槽位不足时，调度器以节约槽位作为调度目标，以减少因计算资源不足而被阻塞的高优先级任务的等待时间；当槽位充足时，调度器以小作业优先(Small Job First，SJF)作为子任务调度策略，以缩短当前子任务的平均完成时间。若算法 1 返回空值，则说明当前槽位应当被预留，而不进行任何调度。在完成槽位分配后，调度器为每一个子任务计算网络优先级，并与子任务一同发送至槽位所在的服务器。

5.2 基本模块

对于 Hadoop 任务调度而言，子任务调度器 nats 的两种工作模式包含两个相同的基本模块，分别为 SelectTaskServerPair 与 SelectServerForTask。如算法 2 所示，第一个模块用于实现处于就绪状态的 Map 阶段子任务与空闲槽位的匹配。在 Hadoop 中，Map 阶段的输入数据被切分为多个大小相同的数据块，存储于多个服务器中，且每个数据块具有多个副本；同时，Map 阶段子任务的数据读取是一对一的，即一个子任务仅需读取一个数据块，且一个数据块仅能被一个子任务读取。这些特性使得 nats 有机会将此类子任务调度至其输入数据所在的分服务器中，以实现数据读取的本地化，从根源上减少网络流量的产生；此外，这些特性使得 nats 能够尽可能地平衡各服务器的负载，从而缓解网络拥塞现象。已有工作证明，利用延迟调度等方法，超过 99% 的 Map 阶段子任务可以实现输入数据的本地化[11]；在本地化失败时，nats 将直接选择负载最低的服务器来完成调度。

算法 2 基本模块

```
1. function SelectTaskServerPair(L)
2.   Local←L 中所有不需要通过网络读取数据的本地子任务
3.   if Local 非空 then
4.     Task,Server←Local 中候选服务器最少的子任务
5.     若有多个满足条件的 Task，则选择 Server 中本地子任务最少的
```

```
6.      若仍有多个满足条件的 Task,则选择 Server 中子任务最少的
7.      若仍有多个满足条件的 Task,则随机选择其中一个
8.    else
9.      Task,Server←随机选择子任务与负载最低的服务器
10.   end if
11.    return Task,Server
12. end function
13. function SelectServerForTask(Task)
14.   Server← 所有存在空余槽位且暂无网络访问的服务器
15.   if Server 为空 then
16.     S← 最小网络调度单位的长度大于 Task 对应长度的服务器
17.     Server←S 中正在访问网络的子任务数目最少的服务器
18.   end if
19.   if Server 为空 then
20.     return null
21.   else
22.     return Server 中的任意一个服务器
23.     end if
24. end function
```

如算法 2 所示,第二个模块用于实现为 Reduce 阶段的子任务选择最合适的槽位。与 Map 阶段的子任务不同,Reduce 阶段子任务的数据输入往往来自不同的服务器,此时追求输入数据的本地化没有意义。在调度此类子任务的过程中,nats 将首先考虑没有任何 Reduce 阶段子任务运行的服务器;如果此类服务器不存在,则尝试寻找能够使当前被调度任务获得最高网络优先级的服务器。这样做的意义在于,如果不采取上述策略,那么当前子任务被调度后将会因网络优先级较低而处于阻塞状态,无法取得明显的执行进展,被该任务占用的槽位即被浪费。如果没有服务器能够满足上述要求,则 nats 将暂时放弃对当前子任务的调度,并将槽位留空。如果后续到达的子任务可以满足调度要求,则将其调度至被留空的槽位,从而消除队头阻塞。

值得注意的是,在 Server 列表非空,即第算法 2 第 15 行条件不成立的情况下,我们不再根据其他条件(如服务器剩余槽位的数目等)对候选服务器进行进一步区分,而是随机返回其中一个。原因在于对其进行进一步区分并不能明确获益:若将网络任务调度至剩余槽位较多的服务器,会导致下一波计算任务到达时,剩余槽位较少的服务器承担所有槽位都被计算任务占用的风险,即对应服务器上的网络资源无法被利用;相反,若将网络任务调度至剩余槽位较少的服务器,那么对于接下来到达的任务,其所有子任务将会被调度至少数几台剩余槽位较多的服务器(因为剩余槽位少的服务器优先被网络任务占满),子任务过于集中将导致分布式执行的优势丧失。

