实时流处理系统 Storm 的调度优化综述*

蔡 宇[1], 赵国锋[1,2†], 郭 航[1]

(1. 重庆邮电大学 电子信息与网络研究院, 重庆 400065; 2. 重庆市光通信与网络高校重点实验室, 重庆 400065)

摘 要: 随着大数据技术的发展,相对于 Hadoop 等传统的批处理系统,流式处理系统具有更好的实时性特点。在已有的流式处理系统中,Storm 系统具有良好的稳定性、高可扩展性以及高容错性等优点,使它在流式数据处理系统中脱颖而出。但是在任务调度方面,Storm 系统并没有做过多的考虑,默认采用相对简单的轮询调度法,导致系统在性能上存在瓶颈。近年来针对 Storm 系统的调度问题,研究提出了各种优化方案。从实时流处理系统 Storm 的调度优化出发,将这些优化方法分为四类,并详细阐述各类中具有一定代表性的方法,分析其优缺点以及适用的场景。最后,讨论了在日益发展的新环境下,Storm 系统的调度优化相关研究未来可能存在的方向。

关键词: 流式数据处理; Apache Storm; 性能优化; 调度

中图分类号: TP301.6 **文献标志码:** A **文章编号:** 1001-3695(2018)09-2567-07

doi:10.3969/j.issn.1001-3695.2018.09.002

Survey of real-time processing system Storm scheduling optimization

Cai Yu[1], Zhao Guofeng[1,2†], Guo Hang[1]

(1. *Institute of Electrical Information & Network Research, Chongqing University of Posts & Telecommunications, Chongqing* 400065, *China*; 2. *Chongqing Key Laboratory of Optical Communication & Network in Colleges & Universities, Chongqing* 400065, *China*)

Abstract: With the development of large data technology, compared with Hadoop and other traditional batch system, streaming processing system has better real-time characteristics. In the existing flow processing system, the Storm system has the advantages of good stability, high scalability and high fault tolerance, so that it can stand out in the flow data processing system. However, in the task scheduling, Storm system has not done too much consideration, the default using a relatively simple polling scheduling method, resulting in the performance bottleneck in the system. In recent years, this paper proposed a variety of optimization schemes have been for the scheduling problem of Storm system. Based on the scheduling optimization of real-time stream processing system Storm, this paper divided the method into four categories, and described the methods of representative representation in each category in detail, and analyzed their advantages and disadvantages. Finally, it discussed the possible future direction of Storm system scheduling optimization in the new environment.

Key words: streaming data processing; Apache Storm; performance optimization; scheduling

0 引言

传统的大数据处理平台如 Hadoop[1],其核心思想是 MapReduce[2],是一种离线的批处理计算框架,将算法抽象成 Map 和 Reduce 两个阶段分别处理。这种思想非常适合密集型数据集的计算[3]。由于它是批次计算,每次只能进行一次计算,而且需要先存储后计算,在某些需要实时处理的环境下[4~7]并不适用。由于目前的互联网发展迅速,数据的时效性变得非常重要,数据的流式特征也越来越突出,所以传统的批处理形式的大数据处理系统已经不能很好地满足当前的需求。因此,实时计算系统得以发展,各类流式处理平台应运而生[8,9]。

实时处理是指随着源数据不断地输入[10],一边输入一边处理与输出[11]。要求处理时间必须小于数据的间隔到达时间。文献[12]中提到,实时计算的需求产生了很多相应的系统,但是它们的系统构架、规则定义、数据模型定义等完全不同。最初的流式处理系统大多为集中式的架构,其典型的代表为 Stream、Aurora。在这些系统中,计算部署在一个单独的节点上,缺点是一旦这个节点的可用资源达到饱和,那么计算速度将会大大减低,导致输出结果不及时,实时性受到影响。因此,研究开始从集中式的数据流处理系统向分布式的数据流处理系统演变,如 Yahoo 公司开发的流式数据实时处理框架 S4、Linkedin 公司开发的流式处理系统 Smaza、基于 Spark 开发的 Spark Streaming 以及由 Twitter 公司开发的 Storm 系统。在目前各类分布式实时流式处理的系统中,Storm 凭借它的高可扩展性、高容错性、高可靠性、易用性[13]等受到业界广泛关注,它设计了一种面向流和计算的架构[14]。

在复杂流事件处理引擎当中,数据流事件必须被及时处理[15,16],才足以满足实时性[17]。所以系统性能对于一个实时流处理系统来说非常重要。衡量一个实时流处理系统性能是否优越,主要看处理时延、吞吐量和 QoS 三个方面[18~20],而要提高实时流处理系统的性能,高效的任务调度是一个关键。目前在 Storm 系统内部,任务调度采用简单的轮询算法,但是它并没有充分考虑到 Storm 内部任务通信机制、负载均衡与弹性机制、资源调度的问题。本文将对当前研究的主要调度优化方案进行总结与归纳。

收稿日期: 2017-06-19; **修回日期:** 2017-08-10 **基金项目:** 国家自然科学基金资助项目(61501075)

作者简介: 蔡宇(1992-),男,四川自贡人,硕士研究生,主要研究方向为流式数据处理;赵国锋(1972-),男(通信作者),陕西人,教授,博导,主要研究方向为互联网技术、网络测试与测量(645656548@qq.com);郭航(1991-),男,福建福州人,硕士研究生,主要研究方向为流式数据处理.

