基于Storm的任务调度:现状与研究展望

张 洲[1] 黄国锐[2] 金培权[1]

(中国科学技术大学计算机科学与技术学院 合肥 230001)[1] (中国人民解放军31002部队 北京 100081)[2]

摘 要 以Apache Storm为代表的分布式流式数据处理系统能够在复杂大数据处理环境中提供低延迟的处理,因此受到了学术界和工业界的普遍关注。在分布式流式数据处理系统中,任务调度是决定系统性能的关键因素。一个优秀的任务调度器能够为系统带来更高的吞吐量、更低的处理延迟和更好的资源利用率。Storm原生的任务调度器需要用户手动设置并行度,并且使用简单的轮询方法进行任务分配,在实际应用中性能较差。针对这一问题,研究者提出了多种面向Storm任务调度机制的优化策略。文中综述了Storm任务调度机制的相关工作,首先介绍了Storm系统以及原生的任务调度机制,并梳理了目前提出的面向Storm任务调度机制的优化技术,总结了各种方法的优点和缺点;最后讨论了Storm任务调度优化在未来的若干发展方向,以期能够为Storm任务调度机制的进一步优化和应用提供参考。

关键词 Apache Storm,流式处理,任务调度,在线调度,任务并行度

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.09.004

Task Scheduling on Storm:Current Situations and Research Prospects

ZHANG Zhou[1] HUANG Guo-rui[2] JIN Pei-quan[1]

(School of Computer Science and Technology,University of Science and Technology of China,Hefei 230001,China)[1]

(PLA 31002,Beijing 100081,China)[2]

Abstract Distributed streaming data processing systems represented by Apache Storm provide low latency processing in complex big data processing environment. Therefore,systems have attracted wide attentions in both academic field and industrial field. In the distributed streaming data processing system,task scheduling is a critical factor to determine system performance. A good task scheduler can result in higher throughput,lower processing latency,and better resource utilization for the system. However,the original Storm task scheduler requires users to set the parallelism manually,and it also uses simple round-robin method to assign tasks,which leads to poor performance in practical application. To handle this problem,researchers have proposed many optimization strategies of Storm task scheduling mechanism. This paper reviewed related works of Storm task scheduling. Firstly,the Storm system and the original task scheduling mechanism were introduced,and current optimization techniques on Storm task scheduling mechanism were sorted. Then the advantages and disadvantages of scheduling strategies were summarized and analyzed. Finally,some future development directions of Storm task scheduling optimization were discussed in order to provide references for further optimization and follow-up researches on Storm scheduling mechanism.

Keywords Apache storm,Stream processing,Task scheduling,Real time scheduling,Task parallelism

1 引言

随着云计算、社交网络等技术的快速发展,人类社会进入了大数据时代。传统的大数据处理采用的是离线的批式处理方式,其中最具代表性的平台是Hadoop[1],它一次性处理一批数据,并且先存储后计算,已经难以满足社交网络、军事等领域中相关应用的实时性需求。流式处理是一种实时的处理模式,Storm[2]是较有代表性的流式处理平台。Storm在数据到达时立即进行处理,适用于对实时性要求较高的场景。除了这种模式,还有一种流式处理模式——微批处理,代表性的平台是Spark Streaming[3-4]。它会在固定的时间窗中收集数据并处理,其延迟比Storm的延迟大一个数量级[5]。

Storm在应用中需要面对大规模的复杂流式计算。它将复杂的计算流程以Topology的形式表达,将计算流程分割成许多简单的计算组件,并且通过增加每个计算组件的线程数量来提高计算的并行度。这就导致了任务调度问题,即如何确定合适的并行度,如何将隶属于不同Topology和计算组件的任务分配给集群中的机器。衡量一个实时处理系统的性能

到稿日期:2018-07-05 返修日期:2018-09-15 本文受国家自然科学基金面上项目(61672479)资助。

张 洲(1995-),男,硕士生,主要研究方向为分布式系统、数据库技术;黄国锐(1974-),男,博士,主要研究方向为大数据与人工智能;金培权(1975-),男,博士,副教授,CCF高级会员,主要研究方向为数据库、大数据,E-mail:jpq@ustc.edu.cn(通信作者)。

是否优秀的主要因素是处理延迟和吞吐量，此外还有能效和可靠性等次要因素。对于 Storm 来说，任务调度决定了系统的处理延迟、吞吐量和能效，是影响系统性能的关键机制。

然而，Storm 默认的调度策略采用了简单的轮询方式，没有考虑能效、任务间的通信、负载均衡等问题，而且缺少在线的调度机制，无法有效应对变化的数据流速和实时插拔的多 Topology 任务。自 2013 年以来，大量的研究对 Storm 调度机制进行了优化，论文数量逐年上升。但是，目前对 Storm 调度优化问题进行总结和梳理的文献还很少。蔡宇等[6]总结了 Storm 调度优化方法，将这些优化分成了 4 类。但是，该研究只总结了 2016 年以前的优化方法，针对 Storm 调度优化的研究在 2017 年呈现井喷式增长；并且，该研究缺少对 Storm 并行度配置优化的总结，也未对离线调度和在线调度加以区分。

本文围绕 Storm 及其任务调度问题，概述了目前基于 Storm 的任务调度研究的现状，总结了目前在 Storm 并行度配置、任务分配策略等方面的研究进展，并讨论了各类策略存在的主要问题；最后给出了基于 Storm 的任务调度在未来的若干发展方向，以期为面向 Storm 任务调度的研究提供有价值的参考。

本文第 2 节介绍 Storm 的基本概念以及 Storm 原生的任务调度策略；第 3 节概述并分析了 Storm 任务调度策略的优化技术；第 4 节对 Storm 任务调度优化的未来研究进行了展望；最后总结全文。