5.3 槽位不足模式

算法 3 总结了槽位不足模式下的子任务调度。该算法基于小任务优先(SJF)策略,只有当前活跃任务中最小任务的子任务可以被调度。这样做的原因在于:1)当槽位不足时,系统负载已经足够高,没有必要考虑其他任务;2)考虑其他任务将违背小任务优先策略,不利于缩短任务的平均完成时间。

算法 3 槽位不足模式(以 Hadoop 为例)

输入:L 为任务列表

输出:Task,Server 分别为选中的子任务与服务器

```
1. L0←L 中优先级最高的的任务
2. if L0 中含有就绪的子任务 then
3.   if L0处于 Map 阶段 then
4.       return SelectTaskServerPair(L0)
5.     else
6. Task←L0 中的最小就绪子任务
7.       Server←SelectServerForTask(Task)
8.       if Server 非空 then
9.       return Task,Server
10.      end if
11.    end if
12. end if
13. retur nnull
```

槽位不足模式下的调度分为两种情况。如果最小任务正处于 Map 阶段,那么 nats 将多次调用基本模块 SelectTaskServerPair,从而选出尽可能多的"子任务-服务器"对并实施调度;如果最小任务正处于 Reduce 阶段,那么 nats 将从最小的子任务出发,依次调用基本模块 SelectServerForTask 为其选择槽位。正如部分已有工作所采取的调度策略,我们注意到最大的子任务常常是一个阶段中最慢的子任务,其完成时间直接决定着任务的完成时间,应当优先考虑;然而,这一策略在槽位不足时并不适用。在槽位不足的场景下,对于一个任务而言,其子任务的并行程度往往严重受限,并呈现出逐个执行的倾向。此时,子任务的调度顺序对任务完成时间的影响不大,而优先调度小任务则使得槽位资源在拥塞期更可能实现快速释放。

5.4 槽位充足模式

算法 4 总结了槽位充足模式下的子任务调度算法。与槽位不足模式下的调度算法不同,在槽位充足时,nats 可以调度并不属于当前最小任务的子任务;但为了满足小任务优先(SJF)策略,nats 依然会优先调度小任务。当被调度的子任务属于最小任务时,算法 4 与算法 3 在结构上是类似的,除了 nats 在调度 Reduce 阶段的子任务时,会从较大的而不是较小的子任务开始尝试。这是由于较大的子任务通常需要消耗更多的时间来接收数据与完成计算,更容易成为当前阶段的执行瓶颈,因此 nats 需要优先为较大的子任务选择合适的槽位并提供合理的网络资源。

算法 4 槽位充足模式(以 Hadoop 为例)

输入:L 为任务列表,Congestion 表示网络是否拥塞

输出:Task,Server 分别为选中的子任务与服务器

```
1. L0←L 中优先级最高的的任务
2. if L0 中含有就绪的子任务 then
3.   if L0处于 Map 阶段 then
4.       return SelectTaskServerPair(L0)
5.     else
6.       Task←L0 中的最大就绪子任务
7.      Server←SelectServerForTask(Task)
8.      if Server 非空 then
9.        return Task,Server
10.       end if
11.   end if
12. end if
13. if Congestion then
```

```
14.     for i=1→len(L)do
15.       if Li 处于 Map 阶段且含有就绪子任务 then
16.         return SelectTaskServerPair(Li)
17.         end if
18.       end for
19. else
20.     for i=1→len(L)do
21.     if Li 处于 Reduce 阶段且含有就绪子任务 then
22.       Task←Li 中的最大就绪子任务
23.       Server←SelectServerForTask(Task)
24.       if Server 非空 then
25.       return Task,Server
26.         end if
27.       end if
28.     end for
29. end if
30. return null
```

当被调度的子任务并不属于最小任务时，nats 将根据网络的拥塞情况执行不同的调度策略。当网络不拥塞时，nats 优先调度网络资源需求量大的子任务，如 Reduce 阶段的子任务。由于此时网络并不拥塞，调度此类子任务有很大的概率能够充分利用网络剩余带宽，从而避免资源的浪费。当网络处于拥塞状态时，nats 仅调度能够实现输入数据本地化的子任务，如大部分 Map 阶段的子任务，因为此类子任务对网络资源的需求量小，不会被网络阻塞，也不会影响正在运行的其他子任务。