1 Storm 基本框架与调度方法

1.1 Storm 基本框架

Storm 系统运行在分布式集群上,需要被处理的数据流被定义为无边界有序的元组[21]。一个 Storm 应用被称为一个 topology。Storm 系统中可以运行多个 topology。Topology 的模型可以看做一个有向图,数据在这张有向图中流动,经过一个个节点处理,最终汇聚得到需要的结果。用户通过提交topology 到 Storm 系统中,实现并行计算。Storm 系统通过两个层级的抽象(逻辑抽象和物理抽象)来进行描述。

1.1.1 Storm 框架的物理抽象

Storm 系统由主节点和从节点组成,如图 1 所示。

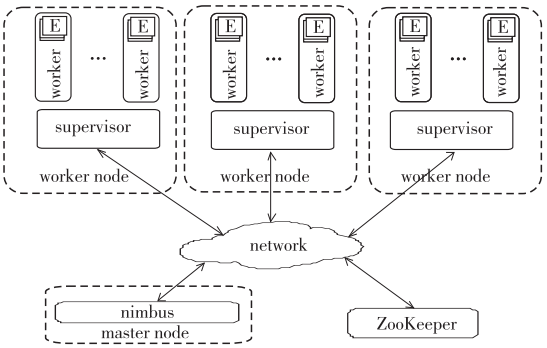

图1 Storm系统架构

主节点被称为 master node,也叫做控制节点,这个节点是用户与 Storm 系统交互的桥梁。在这个节点当中运行着一个叫 nimbus 的进程,nimbus 扮演着 Hadoop 里“JobTracker”相同的角色[22],其主要功能对计算任务进行分配调度和接收用户的命令,然后作出相应的处理。它是用户与 Storm 系统的连接点,用户通过 nimbus 将计算任务提供给 Storm 系统。

从节点被称为 worker node,也称为工作节点,在这个节点当中运行着一个叫 supervisor 进程。Supervisor 可以理解为单机任务调度器,它负责监听 nimbus 的任务调度,启动相应的 worker 进程对 nimbus 分配的任务进行处理。同时,它也会监测由它启动的 worker 进程的运行状态,一旦发现有 worker 进程处于非正常状态,就会 kill 掉这个 worker 进程,并将原来分配给该 worker 的任务交还给 nimbus,进行重新分配。Supervisor 进程会启动三个线程,如图 2 所示。

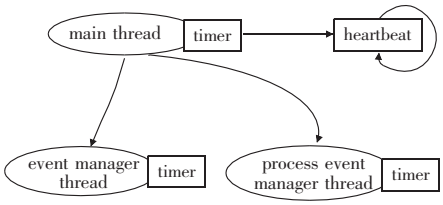

图 2 supervisor 线程图

主线程负责从 Storm 中读取配置,初始化 supervisor 的全局 map,在文件系统中创建一个持久化本地状态,以及调度可以循环的定时器事件。其中,有三种事件分别为 heartbeat 事件、supervisor 同步事件和进程同步事件。

在每一个从节点中,可以存在多个 worker 进程;而一个 worker 进程当中,又有多个线程,这些线程可以执行多个 task。每一个 worker 进程都有 worker 接收线程与 worker 发送线程两个线程,它们都基于 TCP/IP 协议。

所有的 nimbus 与 supervisor 之间的协调工作都是由 ZooKeeper[23]集群来完成。ZooKeeper 是一个分布式应用程序协调服务,可以提供配置维护、域名服务、分布式同步、组服务等功能[24]。ZooKeeper 集群保存着 nimbus 和 supervisor 的工作状态信息,所以,当 Storm 集群中某个节点发生异常后,重启这个节点不会造成数据的丢失。这种方式大大增强了 Storm 系统的安全性与稳定性。

1.1.2 Storm 框架的逻辑抽象

每一个 Storm 应用都可抽象为一个 topology,如图 3 所示。

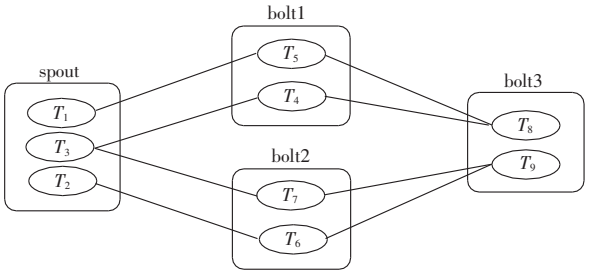

图 3 Storm 逻辑抽象图

其中,负责从数据源读取数据的组件被称为 spout,它是 topology 数据源的源头。通常 spout 会从外部的数据源获取数据(如 Kafka[25]等数据队列以及数据库等),然后将数据封装成 tuple 形式(Storm 支持的 tuple 值有多种,其中包括基本数值类型、字符串、字节数组等),并将其送入流当中。Bolt 则是数据流的处理单元,它会根据用户提供的逻辑来处理这些数据(如将数据持久化到磁盘、执行过滤、聚类等函数操作),处理结束之后会将结果传到出口或者传给下一个处理逻辑的 bolt,这样需要处理的数据就会从这一整张有向图中流过,得到最终的结果。Spout 与 bolt 之间的通信方式通过流来呈现,每一个 spout 或者 bolt 可以运行多个实例,即多个 task。这些实例的数量由用户编写 topology 时配置。

1.2 Topology 的并行度

并行度对于 Storm 系统的性能来说是一个非常重要的概念。只有合理部署 topology 的并行度,才能加快 topology 的执行速度。下面为几个与 topology 并行度有关的概念:

a) Task。

Task 是运行的实例,所有的数据处理,最终都是交给一个个 task 来计算。当用户在编写 topology 时,就需要将每一个 bolt 或者 spout 的 task 数量设计好,在以后的运行过程中,这个设置都不会发生变化。在 Storm 系统的默认情况下,一个 executor运行一个 task,所以在默认情况下,executor 的数量与 task 数量相同。

b) Executor。

Executor 从物理角度讲就是一个线程,它由工作进程产生,且运行在工作进程之内。根据 Storm 系统的规定,一个 executor进程只能为一个 bolt 或者 spout 服务,且 executor 可以运行多个 task。

c) Worker。

在 Storm 系统中,每一个 worker 进程都是属于某一个特定的 topology。从节点的 supervisor 进程可以管理一个或者多个 worker 进程。并且 Storm 会为节点上的每一个 worker 进程分配端口号,这个端口可以看做 slot。Worker 进程内部可以运行多个 executor。