2　Storm 的任务调度策略

2.1　Storm 的基本概念与相关术语

2.1.1　Storm 任务的逻辑抽象

Storm 将实时应用程序的逻辑打包成一个 Storm Topology（以下简称 Topology），如图 1 所示。

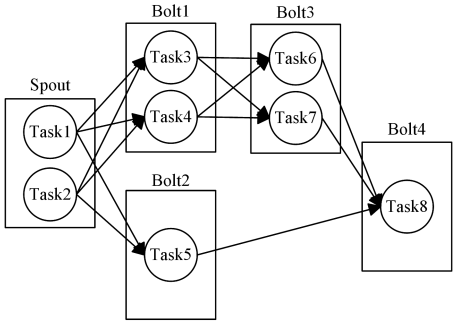

图 1　Storm topology

Fig. 1　Storm topology

Topology 是 Storm 对复杂计算任务的抽象，一个 Topology 就是一个有向无环图（Directed Acyclic Graph，DAG）。Topology 的每个顶点代表一个组件，组件分为两种：Spout 和 Bolt。Spout 负责从数据源读取数据，并向 Topology 发送 Tuple，Tuple 是 Storm 处理数据的单位。Bolt 封装处理逻辑，实现对数据的具体处理，每次处理一个 Tuple。

2.1.2　Storm 的物理架构

Storm 一般工作在集群上，采用主从式架构，如图 2 所示。

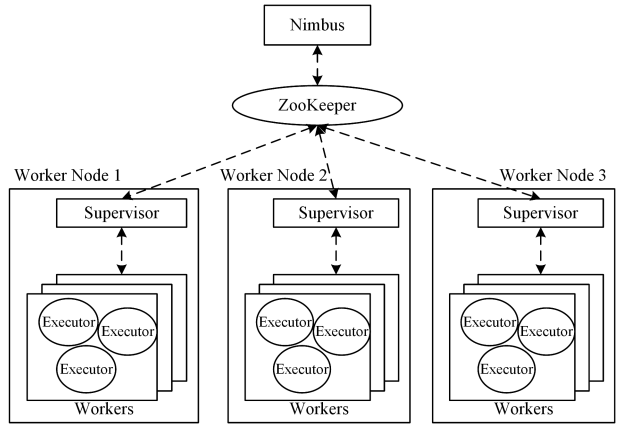

图 2　Storm 架构

Fig. 2　Storm architecture

Storm 的主节点上运行着一个叫作 Nimbus 的进程，它是整个系统的管理者，负责分发任务和监控系统的运行状态等。一个 Storm 系统中只能有一个 Nimbus 进程。Storm 中的其他节点都被视为从节点，从节点又叫作 Worker node，每个 Worker node 上都会运行一个叫作 Supervisor 的进程，它负责监听 Nimbus 分发给它的任务，并管理节点上的工作进程 Worker。Nimbus 与 Supervisor 之间的通信是由 Apache ZooKeeper[7]完成的，它们将状态都存储在 ZooKeeper 上。当某个进程因出错而关闭时，只需要重启出错进程就可以继续工作，不会影响整个 Storm 系统的运行。

Storm 在从节点上可以分为 4 个层次，Storm 术语与实际意义的对照如图 3 所示。

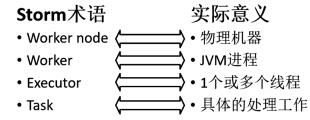

图 3　术语与实际意义的对照

Fig. 3　Glossary of terms and practical meanings

上文提到的 Worker node 指的是物理机器，每个 Worker node 上可以承担一定量的工作，这体现为每个 Worker node 上的 Worker 数目是有上限的。Storm 引入 Slot 来实现计算资源的分配，一个 Worker node 的 Slot 数目就是它能承担的 Worker 数目，每个 Slot 根据配置拥有一定的资源，具体为 CPU 资源和内存资源。一个 Worker 实际上是一个 Java 虚拟机（Java Virtual Machine，JVM）进程，一个 Worker 上可以运行多个 Executor。一个 Executor 是一个或多个线程，上面可以运行多个 Task。Task 指的是具体的处理工作。下一节将解释这样分层的意义。

2.1.3　Storm 的并行度

Storm 的并行度通过 Worker，Executor 和 Task 3 种实体来实现。下面通过一个例子来解释如何使用 Storm 以及 Storm 中的并行度是如何具体实现的。

图 4 给出了一个非常简单的 Topology，它包含 1 个 Spout 和 2 个 Bolt。程序员首先用代码描述 Topology，并指定每个组件的 Task 数目，这里指定 Spout，Bolt1 和 Bolt2 的 Task 数目分别为 2，4，6。然后将代码提交给 Nimbus，如图 5 所示。

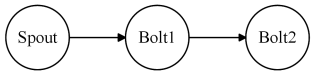

图 4 一个简单的 Topology

Fig. 4 Simple Topology

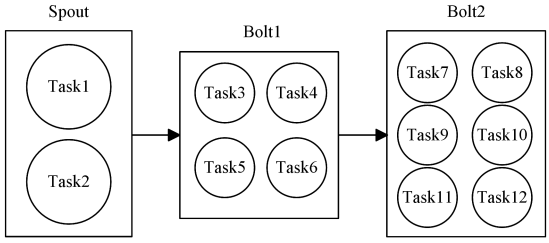

图 5 Topology 中 Task 的数目

Fig. 5 Number of tasks in topology

用户需要指定每个组件的并行度。需要注意的是,并行度指的是 Executor 的数量,而不是 Task 的数目。

如图 6 所示,一个直角方形代表一个 Worker,一个圆角方形代表一个 Executor,一个圆形代表一个 Task。这里指定 Spout,Bolt1 和 Bolt2 的并行度分别为 2,2,6。于是,Spout 和 Bolt2 的每个 Executor 运行 1 个 Task,Bolt1 的每个 Executor 运行 2 个 Task,并且 Executor 会分配给不同的 Worker。在实际使用中,Executor 的数目与 Task 的数目大多数时候是相等的。进行这样区分的原因是 Task 数目在 Topology 运行过程中是不能改变的,而 Executor 数目和 Worker 数目可以通过修改并行度参数来改变,这个过程被称为 rebalancing。