6 实验评估

6.1 实验方法

我们首先通过小规模物理集群对 Duopoly 的效果进行评估，随后利用事件驱动的流级仿真器评估 Duopoly 在不同场景下的执行效果。通过对比二者仿真结果来确认仿真器的准确性，实验结果表明二者间的差距是可接受的。

我们的小规模物理集群包含 10 个服务器，其中 1 个作为主节点，其余 9 个用于实际执行子任务，各服务器通过 Mellanox 交换机以 1Gbps 链路相连接。服务器使用 DELL PowerEdge R730，每台服务器包含 2 个 12 核 Intel Xeon E5-2650V4 CPU，64 GB 内存，以及 1TB 机械硬盘。服务器操作系统使用 Ubuntu Server 14.04，其 Linux 内核版本为 4.4.0；拥塞控制协议使用 DCTCP，以保障更低的端到端时延。我们在交换机上配置了严格优先级排队以及逐队列 ECN 打标，并根据数据包 DSCP 取值将其映射至交换机内的 8 个物理队列之一。

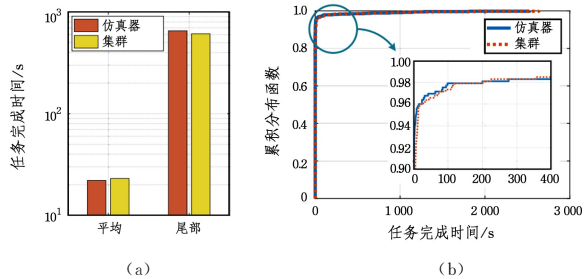
我们利用 Facebook 公开的 Coflow 基准测试数据生成用于效果评估的任务负载。具体而言，我们保持了基准测试数据中的原始流量大小和原始任务达到时间，任务负载累计包含超过 30 TB 的流量。通过对数据的观察，我们发现，该负载在任务数量、网络流量等多个维度具有突发提交的特性。需要注意的是，基于 Coflow 基准测试数据，我们只能得到 Reduce 阶段子任务的输入数据量。因此，我们基于对工业界集群的观测现象[16,20]，根据不用类型应用各阶段的输入输出比，生成了诸如 Map 阶段子任务大小等其他维度的数据，使得 Map 阶段与 Reduce 阶段的执行时间大致相等。

我们选择平均任务完成时间作为主要的评估指标，同时选择任务完成时间的第 95，99，100 百分位作为辅助评估指标。在接下来的实验中，我们首先将 Duopoly 的每个组件替换为平凡方法，并与 Duopoly 的完整实现做对比，以验证其各个组件的有效性；然后在不同场景下，将 Duopoly 与已有工作进行对比，包括感知网络的子任务调度系统 NEAT 与基于 Coflow 的网络调度系统 Aalo。NEAT 包含多种实现方式，这里我们选择带有串行启发式算法的 CCT-TCF 版本。此外，我们假设任务大小在任务到达时即可被估计，并且允许 NEAT 和 Aalo 使用任意多优先级。

实验结果总结如下：1）流级仿真误差是可接受的。我们将小规模物理集群的实验结果与流级仿真器的仿真结果进行对比，实验结果表明二者具有相同的趋势，且误差小于 7%。2）Duopoly 的各个组件都是有意义的。在将各组件逐一替换为平凡方法后，平均完成时间缩短了 7.5%～20%。3）Duopoly 能够在多种场景下缩短任务的平均完成时间。我们在多种场景下对 Duopoly 进行了评估，实验结果表明 Duopoly 几乎在所有涉及的场景下均表现良好。与 Aalo 和 NEAT 相比，Duopoly 将平均任务完成时间分别缩短了 37.65%～70.30%和 37.30%～76.16%。4）Duopoly 不会对其他指标产生明显的负面影响。虽然 Duopoly 的主要目标是缩短平均任务完成时间，但我们观察到任务的尾延迟至少与其他方法接近，甚至更优。与 Aalo 和 NEAT 相比，Duopoly 将任务完成时间的第 99 百分位分别降低了 45.81%和 49.89%。