根据上述概念,topology 的并行度是由 task、executor 以及 worker 的数量决定的。由于适应集群状态的 topology 并行度,才能使得计算更加迅速。所以,如何调度三者的数量,对于 Storm 系统的性能提升至关重要。

1.3　Storm 调度方法

当用户将 topology 提交到 Storm 集群以后,为每个 topology 分配资源是由 Storm 系统中的 scheduler 来负责,这个 scheduler 存在于主节点的 nimbus 当中。Storm 系统默认采取的 scheduler 是 evenScheduler,其算法的基本思想是轮询法,目的是在资源消耗尽量小的情况下能达到一种简便的负载均衡。整个调度过程分为两层:

a)定义好 executor 与 worker 进程之间的关系;

b)将 worker 分配到 slot。

在用户提交 topology 以后,主节点的 nimbus 进程会计算提交 topology 需求的工作量,即获取 task 数量,scheduler 会遍历整个 topology 对应的所有 executor,并以轮询的方式在配置好的 worker 内部创建好这些 executor。然后,scheduler 会从 ZooKeeper 上获取已有的分配任务,在从节点查找可用的 slot,以轮询的方式将 executor 均衡地分布到可用的 slot 上。分配好之后,supervisor 将根据从 nimbus 中获取的调度规则启动 worker 进程,准备执行计算工作。最后,worker 进程会与 ZooKeeper 交互,从 ZooKeeper 中获取到分配给当前 worker 的 task 信息。根据这些信息,再利用 task-id 找到对应的 spout 与 bolt,建立连接。整个调度过程就完成了。

但是这样的轮询调度机制为了追求简便、低开销,而忽略了很多对性能有影响的因素,造成性能不能达到最优。针对 Storm 系统的调度问题,学术界与产业界都提出了新的思路与算法,针对不同的方面,以实现 Storm 系统性能的优化。

2　Storm 集群调度的优化方法

由于实时处理系统最重要的属性是实时性,而通过优化 Storm 的调度算法,可以增加吞吐量,减小时延,以实现提高实时性的目的,所以目前学术界与产业界提出各类优化方法。本文将这些方法根据优化目标分为减小通信开销、负载均衡、优化资源分配以及提升系统弹性四种。下面将对这些优化调度算法进行归纳总结。

2.1　减小通信开销

通过改变 topology 组件的 task 实例的通信方式,尽可能使 task 的通信方式为进程间通信或者进程内通信,以求加快 task 之间数据的传输速率,从而提高整个系统的性能。这是目前 Storm 调度优化比较主流的做法。

当用户提交 topology 以后,topology 便运行于整个集群当中。Topology 的两个互相通信的组件(bolt 或 spout),它们的运行实例 task 之间的通信方式有三种:

a)不同节点之间通信。当相互通信的组件的运行实例 task 在不同的工作节点上,不同的节点通过网络相连,那么这两个 task 的通信只能通过网络来完成,但是通过网路进行通信的效率非常低,通信时延也相对较高。

b)同一节点上不同进程间的通信。当相互通信的组件的运行实例 task 处于同一节点但不同的 executor 中。这样,它们的通信方式将会变为进程间的通信,可以通过管道、消息队列、共享内存、套接字等方式进行通信,这些通信方式相比于节点之间的网络通信,效率更加高、时延也更低。

c)同一进程内通信。当相互通信的组件的运行实例 task 处于同一节点且相同的 executor 中。它们的通信机制为进程间的通信,这种通信方式最高效,通过指针的传递就可以完成。通信的效率非常高,通信时延也可以忽略不计。

由于 Storm 默认的调度算法为轮询算法,在调度过程中,基本没有考虑组件实例的通信开销,通信量大的两个实例可能分配到不同的节点之上,这样两个实例的通信方式只能依靠网络,造成性能的不足。如何将通信开销考虑进调度因素的范围内,是提高 Storm 性能的最重要环节。

针对上述问题,文献[26]提出了两种调度器,即静态调度器与动态调度器。静态调度器的工作模式是当 topology 提交以后,调度器会对 topology 的结构进行检测分析,然后针对 topology 的结构特点,尽可能地将相互通信的两个 task 实例放到同一个节点上。但是静态调度器所有的调度工作都在 topology 执行之前完成,一旦 topology 开始运行,静态调度器就无法进行调度。这个调度器有着简便、易实现的、低开销的优点,但是存在精准度不够、考虑因素不全、不能根据系统运行情况实时调节等缺点。

针对这些不足,文献进而提出了一个自适应动态调度器,它针对静态调度器的不足,通过监控模块,实时监控 executors 对之间传输的 tuples 速率,然后根据这个数值,对 executors 对进行降序排列。根据这个序列对的顺序,依次判定每个 executors 对中 executor 是否已经被分配。若都没有分配,则将它们分配到负载最低的 worker 上。如果这个 executors 对中有一个或者两个都已经被分配了,那么尽量把它们调整到最低负载的 worker 上面。通过贪心算法,得到一个相对优化的调度方案,进行实时调度。由于在线调度器只考虑了 executors 对的排序,而忽略了整个 topology 的通信模式,可能会造成某些调度的不准确,将一些相互通信的 executors 对分开。并且无论是静态调度器还是动态调度器都没有考虑 I/O 性能。其次调度器测试的 topology 不是业界通用的 topology,所以在这一点上是存在疑问的。