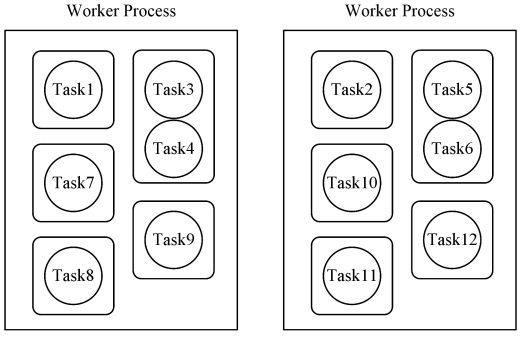

图 6 任务分配的结果

Fig. 6 Result of task scheduling

Storm 的并行度分为 Topology 并行度和组件并行度。上文已有相关描述,组件并行度通过 Executor 实现,Topology 并行度则通过 Worker 实现。每个 Executor 都隶属于某一个组件,每个 Worker 都隶属于某一个 Topology。因此,规定:一个 Executor 上只允许运行属于同一个组件的 Task,一个 Worker 上只允许运行属于同一个 Topology 的 Executor。

当 Topology 被提交之后,Nimbus 会根据并行度设置和调度策略将工作逐层分发下去。与批式处理系统不同的是,在 Storm 中,除非用户手动停止 Topology,否则 Topology 会一直运行下去。

2.2 Storm 原生的任务调度策略

在许多研究中,Storm 的调度过程被分为以下 3 个步骤:并行度设置,将 Executor 分配给 Worker,将 Worker 分配给 Worker node。大多数研究专注于其中的一个或两个环节。下面首先介绍 Storm 原生的任务调度策略。

Storm 的当前版本 V1.2.1 拥有 4 种调度器,分别是默认调度器、隔离调度器、多用户调度器和资源感知调度器。

无论是哪一种调度器,在第一个环节中,Storm 都将并行度设置完全交由用户自己完成,用户凭感觉或者经验设置的值可能是不合理的,这会影响整个系统的性能。

默认调度器的第二和第三个环节是合在一起的。Storm 的默认调度器采用简单的轮询机制,当有 Executor 需要被分配时,它将 Worker node 按照可用 Slot 的数量从多至少排序,然后依次从每个 Worker node 取出第一个可用 Slot,并将 Executor 按顺序分配给它们。如果一轮之后还有未分配的 Executor,再从每个 Worker node 中取出下一个可用 Slot,以此类推,直到所有 Executor 都被分配。由于 Worker 与 Slot 是一对一关系,因此这里的可用 Slot 就是可用 Worker。这种调度策略十分简单,而且开销很低,但它忽略了许多影响系统性能的因素,例如进程间通信、节点间通信、能效等。

Storm 的隔离调度器允许用户指定 Topology 运行在专用机器上,并且具有较高的优先级,使得这些 Topology 的性能能够得到保证。多用户调度器考虑了多用户的应用环境。资源感知调度器源于文献[8],将在下一节详细叙述。

Storm 目前只支持离线调度,没有在线的调度器。当有新的 Topology 提交,或者有 Topology 被停止,或者数据流输入速度显著变化时,原来的调度方案可能不再适用,需要一种在线调度策略来应对这种情况。

针对 Storm 的默认调度器存在的性能问题,学术界和产业界都进行了大量的研究,并且提出了许多新的思路,实现了各种各样的调度算法,其中有些已被采纳为 Storm 的可选调度策略。

3 Storm 任务调度的优化技术

目前,大量研究致力于优化任务分配策略,也就是将 Executor 分配给 Worker 和将 Worker 分配给 Worker node 的过程。在给出离线任务分配策略的同时,许多研究还给出了在线任务分配策略。此外,还有一些研究致力于找到最优的并行度配置。

3.1 离线调度优化策略

Storm 默认的任务调度策略在分配 Executor 时采用简单的轮询方法,这种方法忽略了许多影响系统性能的因素。大部分研究的关注点是如何降低通信代价。此外,一些研究优化了资源的分配,提高了资源利用率,还有一些研究关注容错、错误恢复等其他方面。

3.1.1 降低通信代价

Storm 默认的任务调度策略采用简单的轮询方法,这种方法会将 Executor 随机分配给 Worker 和 Worker node,未考虑节点间通信、进程间通信和进程内通信的代价差异。目前,Storm 调度优化最主流的方法是降低通信的代价,将互相通信的 Executor 更多地放在同一个 Worker 中或者同一台机器上。这是因为节点间通信的代价大于进程间通信的代价,进程间通信的代价大于进程内通信的代价。通信代价具体指通信消耗的资源和通信带来的延迟,这会直接影响系统的性能。

Aniello 等[9]首次提出基于 Topology 结构的调度策略,目标是降低通信代价。该算法比较简单,首先分析 Topology

结构，将存在互相通信的组件标记出来，并且将组件按照数据流动的方向排序。在调度时，按顺序取出组件的 Executor 进行分配，优先将 Executor 分配给与它通信频繁的 Worker。把 Worker 分配给 Worker node 的过程仍采用轮询方法。该策略的有效性在文献[10]中受到质疑，因为该算法忽略了工作负载。

Fischer 等[11]提出了一种基于图划分的调度策略，该策略基于文献[12]的工作。它使用 METIS 图划分软件[13]对通信图进行划分，目标是找到子图之间通信代价最小的局部最优解，从而创建调度方案。Eskandari 等[14]在此基础上进行了改进，对通信图进行两次划分，划分结果更加优秀。但是，文献[11]是基于假定的通信代价实现的；文献[14]会事先收集日志数据，统计通信代价，再执行调度算法。它们都没有实现在线通信负载监测模块，无法根据负载的实时变化动态生成调度策略；而且需要使用额外的软件运行图算法，调度的计算代价较高。