6.2 主要实验结果

我们首先通过小规模物理集群对 Duopoly 的效果进行评估，并与相同配置下的仿真结果进行对比。为了与集群规模相适应，在本实验中我们将测试数据的流量缩小至十分之一。如图 5(a)所示，就平均任务完成时间与尾延迟而言，集群测试与仿真实验间的误差分别为 4.25%与 6.58%。其误差主要来源于如下两个方面：1）流级仿真忽略了传输层协议的细节，如 TCP 慢启动阶段造成的低速率（见图 5(b)放大区域）以及 ACK 数据包带来的带宽占用等；2）流级仿真忽略了优先级调度时抢占的开销，如 TCP 数据流被抢占时发生的重传与降速等。图 5(b)为集群测试与仿真实验中各任务完成时间的累积分布函数，通过观察可以发现，二者对应的曲线非常接近，从而进一步证明了仿真的准确性。



(a) (b)

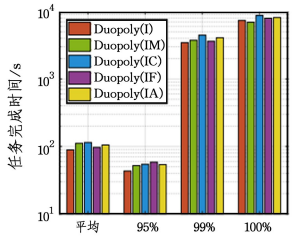
图 5 集群实验结果

Fig. 5 Tested results of cluster

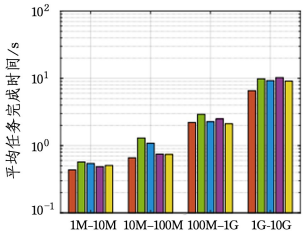
在仿真实验中，我们将网络抽象为一个大交换机，通过 1Gbps 的链路连接 50 台服务器，每台服务器包含 4 个槽位。对于 Map 阶段而言，每个子任务的输入数据包含 3 个备份，随机存储于 3 个不同的服务器中，同时假设每个槽位每秒可以完成对 1GB 数据的计算。

为了证明 Duopoly 各组件设计的有效性，我们将每个组件先后替换为平凡方法，得到 Duopoly 的弱化版，并与 Duopoly 的完整实现进行对比，以观察二者调度效果之间的差距。为了避免网络优先级数目对实验的干扰，我们在本实验中不对优先级数目进行限制，即允许调度器使用无限多的网络优先级。我们在实验结果图中使用后缀(I)表示允许无限多网络优先级的使用，同时使用后缀(M)与(C)分别表示网络调度器 cans 仅基于 Macroflow 与 Coflow 二者之一进行调度的策略，即网络调度不感知计算资源。类似地，我们使用后缀(F)表示子任务调度器 nats 随机将子任务分配至空闲槽位的策略，即子任务调度不感知网络资源。Duopoly 通过判断当前能否完成对最小活跃任务中全部就绪子任务的调度，来判断是否应当在槽位充足与槽位不足两种工作模式间切换。这里，我们使用后缀(A)来表示另一种策略，即以当前全部就绪子任务能否被调度作为模式切换的依据。

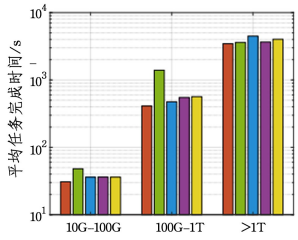
实验结果如图 6 所示，带有两个后缀的实验数据即为 Duopoly 的各个弱化版。实验结果表明，即使仅将一个组件替换为平凡方法，平均任务完成时间即可增加 1.08～1.25 倍，并在大多数情况下伴随着尾延迟的增加，这说明 Duopoly 各组件的设计都是有效的。为了进一步分析调度细节，我们按照任务大小将任务分为 7 组，并分别统计组内的平均任务完成时间，其结果如图 6(b)与图 6(c)所示。



(a)调度效果概览



(b)小任务调度效果



(c)大任务调度效果

图 6 主要结果

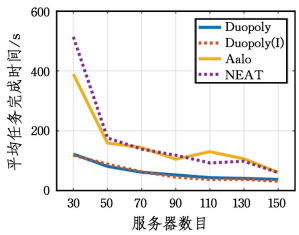
Fig. 6 Main results

通过观察发现，对 Duopoly 任何组件的替换均可导致所有分组的平均完成时间延长。接下来，我们将重点关注(IC)与(IM)两组数据。除了最后一个任务分组外，(IC)的调度效果总是优于(IM)。这是因为，无论槽位充足与否，基于 Macroflow 的网络调度策略总会使不同任务的子任务交织在一起，而不是把同一任务下的所有子任务视为一个整体进行调度，这与小任务优先(SJF)的调度策略相矛盾，从而导致小任务的完成时间显著延长。虽然这一调度策略存在明显的副作用，但其并非没有价值。当计算资源紧张时，基于 Macroflow 的网络调度策略能够有效缓解槽位紧张的状况，恰当使用可以使平均任务完成时间进一步缩短，而这一点正是 Duopoly 实行调度模式切换的依据。