针对同样的问题,文献[27]提出了一个名为 T-Storm 的框架,以优化 Storm 默认调度器。图 4 为 T-Storm 架构。

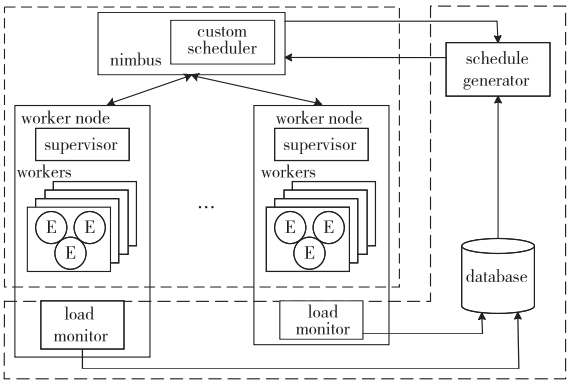

图4　T-Storm架构

与之前所述的动态调度器不同的是,T-Storm 对每一个节点只设置一个 slot,这样可以合理减少进程间的通信,使其变为进程内通信。在整个系统当中,核心是 traffic-aware online scheduling 算法,算法在对 executor 排序,并分配它们到 slot 的同时,设定了三个限定条件,保证了调度器的可行性和安全性。最后根据实测,系统在吞吐量和通信时延上的性能有所提高。此调度器的性能超过了默认的 Storm 调度器,但是它不能保证相互通信的一对组件在同一个节点上。因为在将 executor 分配给节点时,每个 executor 的总负载是被分开考虑的,并且它忽略了 executor 之间的通信速率。文献[28]同样针对通信开销过大这方面进行考虑。文献认为,在 topology 当中,互相通信的组件相互之间是有依赖关系的,所以第一步就是要针对这些依赖关系来对 topology 进行划分。使用的方法与之前的文献有些类似,也是通过排序得到一个序列,然后将这个序列中

的 executor 分配到对应的节点上。同时,文献在之后的过程考虑了负载均衡的问题。通过这一过程,达到系统性能的优化。但是在针对减小通信开销方法上,这篇文献与之前的方法大致相同,只是在算法细节上有所不同。

文献[29]总结了上述调度方法,在此基础上考虑了调度开销的问题。如果针对每一个 task 都进行调度,那么整个系统的开销将会非常高,导致性能不足。所以,作者提出将 task 之间的 tuple 传输速率作为判定条件,筛选出 tuple 传输数量大的两个 task,然后将这类 task 对尽可能地调度到同一个节点当中。如果不满足条件的 task 对,则不进行调度,维持原来的状态。然后作者提出热边算法,作为判断是否调度的条件。这样就实现了尽可能减小调度的开销。但是文章提出的判断算法只是使用的期望、方差等数学方法,这种判断模型过于简单,容易导致判断不准确。

文献[30]根据已有的思路,提出了新的想法,使用基于图分层算法来对任务进行调度。其思路是将整个 topology 看做一张有向图,记做 $G=(V,E)$。其中:V 代表 topology 上节点的集合;E 代表 topology 中边的集合。然后根据任务之间的通信量,将这个图分为 K 个子图集,记做 $P=\{P_1,\cdots,P_k\}$,之后就可以根据公式:

$$\text{cost}(G,P)=\sum_{i=1}^{V}\sum_{j=1}^{V}\text{differentPart}(i,j)\times ew_{ij} \tag{1}$$

得到整个 topology 的通信开销。其中:ew_{ij} 代表 i 到 j 节点的通信量;而函数 differentPart(i,j) 可以定义为

$$\text{differentPart}(i,j)=\begin{cases}1 & \text{part}(v_i)\neq\text{part}(v_j)\\ 0 & \text{otherwise}\end{cases} \tag{2}$$

通过最优化算法得到一个通信开销最小的局部最优解,然后根据这个决策去进行调度。文献[31]在此基础上进行了改进,将一次划分变为两次划分,这样提高了图划分的精准性,进一步提高了系统性能。但是这两种模型都需要用到一个额外的划分软件 METIS,这种通过第三方软件进行划分的方式很可能会加重系统开销。

由于 Storm 内部默认的调度器使用了简单的轮询算法,将整个 topology 的计算组件均匀地分配到整个工作集群当中,而不考虑节点与进程之间的通信流量问题,会对系统性能产生巨大的问题。这一类调度器通过获取一个 topology 内部相互通信组件之间的流量,然后根据各种数学模型,对其结构进行划分,尽可能地减小计算应用内部的通信开销。在对于 Storm 系统的优化方案中,这类思路已经被各界广泛使用,发展已经较为成熟。

通过减小通信开销以实现系统性能的优化有一定的效果,大多数方法基本都实现了一定的性能提升。不过这类方法总体来说思路比较单一,模式较为固定,大部分只是在数学模型上有些不同。

2.2 优化资源分配

并行计算当中资源的分布是非常重要的,只有合理的资源分配才能使系统计算速度提高,实现真正的实时计算。但是 Storm 默认的调度算法常常忽略资源的需求和可用性。因为这个原因,现在研究者也开始关注通过资源调度来实现 Storm 性能的提升。

文献[32]中提出了一个叫 R-Storm 的调度器,这个资源感知调度器通过最大化资源利用率来提高整体的吞吐量,同时减小网络延迟。其核心算法的关键点是要将任务根据系统的资源状况合理地分配给机器。该方案主要根据 CPU、内存以及带宽使用率三种不同的资源来进行调度,并且设定了硬件与软件两个约束。其中,硬件约束必须全部满足,软件约束则是尽可能去满足。算法将 CPU、内存、带宽抽象成一个三维空间,如图 5 所示,并将每一个 task 和工作节点根据资源状况映射到这个三维空间当中,最后利用 task 与工作节点的距离来调度。这种方法的优点在于通过多维度的资源参数来进行决策,很大程度上提高了决策的精准度;但是没有考虑内部通信的问题,且数学模型缺乏有力的证明。