Jiang 等[15]进一步研究了基于图划分的任务调度策略，并给出了详细的调度算法。首先，对节点的计算能力和 Tuple 的计算代价进行建模；然后，将 Topology 线性化，并使用启发式算法对结果进行图划分，再按照划分结果进行调度。该方法的计算代价小于使用额外软件的方法[11,14]。

熊安萍等[16]提出的方法与基于图划分的方法类似，但是简化了调度流程。该策略首先采集周期内 Executor 之间的 Tuple 传输速率并求出期望，将速率高于某一阈值的边称为热边，其对应一对传输速率较高的 Executor，调度算法将这样的 Executor 对作为一个整体进行调度。该方法与前文所述方法的区别在于：它使用 Tuple 传输速率作为通信度量，并且只考虑热边，省略了许多调度过程中的计算代价。

上文介绍的是以降低通信代价为目标的离线任务调度策略，还有许多文献提出了基于通信的在线任务调度策略，详情请见 3.2 节。

3.1.2　优化资源分配

除了考虑通信代价之外，许多研究致力于提升资源利用率。

Peng 等[8]提出了一种基于资源的任务调度策略，即 R-Storm。它考虑了 3 种资源，即 CPU 资源、内存资源以及带宽资源；并认为内存资源是一种硬约束，另外两种资源是软约束。算法从 Spout 开始广度优先遍历组件，以此对 Task 进行排序；将 Topology 中的第一个 Task 分配给资源最多的机柜中资源最多的 Node；对于之后的 Task 分配，使用该 Node 到其他 Node 的网络距离作为带宽资源，并与 CPU 资源、内存资源一起映射到三维空间中。该算法选择欧几里得距离最短的 Node 进行分配，同时保证满足硬约束。欧几里得距离在 3 个维度上的投影分别为：Task 所需 CPU 资源与 Node 剩余 CPU 资源的差、Task 所需内存资源与 Node 剩余内存资源的差、该 Node 到第一个 Node 的网络距离。该调度策略同时考虑了节点资源的利用率以及通信代价，可以节约节点并提高性能，但是需要对 Task 的资源需求与 Node 的剩余资源总量进行度量。该策略很大程度地提高了系统的性能，被 Storm 采纳为一种可选的任务调度策略，并在最新版本中得以实现。

与该方法类似的是基于 QoS 的任务调度策略[17]，该策略适用于大型分布式环境。它建立了一个四维成本空间，其中两个维度是延迟属性，构成了一个延迟子空间；另外两个维度是节点的可用计算能力和节点利用率。通过在成本空间中放置任务，来得到调度结果。该算法是分布式的，与之相似的还有文献[18-19]中的方法。

Sun 等[20]提出了一种基于关键路径的任务调度策略。该方法首先在 Topology 中分别按照计算代价和通信代价给顶点和边赋予权值，然后找到一条总权值最大的路径，称其为关键路径，并优先分配关键路径上的顶点给能使得关键路径最快完成的 Node。本文的问题是 Storm 可以通过增大并行度来减轻 Executor 的计算负载，因此关键路径本身是不存在的，作者忽略了 Storm 的不同组件可以拥有不同并行度的问题。

以资源分配为目标的任务调度策略直接将任务分配给节点，忽略了进程层面的优化。这一部分算法旨在提高资源利用率，通信代价也会作为次要因素考虑。这些算法都是离线的，不考虑数据输入速率的波动性。当数据输入速率显著提高时，节点可能会过载。

3.1.3　基于其他目标的调度优化

文献[21]基于文献[20]的方法找到关键路径，然后基于关键路径进行基于容错的任务调度。该方法把关键路径上的任务优先分配给错误恢复速度快的节点，以此提高系统整体的错误恢复能力。为了降低某一节点失效对最终结果的准确程度的影响，文献[22]提出将相关联的任务放在同一节点上。Li 等[23]提出了一种贪心算法和一种启发式算法，用于在给定的容错配置下进行任务调度，以在恢复过程中减小性能下降程度的同时减少资源占用。

Chen 等[24]将 Storm 扩展到 GPU。目前，Storm 还没有导入 CPU 以外的设备作为计算资源，它对 GPU 的计算能力进行了量化，并设计了 Executor 调度算法。

文献[25-26]针对资源不足以计算所有任务的情况，提出了基于优先级的任务调度策略。该策略优先保证高优先级任务的执行，但是没有对系统的性能进行提升。Storm 原生的隔离调度器也可以在一定程度上解决这一问题。

3.2　在线调度优化策略

3.2.1　基于通信的在线任务调度

Aniello 等[9]在基于 Topology 结构的离线任务调度策略的基础上，提出了基于通信的在线任务调度策略。它通过监控模块实时监控 Executor 之间的 Tuple 传输速率，然后按照通信速率对 Executor 对进行降序排列，再依次判定 Executor 对是否被分在同一个 Worker 中，如果没有，则将它们迁移到最低负载的 Worker 中。这种方法忽略了 Topology 结构，只考虑了 Executor 之间的通信代价，而且实验测试使用的是他们自己的应用程序，不是业界通用的测试数据，算法的有效性存在疑问。

Chatzistergiou 等[27]提出的方法与文献[9]相比有一些改进，它将 Task 对按照通信代价降序排列，将 Node 按处理能力降序排列，并依次将 Task 对分配给 Node，如果目标 Node 的剩余计算资源无法满足 Task 的需求，则依次选择队列中靠后的 Node 。文献[27]使用 Task 和 Node 等术语，原因是该研究没有针对某一特定的流式处理系统，但实验测试是基于 Storm 系统实现的；同时，还利用 Topology 结构来改进算