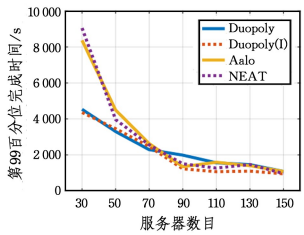
6.3 各场景下的实验结果

我们利用大规模仿真对 Duopoly 在各场景下的调度性能进行了评估，并将已有工作 NEAT 与 Aalo 进行了比较，二者分别为近期具有代表性的子任务与网络调度系统。为了公平比较，二者在任务调度期间均使用小任务优先(SJF)策略作为分配优先级的依据。在本节实验中，我们限制调度器只允许使用至多 8 个网络优先级；作为对比，我们额外对使用无限多网络优先级的 Duopoly 版本进行了评估。本实验覆盖多类场景，如等集群规模、槽位数目、网络负载等。接下来我们将逐一介绍各场景下的评估结果。

图 7 给出了集群规模对调度结果的影响。我们在不同的集群规模范围内对 Duopoly 进行了评估，涉及 30～150 台服务器。当集群规模较小时，基于 Coflow 的调度算法性能受到严重影响，这一现象是由槽位瓶颈所致的任务阻塞引起的。随着集群规模的增大，可用的网络资源与计算资源变得充足，这使得各算法的调度效果均有所提升，且不同算法间的差距逐渐减小。相较于 Aalo 与 NEAT，Duopoly 可以分别将平均任务完成时间缩短至少 38.00%与 37.30%；同时，Duopoly 尾延迟整体优于对比工作，或至少与对比工作近似持平。



(a)

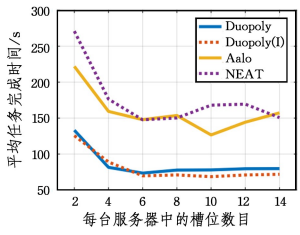


(b)

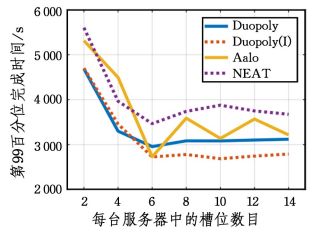
图 7 集群规模的影响

Fig. 7 Impact of cluster scale

图 8 给出了槽位数目对调度结果的影响。槽位数目影响着计算资源的充足程度，决定了 Duopoly 模式切换的时机。在固定服务器数目的前提下，我们让每台服务器内的槽位数目在 2～14 之间变化，并保持网络资源不变。



(a)



(b)

图 8 服务器槽位的影响

Fig. 8 Impact of slots per server

实验结果表明，所有算法的调度效果均随槽位数目的

增加而提升，这是计算资源在此过程中成为瓶颈的机会逐渐减小导致的。当每台服务器内的槽位数目超过 6 个时，Duopoly 的调度效果不再随计算资源的增多而提升，此时系统内的资源瓶颈由计算资源转移至网络资源，槽位总量的增加导致网络竞争加剧，从而使得平均任务完成时间有小幅增加。相较于 Aalo 与 NEAT，Duopoly 可以分别将平均任务完成时间缩短至少 38.40%与 47.05%，并分别将尾延迟减少至少 1.74%与 14.72%。

图 9 给出了网络负载对调度结果的影响。我们通过对基准测试数据中的流量大小进行缩放来实现对网络负载的调整，缩放倍率在 0.4～1.4 之间变化。实验结果表明，对于所有算法而言，平均任务完成时间与尾延迟均随负载的增大而增加，而 Duopoly 在所有负载下的表现均优于对比算法。相较于 Aalo 与 NEAT，Duopoly 可以分别将平均任务完成时间缩短至少 43.51%和 41.13%，并分别将尾延迟减少至少 18.21%与 10.84%。