文献[33]针对这个问题,提出了 QoS-aware scheduler。首先这个调度器破除了 Storm 的固定模式,将调度器分为两个层级。其核心的算法为 QoS-aware scheduling 算法,建立起一个布局的空间成本模型,通过将 CPU、内存等状态指标转换为空间成本,然后根据这个成本模型,对新的任务进行放置。这个思想源自于文献[34]。这样,根据成本进行 topology 组件的放置,有效地提高了资源的使用效率,从而实现性能的提升。与其方法相仿的还有文献[35,36]。

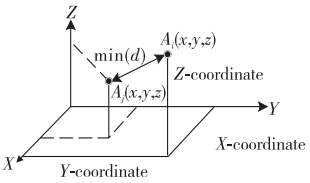

图5 R-Storm三维空间图

与放置策略相关的还有文献[37],提出了一个可扩展的集中式解决方案,它有效地使用基于组的作业来分配组件,并适应执行环境的变化。这一类调度器的性能很大程度上取决于放置策略。但是这些放置策略模型大多只针对某一个场景,换句话说使用了这些调度器的 Storm 系统可能在某一个场景的性能上表现非常优越,但是换一个场景就存在性能不足的情况。文献[38]在 Storm 平台之上提出了一种实时和节能的资源调度方案和优化框架,称为 Re-Stream。通过利用启发式和关键路径调度机制实现任务调度。并且用实验结果证明,Re-Stream 可以提高大型数据流计算系统的能源效率,并且减少了响应时间。但是它并没有提供资源缩放的功能。

在一些特殊场景当中,当资源已经不足以提供给所有的业务进行计算时,如何为优先级高的任务调度资源,是值得考虑的问题。这样一来,在某种程度上也算提高了系统性能。但是 Storm 默认的调度器并没有提供此类方法。所以文献[39]提出了这种思路,根据任务的优先级来进行资源调度。文献[40]提出了一个基于优先级的资源调度框架。这个调度框架包含调度器以及优先级管理系统两个组件。提交的 topology 需要实现优先级管理系统的接口。然后提交了包含优先级标志码的 topology 后,调度器会根据这些优先级标志码,对优先级高的 topology 优先分配资源并进行计算。这一类调度器实现功能较为单一,对于性能的提升并没有太大的作用。

文献[41]针对分布式集群的 GPU 资源,提出了一个名为 G-Storm 的调度器,实现支持 GPU 的 Storm 并行系统。调度器需要考虑 GPU 的容量,并且利用 GPU 的高速计算的特性,提高了系统的计算能力,实现了吞吐量的优化。文献[42]针对相同的思想,也提出了一个 G-Storm。同样是将 GPU 考虑为一个计算资源,不同的是这篇文献考虑更多的是如何实现将 GPU 作为一个 Storm 系统的计算资源,关于资源调度方面没有做过多的研究。

集群的计算资源是固定不变的,如何针对计算任务的特点进行合理的资源分配,对一个实时处理系统的性能是至关重要的。目前已有的这些调度方法从各个方面进行了考虑,其中最重要的一点就是根据资源情况,使用数学模型得到一个合理的

放置策略。好的放置策略对于系统的性能提升是非常有益的。但是现有的方法都是针对某一个场景而言,在其他场景就不适用了,而且大多数调度方法的模型缺乏有力的数学理论支持。

2.3　负载均衡

负载均衡是针对集群运行的实时状况来进行调度,使得计算任务能够更好地运行在集群当中。

针对大多数实时计算系统而言,目前都是采用的静态负载均衡技术,Storm 也不例外。当使用静态负载均衡技术部署后,实时计算任务就很难被改变,这样容易造成与实时计算系统的数据流具有的高度可变性发生冲突。导致的结果就是在某些时候,工作节点的数据流负载的波峰与波谷存在很大的差距。如果当前的数据流负载已经处于波峰值,但是部署的节点数量却少于实际上所需要的节点数量,这样就被称为 under-provisioning;反之,当数据流负载过低,部署的节点数量远多于所需要的节点数量时,这种现象称为 over-provisioning。无论是 under-provisioning 还是 over-provisioning,都会对当前的计算性能造成不可忽略的影响。所以实时计算系统需要动态负载均衡技术。然而 Storm 默认的负载均衡技术需要用户在创建 topology时手动指定并行度,然后通过这个并行度来计算实例个数,且 Storm 无法根据运行时的数据流的负载情况来进行动态调整,这样就会造成 Storm 系统的性能不足。所以针对这个问题,也有很多国内外学者提出了不同的解决办法。

文献[43]提出在 Storm 系统运行当中,如果发生 topology 运行失败或者集群中有新的节点加入,Storm 不能很好地应对这两种情况下的调度。文章通过 slot-aware 队列来实现一个细粒度的 evenScheduler;使用合并因子来避免在同一机器中的 spout 或者 blot 进行资源竞争。