法，将属于同一 Topology 的 Task 对尽量分配在一起。该方法需要对通信代价和处理能力进行量化与度量。与文献[9]相比，文献[27]中的方法考虑了 Node 的处理能力，也考虑了 Topology 结构。Zhang 等[28]也采用类似的方法将 Executor 分配给 Worker，但是在把 Worker 分配给 Worker node 时还考虑了负载均衡。

Xu 等[10]提出了一种基于通信的在线任务调度策略，即 T-Storm。它实现了负载监测模块，周期性地收集计算负载和通信负载并将其存于数据库中；任务生成模块周期性地读负载信息并计算出一个调度；调度器周期性地抓取当前调度生成模块生成的调度，并通过 Nimbus 将其分发下去。T-Storm 的架构如图 7 所示，这种架构被大多数在线调度策略所采用。T-Storm 的调度生成模块会将 Executor 按照通信代价降序排列，然后按顺序分配，优先将属于同一 Topology 的 Executor 分配给同一个 Slot，并保证分配满足其他两条限制，分别用于防止节点过载、细粒度控制节点合并。T-Storm 希望通过减少节点数量来降低通信代价，但是在分配过程中没有考虑节点间的通信代价。

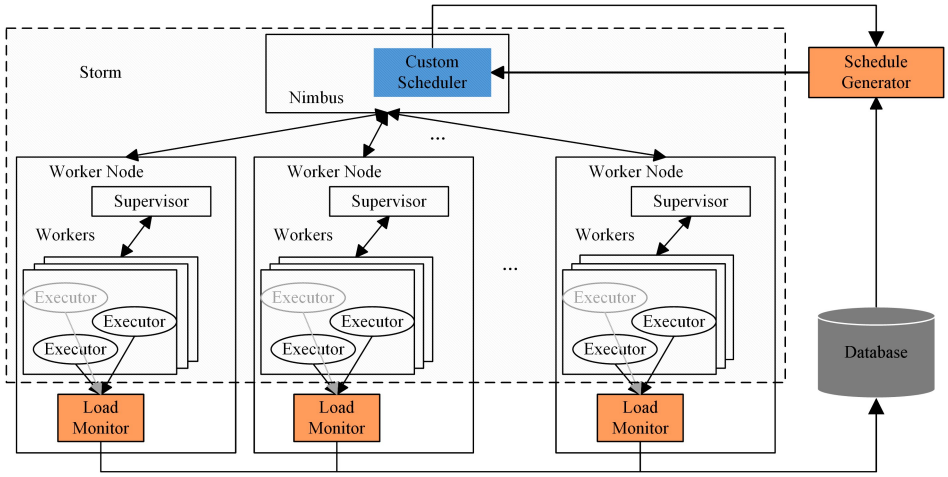

图 7 T-Storm 架构

Fig. 7 T-Storm architecture

3.2.2 其他在线调度策略

Jiang 等[15]设置了一个参数，叫做节点倾斜度，其代表节点负载与所有节点平均负载的差异。当节点倾斜度超过一定阈值时，触发调度操作，这种调度策略是为了实现动态的负载均衡。

文献[20-21]使用的都是基于关键路径的任务调度策略，它们都实现了在线调度机制。这类调度策略会实时监测当前的关键路径，当数据流到达速率发生变化时，响应时间也可能改变。如果当前调度不再拥有最小响应时间，则会触发新的调度操作，将关键路径上的关键顶点调度到新的计算节点上。

这些在线任务调度策略都能够适应变化的负载，但是它们都只关注了调度策略本身，没有关注如何实现运行中 Topology 的任务迁移；同时，它们在改变调度时，需要先关闭运行的 Topology，才能执行新的调度，这带来了较长的等待延迟。

3.3 负载均衡与节能

在 Storm 调度过程中，是使用尽量多的节点，还是使用尽量少的节点，许多研究持有不同观点。

Xu 等[10]提出了一种细粒度的节点合并策略，该策略在保证节点不过载的情况下减少节点的使用量。该研究认为使用较少的节点可以减少节点间的通信，从而获得更好的性能。

Sun 等[20]实现了基于能耗的任务调度策略，通过合并非关键路径的非关键顶点来降低整体的 CPU 使用率，从而提高能效。文献[20]与文献[10]虽然是从两个角度入手，但都是通过合并计算代价低的任务来达到减少节点数量的目的。

与之相反，文献[15-16,28-30]都提出了负载均衡策略，它们都认为负载不能集中在某几个节点上，而应该平均分配到所有节点上。

Qian 等[30]改进了 Storm 的默认任务调度机制，提出了 S-Storm。S-Storm 总是优先向剩余 Slot 最多的 Worker node 中的 Slot 分配任务，这一策略使得所有 Worker node 上的剩余 Slot 数目尽量平均。文献[16,28-29]都是将 Worker node 按负载大小升序排列，优先将 Worker 分配给负载最低的 Worker node。

文献[15]已经通过节点倾斜度触发调度操作，实现了动态负载均衡。

对于流式处理来说，低延迟、高吞吐量是系统的首要目标，同时还应考虑大数据处理过程中复杂多变的情形。将互相通信的任务合并到同一节点可以降低通信延迟，但是负载较重的节点应对数据流波动的能力较差，容易引起节点过载。因此，具体的应用环境(例如，数据流速是静态的还是动态的、系统总体负载大小等)决定了哪一种策略更有效。此外，对节点负载设定一个危险阈值也是一种解决办法。

3.4 任务并行度配置优化策略

除了任务分配外，一些研究通过优化任务并行度配置来提升系统性能。Storm 没有并行度配置机制，需要用户手动配置并行度，这对于用户来说是一道难题。

Sax 等[31]通过监测 Executor 的数据到达速率和数据流出速率，动态调节并行度的大小，即组件的 Executor 数量。Fu 等[32]使用相似的方法动态调整并行度。Cardellini 等[33]通过度量 CPU 使用率来动态调整并行度。