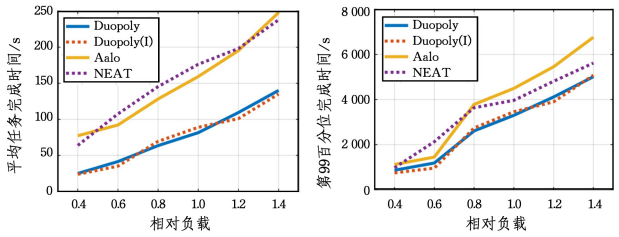


图 9 网络负载的影响

Fig. 9 Impact of traffic loads

图 10 给出了传输计算比对调度结果的影响。默认情况下，我们假设每个槽位每秒可以完成对 1GB 数据的计算；本实验中，我们让每个槽位的计算速率在 0.2～1.8GB/s 范围内变化，以评估传输计算比对调度结果的影响。实验结果表明，对比算法 Aalo 与 NEAT 在计算速率提高后，任务完成时间反而出现延长。这是因为计算速率的提高使得同一时间内执行实际计算的子任务数目减少，而进行网络传输的子任务数目增多，导致更多的并行数据流竞争网络资源，从而拖延小任务的完成时间；与此同时，大量数据流并行使得大量子任务处于低速传输或无法传输的空等状态，从而导致计算资源无法得到充分利用。这一实验揭示了计算网络联合调度的必要性，与单资源调度相比，考虑多资源的联合调度可以显著提升任务调度效果。相较于 Aalo 与 NEAT，Duopoly 可以分别将平均任务完成时间缩短至少 37.65%与 38.55%，并分别将尾延迟减少至少 20.54%与 15.61%。

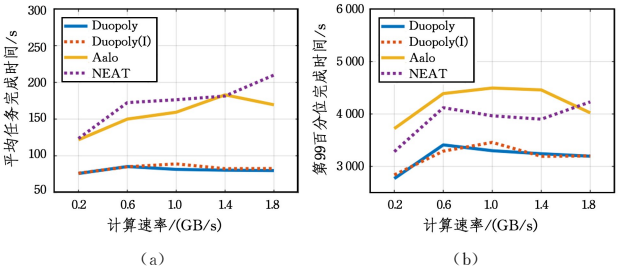


图 10 计算时间百分比的影响

Fig. 10 Impact of time percentage of computation

图 11 给出了网络拓扑对调度结果的影响。默认情况下，我们使用大交换机模型描述集群网络，此时网络拥塞点均在边缘服务器，而网络内部不存在拥塞点；本实验中，我们利用 2 个带有带宽收缩的分层拓扑 Topo1 与 Topo2，对各算法的调度结果进行评估。Topo1 包含 128 台服务器，均分于 8 个机架内；Topo2 包含 64 台服务器，均分于 4 个机架内。两个网络拓扑的带宽收缩比例均为 4:1，即网络核心层带宽仅为边缘带宽的 25%。实验结果表明，Duopoly 的调度性能几乎不受网络拓扑与带宽收缩的影响；同时，在带有内部拥塞点的拓扑中，增加网络优先级的数目可以获得更大的调度收益。

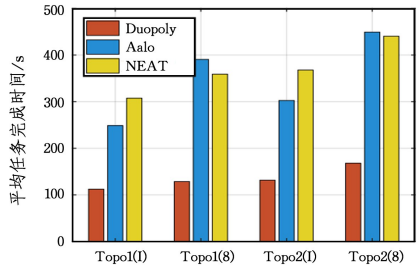


图 11 网络拓扑的影响

Fig. 11 Impact of network topology

图 12 给出了任务调度策略对调度结果的影响。默认情况下，我们使用基于小任务优先(SJF)的调度策略为任务分配优先级。本实验中，我们将放弃这一设计，使用公平调度的策略，即时刻保障各个活跃的任务中有相同数目的子任务处于运行状态。我们希望通过这一实验来进一步说明 Duopoly 的调度收益并不仅仅来自小任务优先(SJF)这一成熟的调度策略。实验结果表明，在使用公平调度策略的情况下，Duopoly 的平均任务完成时间仍然显著优于对比算法，而尾延迟与对比算法相似。