图 6 为 S-Storm 系统结构。可以看到,其核心模块为一个 slot-aware scheduler 与合并因子 α。

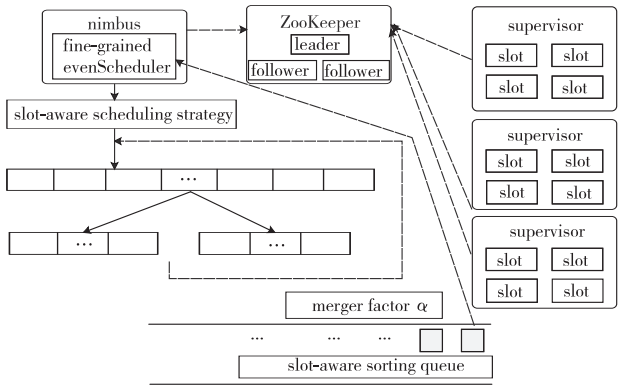

图6　S-Storm系统结构

首先细粒度调度器收集可用的 slots;然后对 supervisor-id 将这些 slots 分组,相同节点的 slots 分为一组;同时根据数量对这些组进行排序,可用 slot 数量越多的节点优先级越高;接着将第一组的第一个 slot 放入 slot-aware 队列,并重复进行之前的步骤,直到序列为空;最后细粒度调度器会根据计算现在计算任务所需要的 slot,并按照百分比从 slot-aware 队列中取出 slot,分配给计算任务。合并因子 α 代表一个 topology 在某一个节点的合并度,表示分配到一个节点的 topology 的 tasks 比例。由式(3)得到

$$a_{ij}=\frac{\text{worker}_{ij}}{\sum_{j=1}^{n}\text{worker}_{ij}}=\frac{\text{worker}_{ij}}{\text{worker}_{i}} \tag{3}$$

其中:worker_{ij} 代表 topology_i 在节点 j 的 worker 进程数量;而 worker_j 代表 topology_i 所需要的 worker 进程数量。通过实验证明,S-Storm 在吞吐量和平均处理时间方面取得了更优越的性能;同时,S-Storm 实现了将系统从负载不均衡状态恢复为均衡状态的功能。但是文献只考虑了工作节点之间的时隙分配,并没有考虑其他资源的问题,导致精准度存在一定的不足。

文献[44]针对负载均衡问题,提出了一个 TS-Storm 框架。其中,文献提出的 slots allocation algorithm 可以有效解决当前负载不均衡问题。其思路是根据 CPU 和内存的利用率计算工作节点的负载。当有新的 topology 提交时,工作节点将按照负载值升序排列,会将 topology 任务分配在负载较轻的工作节点。当节点的负载高于阈值 L,将会对负载过高的节点进行重分配,以保证节点的负载都在阈值之下。与这种做法类似的还有文献[28]。

由于 Storm 内部的 evenScheduler 忽略了工作节点上可用 slot 的数量以及 Storm 中 slot 的分配关系,所以上述文献针对这个问题提出了各种方案。通过现实运行系统的负载均衡,提升了系统性能。但是目前专门针对负载均衡的文献相对较少,方案不是很多,未来还需要研究更多针对负载均衡问题的优化方法。

2.4　提高系统弹性

由于需要实时处理的数据流具有多变性的特点,一旦将 topology 所分配的 worker 固定,当数据量突然变小时,可能造成不必要的浪费;反之,如果数据量变大,那么目前的 worker 数量可能满足不了需求[45],所以动态的调度 worker 数量可以提高系统的性能。这种方式称为提高系统弹性。弹性方法在系统框架下实现了动态水平缩放,即它允许自动调整 worker 的数量,根据每一个 topology 不同的运行状态实现资源的缩放策略。

文献[46]针对弹性方法问题,提出了一个 DBalancer 调度器。其算法思路是通过一个滑动窗口用于统计流数据信息,通过这些信息进行计算,得到是否添加和减少 worker 进程。由于添加 worker 进程和减少 worker 进程所消耗的资源不同,所以作者将两种方式分开。这样,使得运行在 Storm 中的所有 topology 的 worker 部署都达到最合适的状态,提高了系统性能。此调度器能很好地根据资源需要对 topology 进行一个动态的配置调整,使得 topology 的资源配置更好地适应当前的数据流状态。但是它没有考虑结束一个进程的状态迁移问题,这样会导致系统的不安全。此外,直接对 worker 进程进行配置操作太粗粒度化,调整幅度过大,可能导致性能的下降。类似的方法还有文献[47,48]。

文献[49]同样针对上述问题,进行了进一步的优化。作者在 nimbus 中新加入了 migrationNotigier 与 elasticityManager 两个模块。其中起到弹性机制的模块是 elasticityManager,它负责判定是否调整并行度,即添加或减少 topology 组件的 executor 数量,然后实施调度;而 migrationNotigier 的工作在调度发生之后,它通过通知需要调整的 executor 内部的 tasks,来保证内部状态不会被改变。进行调度判定的方法与上述文献类似,通过 CPU 使用率与设置的阈值进行比较。这样一来提供了动态迁移模块,保证系统处理数据的安全性,并且对 executor 线程而不是 worker 进程进行操作,使得调度器更加细粒度化,精准度得到提升。但是系统对弹性机制缩放策略只是简单依赖于 CPU 的上、下阈值进行考虑,这一点是显得过于简单。

所以,下述文献对弹性机制的缩放策略进行了更多的思考,如文献[50~53]也是根据当前测量的 CPU 利用率来制定弹性策略:文献[50]定义了所有节点之间的负载方差的上限阈值;而文献[51]则是为每个主机定义了上、下限;文献[52]

则是提出了一种机器学习的方法来得出阈值,这样其适应性得到很大的提升;文献[53]提出用一种模型来估计由一组组件工作产生的延迟尖峰,并使用它来定义延迟最小化的弹性机制策略,避免了手动调节阈值。

目前,Storm 系统并没有提供弹性机制,只支持静态或手动对 topology 资源进行配置。因此,需要将计算应用固定地部署在工作集群上,这样不能很好地满足流式计算特点。上述文献针对这个问题,提出了各类弹性机制用于支持配置的动态缩放,提高系统弹性,可以更合理地为运行中的 topology 配置 worker、executor 等,这样使得 topology 的计算配置更贴合当前的处理情况。