Shieh 等[34]使用与文献[31]类似的方法改变 Worker 的数量，是一种更粗粒度的并行度配置。Li 等[35]基于滑动窗口收集信息，以决定是否改变 Worker 的数量。

Li 等[29]提出了一种基于约束理论（STDO-TOC）[36]的 Topology 动态优化算法，该理论用于消除流水线操作的性能瓶颈。首先计算 Bolt 的计算代价；然后分析 Bolt 消息队列的拥塞度，确定 Topology 的性能瓶颈；最后通过修改参数配置来消除瓶颈。该研究使用系统支持的 rebalancing 方法来改变并行度设置，并增加拥塞度高的消息队列的长度。

文献[37-39]引入机器学习技术来获得参数配置。Weng 等[37]实现了 AdaStorm，它抽取了集群特征、Task 特征、Topology 特征和数据特征，并进行了大量训练；然后使用事先给出的配置集合，加上抽取的特征，运行两个预测模型，使用多层感知机预测 CPU 和内存使用，使用 J48 算法预测吞吐量和延迟能否满足用户需求；最后从满足用户需求的配置子集中选取使用资源最少的子集。AdaStorm 会实时监控系统状态，如果当前数据的消费速率与之前的明显不同，或者数据生产速率超过消费速率达到一定程度，则重新计算最优配置。

Wang 等[38]提出了 OrientStream，与文献[37]中的模型相比，它使用了增量学习算法，并且从集群层、操作符层、查询计划层和数据层抽取特征，还提出了一种集成学习模型——EDKR 模型。与文献[37]使用单一的模型相比，集成学习方法预测的结果更精确，而且加入的增量学习算法 hoeffding tree 使得模型在运行过程中可以自我更新。在文献[38]的基础上，文献[39]使用了 Ding 等[40]提出的操作符状态迁移方法，大大缩短了配置更新时的等待时间。

3.5　各类任务调度策略的总结

Storm 调度的优化技术主要可分为：减小通信代价、提高资源利用率、均衡负载负载和动态配置并行度等。其中，减小通信代价、均衡负载和动态配置并行度的大多数研究都包含了在线的方法。

3.5.1　方法对比与总结

最早诞生的是基于 Topology 的离线调度算法和基于通信的在线调度算法[9]，近年来基于这一思路的研究有许多，方法越来越完善。Zhang 等[28]提出了 TS-Storm，并将其与 Storm 默认调度、基于 Topology 的调度[9]、基于通信代价的调度 T-Storm[10]进行了对比。该实验在线形 Topology、菱形 Topology 和星形 Topology 3 种 Topology 结构上进行，其结构如图 8 所示。TS-Storm 在 3 种 Topology 结构上的吞吐量分别比默认调度提升了 17%，47%和 52%。在吞吐量方面，基于 Topology 的调度高于默认调度器，而 T-Storm 高于基于 Topology 的调度，但低于 TS-Storm。

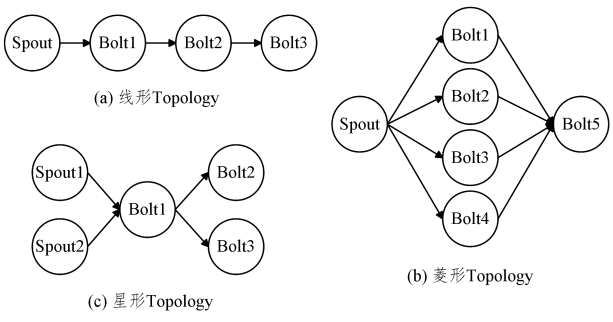

图 8　线形 Topology、菱形 Topology 和星形 Topology

Fig. 8　Linear Topology, diamond Topology and star Topology

基于资源的调度算法 R-Storm[8]在离线算法中表现优秀，该算法实现简单、性能优秀，因此在新版本 Storm 中已被采纳为一种可选的调度策略。同样在线形 Topology、菱形 Topology 和星形 Topology 上对该算法进行了实验，并对任务进行了区分。对于计算量小的任务，延迟主要体现在网络通信上，R-Storm 的吞吐量比默认调度器提升了 50%，30%，47%；对于计算密集型任务，R-Storm 提升了 CPU 利用率，在 3 种 Topology 结构上分别提升了 69%，91%，350%。Jiang 等[15]提出的 Bootstrap 调度器使用了基于图划分的方法，在真实的工作负载上，该方法比默认调度器提升了 39%的性能。

文献[28，30]开始考虑负载均衡问题，除了优化 Executor 的分配过程外，还对 Worker 分配的过程进行优化，并且在实验中加入了数据流波动因子。文献[37-39]将机器学习算法应用在了 Storm 任务调度策略上，它们的优化思路是在满足任务需求的前提下提高资源利用率，这与 R-Storm 的思想是相近的，但是它们的方法同时是一种在线调度策略，可以应对数据流波动。

3.5.2　已有方法存在的不足

基于 Topology 和基于通信代价的算法都减小了通信代价，这两种方法可以结合在一起使用[9，27-28]，但是这种方法在离线时表现不佳，而且需要在线地监控所有线程的负载，这带来了额外的代价。此外，更改调度方案时需要将整个 Topology 停止，然后重新启动进程和线程，每次重启都会带来秒级延迟[10，41]，如此巨大的代价不允许频繁地更改调度方案。

基于资源的任务调度是离线的调度策略，目前没有配套的在线调度算法。虽然在均匀的数据流速下，基于资源的任务调度算法有着优秀的性能，但是由于它注重提升资源利用率，在数据流速波动较大时存在节点过载的风险。

是负载均衡更重要，还是能效更重要，仍是一个值得研究的问题，需要找到一种既能适应负载波动又能降低通信延迟的折中方法。机器学习算法的引入解决了基于资源的任务调度策略难以应对数据流波动的问题，但是它使用的模型需要进行训练，需要抽取大量的特征，额外开销很大，性能不如 R-Storm。目前，这方面的研究尚处于起步阶段，有很大的提升空间。Storm 是一种实时处理系统，任务调度结果的优越性固然重要，但算法的计算代价和带来的延迟也不容忽视。