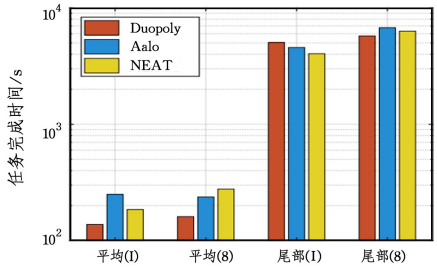


图 12 任务调度策略的影响

Fig. 12 Impact of job scheduling policy

7 相关工作

Duopoly 通过资源调度来缩短大数据系统中的平均任务完成时间，其相关工作可以分为网络调度与子任务调度两个方面。

已有的网络调度工作均以 Coflow 调度为基础，通过最优化 Coflow 完成时间来间接缩短任务完成时间。Orchestra 系统发现，通过简单的先进先出策略即可明显缩短平均 Coflow 完成时间[7]；Barrat 系统通过混合多个任务的传输过程来避免队头阻塞的出现[8]；Varys 系统利用最小瓶颈优先与最小总长度优先等策略实现集中式 Coflow 调度[9]；Aalo 则可以

视为 Varys 的分布式版本[10]。

已有的子任务调度工作大多以实现数据本地化为目的，以此减少需要通过网络传输的数据量，从而缩短任务完成时间。DelayScheduling 和 Quincy 系统均致力于实现 Hadoop 任务 Map 阶段的数据本地化[11-12]；而 Mantri 系统致力于提升 Reduce 阶段的数据本地化程度[13]；Corral 实现了输入数据与子任务的联合调度，从而获得更高的数据本地化程度[15]。Duopoly 的调度算法与上述工作是兼容的。

另一部分子任务调度工作则考虑到了网络负载的影响。DynMr 系统检测 Reduce 阶段子任务的数据接收过程，当发现其被网络阻塞时，利用 Map 阶段子任务填充所在的服务器，以避免资源浪费[17]。ShuffleWatcher 系统在发现网络拥塞时，将不再调度 Reduce 阶段子任务，而仅调度 Map 阶段子任务[14]。在选定网络调度方案的前提下，NEAT 系统可以根据网络调度方案来预估子任务耗费在网络传输上的时间，并以此作为调度依据[25]。与上述工作相比，Duopoly 的创新性在于其可以选择不调度任何子任务，而是将资源预留给未来可能到达的子任务，以避免队头阻塞。

目前已经出现计算-网络联合调度的工作[26-27]，这些工作通常仅支持离线调度，即需要在一开始就获得所有将会到达的任务信息。相比之下，Duopoly 是面向在线调度问题的调度系统，该系统无法预知接下来会有哪些任务到达。

结束语 在如何缩短数据并行框架中平均任务完成时间这一问题上，已有工作通常假设计算资源相对充足，从而通过降低网络流量或缓解网络拥塞的方法来解决问题。然而，在计算资源相对不足的场景下，上述方法很容易导致计算资源用尽，从而使等待中的任务因长期无法得到计算资源而陷入队头阻塞。为了解决这个问题，本文设计并实现了基于计算-网络联合调度的解决方案，其具有良好的稳定性与可扩展性。与已有的工作相比，本文最多可将平均任务完成时间缩短 76.16%。

参 考 文 献

[1] OUSTERHOUT K, WENDELL P, ZAHARIA M, et al. Sparrow: distributed, low latency scheduling [C]// Proceedings of the 24th Symposium on Operating Systems Principles (SOSP 2013). New York: ACM, 2013: 69-84.

[2] SHINNAR A, CUNNINGHAM D, SARASWAT V, et al. M3r: increased performance for in-memory hadoop jobs[J]. Proceedings of the VLDB Endowment, 2012, 5(12): 1736-1747.

[3] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C]// Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 2012). Berkeley: USENIX Association, 2012.

[4] TRIVEDI A, STUEDI P, PFEFFERLE J, et al. On the [ir] relevance of network performance for data processing [C]// Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud 2016). Berkeley: USENIX Association, 2016: 126-131.

[5] OUSTERHOUT K, CANEL C, RATNASAMY S, et al. Monotasks: Architecting for performance clarity in data analytics frameworks [C]// Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017). New York: ACM, 2017: 184-200.

[6] OUSTERHOUT K, RASTI R, RATNASAMY S, et al. Making sense of performance in data analytics frameworks [C]// Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI 2015). Berkeley: USENIX Association, 2015: 293-307.

[7] CHOWDHURY M, ZAHARIA M, MA J, et al. Managing data transfers in computer clusters with orchestra [C]// Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM 2011). New York: ACM, 2011: 98-109.