3 结束语

传统的批处理大数据需要将数据存储到本地,然后才能开始进行海量数据的处理。流式处理打破了传统批处理模式的制约,能够在不对数据进行存储的情况下进行数据的分析处理,大大提高了实时性。但是流式处理系统需要解决性能问题,高效的调度机制是一个关键。Storm 作为目前主流的流式处理系统具有独特的优势。由于所采用的调度算法过于简单,只能满足简单的负载均衡,导致系统在性能上不尽如人意。针对这个问题,研究者们提出了各种解决方案,但是在这一方面仍然存在很大的挑战。本文对流式处理系统 Storm 调度优化的发展方向进行了如下思考:

a)针对不同场景的优化。未来可以针对不同的场景,制定出一个符合于当前场景的调度方法。比如,设置一个历史任务状态存储器,以便于集群节点从故障中恢复,能够继续计算;当 topology 计算任务量不大时,将整个 topology 的任务放置到同一个节点进行计算,这样可以大大减小通信消耗。

b)更优秀的资源调度模型。在资源调度方面,目前数学模型最多只考虑了三个维度的资源,在未来可以加入更多维度的资源,用于建立数学模型,这样可以更好地使计算资源得以充分利用。

c)更加合理的调度算法。在算法方面,由于考虑到实时性,调度方案生成的速率需要尽可能地快,所以各类方法都使用的启发式算法得到局部最优解,但在速率与精准度的权衡问题上还值得更深入的考虑,未来获取可以通过更优秀的算法来平衡速率与精准度的问题。

d)弹性机制更多的思考。目前弹性机制的判定主要依靠 CPU 利用率这一单一资源,但是未来可以通过多维度的资源来进行判断,如内存、I/O、带宽等。这些资源的争夺都会对 Storm 性能造成很大的影响。如何综合考虑这个问题,对未来提升系统性能有很大的意义。此外,弹性机制必定会带来任务的迁移问题,如果不能很好地解决这个问题,弹性机制就会导致系统不安全。所以,如何更好地将弹性机制与任务迁移结合起来,也是未来值得研究的问题。

e)针对任务调度之后,目前还没有太多文献对任务的迁移问题进行考虑。由于任务调度时会结束掉一些正在计算的任务,需要将其放置于工作集群的其他地方。这样很可能造成部分正在运行的任务丢失,即便是 Storm 采用 ack 机制保障丢失的任务会重新计算,系统的性能也会受到一定的影响。现在有少部分的文献考虑了这个问题,但是还不够深入,未来可能从任务调度之后的处理来考虑,提升调度器的性能。

本文针对 Storm 系统的任务调度机制进行综述分析,从以下四个方面进行了分类总结:通过减小任务之间的通信开销;通过调度使得系统运行过程中尽力实现负载均衡;通过弹性机制使得运行过程中对每一个 topology 的配置达到最优;通过资源调度使得运行过程中系统效率达到最高。最后,提出了未来可能的研究方向。

参考文献:

[1] Shvachko K, Kuang Hairong, Radia S, *et al*. The Hadoop distributed file system[C]//Proc of the 26th IEEE Symposium on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE Press, 2010: 1-10.

[2] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[3] Marz N, Warren J. Big data: principles and best practices of scalable realtime data systems[M]. [S. l.]: Manning Publications Co, 2015.

[4] Yasumoto K, Yamaguchi H, Shigeno H. Survey of real-time processing technologies of IoT data streams[J]. Journal of Information Processing, 2016, 24(2): 195-202.

[5] Liu Xiufeng, Iftikhar N, Xie Xike. Survey of real-time processing systems for big data[C]//Proc of the 8th International Database Engineering & Applications Symposium. New York: ACM Press, 2014: 356-361.

[6] Hesse G, Lorenz M. Conceptual survey on data stream processing systems[C]//Proc of the 21st IEEE International Conference on Parallel and Distributed Systems. Piscataway, NJ: IEEE Press, 2015: 797-802.

[7] Gorawski M, Gorawska A, Pasterak K. A survey of data stream processing tools[M]//Information Sciences and Systems. Berlin: Springer International Publishing, 2014: 295-303.

[8] Karunaratne P, Karunasekera S, Harwood A. Distributed stream clustering using micro-clusters on Apache Storm[J]. Journal of Parallel & Distributed Computing, 2017, 108(10): 74-84.

[9] Lu Ruirui, Wu Gang, Xie Bin, *et al*. Stream bench: towards benchmarking modern distributed stream computing frameworks[C]//Proc of the 7th IEEE/ACM International Conference on Utility and Cloud Computing. Washington DC: IEEE Computer Society, 2014: 69-78.

[10] Van Der Veen J S, Van Der Waaij B, Lazovik E, *et al*. Dynamically scaling apache storm for the analysis of streaming data[C]//Proc of the 1st IEEE International Conference on Big Data Computing Service and Applications. Washington DC: IEEE Computer Society, 2015: 154-161.

[11] Abadi D J, Ahmad Y, Balazinska M, *et al*. The design of the borealis stream processing engine[C]//Proc of the 2nd Biennial Conference on Innovative Data Systems Research. 2005: 277-289.

[12] 黄馥浩. 基于 Storm 的微博互动平台的设计与实现[D]. 广州: 中山大学, 2013.

[13] Hunt P, Konar M, Junqueira F P, *et al*. ZooKeeper: wait-free coordination for Internet-scale systems[C]//Proc of USENIX Annual Technical Conference. 2010: 9.

[14] Toshniwal A, Taneja S, Shukla A, *et al*. Storm@ Twitter[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2014: 147-156.

[15] Carney D, Etintemel U, Cherniack M, *et al*. Monitoring streams: a new class of data management applications[C]//Proc of the 28th International Conference on Very Large Data Bases. 2002: 215-226.

[16] Andrade H C M, Gedik B, Turaga D S. Fundamentals of stream processing: application design, systems, and analytics[M]. New York: Cambridge University Press, 2014.