4　未来研究展望

上文对已有的 Storm 任务调度优化技术进行了梳理和总结，本节针对研究现状存在的空白，讨论了 Storm 调度优化在未来的研究方向。

4.1　面向异构处理器的调度

目前 Storm 只能工作于 CPU 上，异构处理器作为新兴的处理器，是 Storm 未来的扩展方向。目前，将 Storm 扩展到 GPU 的研究仍在起步阶段，在 GPU 集群或者混合集群中进行 Storm 任务调度将是未来的重要发展方向。

由于 GPU 是针对类型高度统一的、相互无依赖的大规模数据和不需要被打断的纯净的计算环境设计的，因此它的架构与 CPU 存在差异。基于 GPU 架构对 Storm 的工作模型

进行改进，从而更好地利用 GPU 的性能，是扩展的首要工作。对 GPU 计算能力的量化也不能采取简单的 Slot 方法，应该针对 CPU 和 GPU 计算能力的差异设计专门的量化方法。此外，在异构集群中分配任务，还要考虑任务的差异，将适合 CPU 的任务分配给 CPU，适合 GPU 的任务分配给 GPU，从而提升系统性能。

4.2 细粒度任务迁移

目前，Storm 在线调度策略在改变调度方案时，需要先停止运行中的 Topology 任务，再执行新的调度，这带来了较高的等待延迟。目前，针对 Storm 任务的细粒度实时迁移、实时扩展、实时合并的研究比较匮乏，这将是未来的一个主要研究方向。

一种思路是将 Storm 的 Task 层与 Executor 层合并，当并行度改变时，直接动态地修改组件的 Task 数目，这会带来分组策略的改变，上层的组件都需要变化，实现难度很大。另一种思路是保留目前的分层策略，对 Executor 数目进行动态修改，这种思路需要关心 Task 在 Executor 中的分配。对于任务的实时迁移，可以先开启一个任务的复制，并同步新任务和老任务，然后关闭原来的任务。实时任务分裂、任务合并也可以采用类似的思路。

4.3 机器学习的更多应用

机器学习算法虽然需要较高的训练代价，但是在执行预测时计算代价低、速度快、精确度高。目前本领域中机器学习算法的应用仍然很少，未来将会有更多种类的机器学习算法从更多的角度应用在 Storm 任务调度中，这会是未来的主要研究方向。

目前的方法只抽取了简单的特征，而且没有对特征进行筛选。研究不同的特征集对不同种类预测的影响，筛选出合适的特征，可以提高预测的准确性。目前预测的种类也很少，而且缺少直接对并行度大小的预测。预测在资源允许的前提下怎样的配置能够达到最高的吞吐量或者最低的处理延迟，能够为高优先级的 Topology 提供配置参考。在数据流速波动时，预测配置的改变方向和改变幅度，能够实现全自动的并行度配置。此外，机器学习算法的选择和集成也是一个重要的研究方向。

结束语 随着人们对大数据处理实时性的要求越来越高，Storm 受到了越来越多的关注。本文综述了 Storm 原生的任务调度策略，详细梳理了 Storm 调度优化的研究现状，在此基础上分析并总结了目前该研究领域的主流方法和存在的问题，并给出了 Storm 调度优化未来的发展方向，以期对此领域未来的研究提供有价值的参考。

参 考 文 献

[1] Apache Hadoop[EB/OL]. http://hadoop.apache.org/.

[2] Apache Storm[EB/OL]. http://storm.apache.org/.

[3] Apache Spark[EB/OL]. http://spark.apache.org/.

[4] ZAHARIA M, DAS T, LI H, et al. Discretized streams: Fault-tolerant streaming computation at scale[C]// Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 423-438.

[5] CHINTAPALLI S, PENG B J, POULOSKY P, et al. Benchmarking streaming computation engines: storm, flink and spark streaming[C]// 2016 IEEE International Conference on Parallel and Distributed Processing Symposium Workshops. IEEE, 2016: 1789-1792.

[6] CAI Y, ZHAO G F, GUO H. A review on the scheduling optimization of real-time stream processing system Storm [J]. Computer Application Research, 2018, 35(9): 1-9. (in Chinese)
蔡宇，赵国锋，郭航. 实时流处理系统 Storm 的调度优化综述[J]. 计算机应用研究，2018，35(9)：1-9.

[7] Apache ZooKeeper[EB/OL]. http://zookeeper.apache.org/.

[8] PENG B, HOSSEINI M, HONG Z, et al. R-storm: Resource-aware scheduling in storm[C]// Proceedings of the 16th Annual Middleware Conference. ACM, 2015: 149-161.

[9] ANIELLO L, BALDONI R, QUERZONI L. Adaptive online scheduling in storm[C]// Proceedings of the 7th ACM international conference on Distributed event-based systems. ACM, 2013: 207-218.

[10] XU J, CHEN Z, TANG J, et al. T-Storm: Traffic-Aware Online Scheduling in Storm[C]// IEEE International Conference on Distributed Computing Systems. IEEE, 2014: 535-544.

[11] FISCHER L, BERNSTEIN A. Workload scheduling in distributed stream processors using graph partitioning[C]// IEEE International Conference on Big Data. IEEE, 2015: 124-133.

[12] FISCHER L, SCHARRENBACH T, BERNSTEIN A. Scalable linked data stream processing via network-aware workload scheduling[C]// International Conference on Scalable Semantic Web Knowledge Base Systems. CEUR-WS. org, 2013: 81-96.

[13] KARYPIS G, KUMAR V. A fast and high quality multilevel scheme for partitioning irregular graphs[J]. SIAM Journal on scientific Computing, 1998, 20(1): 359-392.