[8] DOGAR F R, KARAGIANNIS T, BALLANI H, et al. Decentralized task-aware scheduling for data center networks [J]. ACM SIGCOMM Computer Communication Review, 2014, 44(4): 431-442.

[9] CHOWDHURY M, ZHONG Y, STOICA I. Efficient coflow scheduling with varys[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(4): 443-454.

[10] CHOWDHURY M, STOICA I. Efficient coflow scheduling without prior knowledge[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4): 393-406.

[11] ZAHARIA M, BORTHAKUR D, SARMA J S, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling [C]// Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010). New York: ACM, 2010: 265-278.

[12] ISARD M, PRABHAKARAN V, CURREY J, et al. Quincy: fair scheduling for distributed computing clusters [C]// Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009). New York: ACM, 2009: 261-276.

[13] ANANTHANARAYANAN G, KANDULA S, GREENBERG A G, et al. Reining in the outliers in map-reduce clusters using mantri [C]// Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI 2010). Berkeley: USENIX Association, 2010: 265-278.

[14] AHMAD F, CHAKRADHAR S T, RAGHUNATHAN A, et al. Shuffle-aware scheduling in multi-tenant map-reduce clusters [C]// Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 2014). Berkeley: USENIX Association, 2014: 1-12.

[15] JALAPARTI V, BODIK P, MENACHE I, et al. Network-aware scheduling for data-parallel jobs: Plan when you can[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4): 407-420.

[16] CHEN Y, ALSPAUGH S, KATZ R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads[J]. Proceedings of the VLDB Endowment, 2012, 5(12): 1802-1813.

[17] TAN J, CHIN A, HU Z Z. Dynmr: Dynamic mapreduce with reduce task interleaving and maptask backfilling [C]// Procee-

dings of the Ninth European Conference on Computer Systems (EuroSys 2014). New York: ACM, 2014: 1-14.

[18] KAVULYA S, TAN J, GANDHI R, et al. An analysis of traces from a production mapreduce cluster [C]// Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID 2010). Washington: IEEE, 2010: 94-103.

[19] CHOWDHURY M, KANDULA S, STOICA I. Leveraging endpoint flexibility in data-intensive clusters[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 231-242.

[20] CHEN Y, GANAPATHI A, GRIFFITH R, et al. The case for evaluating mapreduce performance using workload suites [C]// Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2011). Washington: IEEE, 2011: 390-399.

[21] COSTA P, DONNELLY A, ROWSTRON A, et al. Camdoop: Exploiting in-network aggregation for big data applications [C]// Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 2012). Berkeley: USENIX Association, 2012.

[22] BAI W, CHEN L, CHEN K, et al. Information-agnostic flow scheduling for commodity data centers[C]// Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI 2015). Berkeley: USENIX Association, 2015: 455-468.

[23] PENG Y, CHEN K, WANG G, et al. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing [C]// The 33rd Annual IEEE International Conference on Computer Communications (INFOCOM 2014). Piscataway: IEEE, 2014: 19-27.

[24] WANG H, CHEN L, CHEN K, et al. Flowprophet: Generic and accurate traffic prediction for data-parallel cluster computing [C]// International Conference on Distributed Computing Systems (ICDCS 2015). Piscataway: IEEE, 2015: 349-358.

[25] MUNIR A, HE T, RAGHAVENDRA R, et al. Network scheduling aware task placement in datacenters [C]// Proceedings of the 12th International on Conference on Emerging Networking Experiments and Technologies (CoNEXT 2016). New York: ACM, 2016: 221-235.

[26] CHANG H, KODIALAM M, KOMPELLA R R, et al. Scheduling in mapreduce-like systems for fast completion time [C]// The 30th IEEE International Conference on Computer Communications (INFOCOM 2011). Piscataway: IEEE, 2011: 3074-3082.

[27] CHEN F, KODIALAM M, LAKSHMAN T. Joint scheduling of processing and shuffle phases in mapreduce systems [C]// The 31th IEEE International Conference on Computer Communications (INFOCOM 2012). Piscataway: IEEE, 2012: 1143-1151.

TIAN Bing-chuan, born in 1993, Ph. D. His main research interests include network verification, programmable networks and distributed systems.

TIAN Chen, born in 1978, Ph. D, associate professor, Ph. D supervisor, is a member of China Computer Federation. His main research interests include data center networks, distributed systems, internet streaming and urban computing.

(责任编辑:李亚辉)