[14] ESKANDARI L, HUANG Z, EYERS D. P-Scheduler: adaptive hierarchical scheduling in apache storm[C]// Proceedings of the Australasian Computer Science Week Multiconference. ACM, 2016: 26.

[15] JIANG J, ZHANG Z, CUI B, et al. StroMAX: Partitioning-Based Scheduler for Real-Time Stream Processing System[C]// International Conference on Database Systems for Advanced Applications. Springer, 2017: 269-288.

[16] XIONG A P, WANG X W, ZOU Y. Scheduling algorithm based on hot edge of Storm topological structure [J]. Computer Engineering, 2017, 43(1): 37-42. (in Chinese)
熊安萍，王贤稳，邹洋. 基于 Storm 拓扑结构热边的调度算法[J]. 计算机工程，2017，43(1)：37-42.

[17] CARDELLINI V, GRASSI V, PRESTI F L, et al. Distributed QoS-aware scheduling in Storm[C]// Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. ACM, 2015: 344-347.

[18] NARDELLI M. QoS-aware deployment of data streaming applications over distributed infrastructures[C]// International Convention on Information and Communication Technology, Electronics and Microelectronics. Croatian Society MIPRO, 2016: 736-741.

[19] FARAHABADY M R H, SAMANI H R D, WANG Y, et al. A

QoS-aware controller for Apache Storm[C]// IEEE, International Symposium on Network Computing and Applications. IEEE, 2016: 334-342.

[20] SUN D, ZHANG G, YANG S, et al. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments[J]. Information Sciences, 2015, 319: 92-112.

[21] SUN D, ZHANG G, WU C, et al. Building a fault tolerant framework with deadline guarantee in big data stream computing environments[J]. Journal of Computer and System Sciences, 2017, 89: 4-23.

[22] SU L, ZHOU Y. Tolerating correlated failures in Massively Parallel Stream Processing Engines[C]// IEEE International Conference on Data Engineering. IEEE, 2016: 517-528.

[23] LI H, WU J, JIANG Z, et al. Integrated recovery and task allocation for stream processing[C]// IEEE, International PERFORMANCE Computing and Communications Conference. IEEE Computer Society, 2017: 1-8.

[24] CHEN Y R, LEE C R. G-Storm: A GPU-Aware Storm Scheduler[C]// Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress. IEEE, 2016: 738-745.

[25] CHAKRABORTY R, MAJUMDAR S. A priority based resource scheduling technique for multitenant storm clusters[C]// International Symposium on PERFORMANCE Evaluation of Computer and Telecommunication Systems. IEEE, 2016: 1-6.

[26] BELLAVISTA P, CORRADI A, REALE A, et al. Priority-Based Resource Scheduling in Distributed Stream Processing Systems for Big Data Applications[C]// IEEE/ACM International Conference on Utility and Cloud Computing. IEEE, 2015: 363-370.

[27] CHATZISTERGIOU A, VIGLAS S D. Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters[C]// ACM International Conference on Conference on Information and Knowledge Management. ACM, 2014: 1579-1588.

[28] ZHANG J, LI C, ZHU L, et al. The Real-Time Scheduling Strategy Based on Traffic and Load Balancing in Storm[C]// IEEE International Conference on High PERFORMANCE Computing and Communications; IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems. IEEE, 2016: 372-379.

[29] LI C, ZHANG J, LUO Y. Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm[J]. Journal of Network and Computer Applications, 2017, 87: 100-115.

[30] QIAN W, SHEN Q, QIN J, et al. S-Storm: A Slot-Aware Scheduling Strategy for Even Scheduler in Storm[C]// IEEE International Conference on High PERFORMANCE Computing and Communications; IEEE, International Conference on Smart City; IEEE International Conference on Data Science and Systems. IEEE, 2017: 623-630.

[31] SAX M J, CASTELLANOS M, CHEN Q, et al. Aeolus: An optimizer for distributed intra-node-parallel streaming systems[C]// IEEE International Conference on Data Engineering. IEEE, 2013: 1280-1283.

[32] FU T Z J, DING J, MA R T B, et al. DRS: Dynamic Resource Scheduling for Real-Time Analytics over Fast Streams[C]// IEEE International Conference on Distributed Computing Systems. IEEE, 2015: 411-420.

[33] CARDELLINI V, NARDELLI M, LUZI D. Elastic stateful stream processing in storm[C]// International Conference on High PERFORMANCE Computing & Simulation. IEEE, 2016: 583-590.

[34] SHIEH C K, HUANG S W, SUN L D, et al. A topology-based scaling mechanism for Apache Storm[J]. International Journal of Network Management, 2017, 27(3): e1933.

[35] LI J, PU C, CHEN Y, et al. Enabling Elastic Stream Processing in Shared Clusters[C]// IEEE International Conference on Cloud Computing. IEEE, 2017: 108-115.

[36] RHEE S H, CHO N W, BAE H. Increasing the efficiency of business processes using a theory of constraints[J]. Information Systems Frontiers, 2010, 12(4): 443-455.

[37] WENG Z, GUO Q, WANG C, et al. AdaStorm: Resource Efficient Storm with Adaptive Configuration[C]// IEEE International Conference on Data Engineering. IEEE, 2017: 1363-1364.

[38] WANG C, MENG X, GUO Q, et al. Orientstream: A framework for dynamic resource allocation in distributed data stream management systems[C]// Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. ACM, 2016: 2281-2286.

[39] WANG C, MENG X, GUO Q, et al. Automating Characterization Deployment in Distributed Data Stream Management Systems[J]. IEEE Transactions on Knowledge and Data Engineering, 2017, 29(12): 2669-2681.

[40] DING J, FU T Z J, MA R T B, et al. Optimal Operator State Migration for Elastic Data Stream Processing[J]. HAL-INRIA, 2015, 22(3): 1-8.

[41] YANG M, MA R T B. Smooth task migration in apache storm [C]// Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015: 2067-2068.