[17] Córdova P. Analysis of real time stream processing systems considering latency[D]. Toronto: University of Toronto, 2015.

[18] Stonebraker M, Cetintemel U, Zdonik S. The 8 requirements of real-time stream processing[J]. ACM Sigmod Record, 2005, 34(4): 42-47.

[19] Liu Yaxiao, Liu Weidong, Song Jiaxing, *et al*. An empirical study on implementing highly reliable stream computing systems with private cloud[J]. Ad hoc Networks, 2015, 35(10): 37-50.

[20] DeMatteis T, Mencagli G. Proactive elasticity and energy awareness in data stream processing[J]. Journal of Systems & Software, 2017, 127(5): 302-319.

[21] Yang Wenjie, Liu Xingang, Zhang Lan, *et al*. Big data real-time processing based on Storm[C]//Proc of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. Washington DC: IEEE Computer Society, 2013: 1784-1787.

[22] Vavilapalli V K, Murthy A C, Douglas C, *et al*. Apache Hadoop Yarn: yet another resource negotiator[C]//Proc of the 4th Annual Symposium on Cloud Computing. New York: ACM Press, 2013: 5.

[23] Junqueira F, Reed B. ZooKeeper: distributed process coordination [M]. [S. l.]: O'Reilly Media Inc, 2013.

[24] Batyuk A, Voityshyn V. Apache Storm based on topology for real-time processing of streaming data from social networks[C]//Proc of the 1st IEEE International Conference on Data Stream Mining & Processing. Piscataway, NJ: IEEE Press, 2016.

[25] Ranjan R. Streaming big data processing in datacenter clouds[J]. IEEE Cloud Computing, 2014, 1(1): 78-83.

[26] Aniello L, Baldoni R, Querzoni L. Adaptive online scheduling in Storm [C]//Proc of the 7th ACM International Conference on Distributed Event-based Systems. New York: ACM Press, 2013: 207-218.

[27] Xu Jielong, Chen Zhenhua, Tang Jian, *et al*. T-Storm: traffic-aware online scheduling in Storm[C]//Proc of the 34th IEEE International Conference on Distributed Computing Systems. Piscataway, NJ: IEEE Press, 2014: 535-544.

[28] Zhang Jing, Li Chunlin, Zhu Liye, *et al*. The real-time scheduling strategy based on traffic and load balancing in Storm[C]//Proc of the 18th High Performance Computing and Communications; the 14th IEEE International Conference on Smart City; the 2nd IEEE International Conference on Data Science and Systems. Piscataway, NJ: IEEE Press, 2016: 372-379.

[29] 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.

[30] Fischer L, Bernstein A. Workload scheduling in distributed stream processors using graph partitioning[C]//Proc of IEEE International Conference on Big Data. Piscataway, NJ: IEEE Press, 2015: 124-133.

[31] Eskandari L, Huang Zhiyi, Eyers D. P-Scheduler: adaptive hierarchical scheduling in Apache Storm[C]//Proc of Australasian Computer Science Week Multiconference. New York: ACM Press, 2016: 1-10.

[32] Peng Boyang, Hosseini M, Hong Zhihao, *et al*. R-Storm: resource-aware scheduling in Storm[C]//Proc of the 16th ACM Annual Middleware Conference. Vancouver: ACM Press, 2015: 149-161.

[33] Cardellini V, Grassi V, Lo Presti F, *et al*. Distributed QoS-aware scheduling in Storm[C]//Proc of the 9th ACM International Conference on Distributed Event-Based Systems. New York: ACM Press, 2015: 344-347.

[34] Pietzuch P, Ledlie J, Shneidman J, *et al*. Network-aware operator placement for stream-processing systems[C]//Proc of the 22nd International Conference on Data Engineering. 2006: 49.

[35] Nardelli M. QoS-aware deployment of data streaming applications over distributed infrastructures[C]//Proc of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics. Piscataway, NJ: IEEE Press, 2016: 736-741.

[36] Farahabady M R H, Samani H R D, Wang Yidan, *et al*. A QoS-aware controller for apache Storm[C]//Proc of the 15th IEEE International Symposium on Network Computing and Applications. Cambridge, MA: IEEE Computer Society, 2016: 334-342.

[37] Chatzistergiou A, Viglas S D. Fast heuristics for near-optimal task allocation in data stream processing over clusters[C]//Proc of the 23rd ACM International Conference on Conference on Information and Knowledge Management. New York: ACM Press, 2014: 1579-1588.

[38] Sun Dawei, Zhang Guangyan, Yang Songlin, *et al*. Re-stream: real-time and energy-efficient resource scheduling in big data stream computing environments[J]. Information Sciences, 2015, 319(10): 92-112.

[39] Chakraborty R, Majumdar S. A priority based resource scheduling technique for multitenant Storm clusters[C]//Proc of International Symposium on Performance Evaluation of Computer and Telecommunication Systems. Piscataway, NJ: IEEE Press, 2016.

[40] Bellavista P, Corradi A, Reale A, *et al*. Priority-based resource scheduling in distributed stream processing systems for big data applications [C]//Proc of the 7th IEEE/ACM International Conference on Utility and Cloud Computing. Piscataway, NJ: IEEE Press, 2015: 363-370.

[41] Chen Yiren, Lee C R. G-Storm: a GPU-aware Storm scheduler[C]// Proc of the 14th International Conference on Dependable, Autonomic and Secure Computing, 14th International Conference on Pervasive Intelligence and Computing, 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress. Piscataway, NJ: IEEE Press, 2016: 738-745.

[42] Chen Zhenhua, Xu Jielong, Tang Jian, *et al*. GPU-accelerated high-throughput online stream data processing[J]. IEEE Trans on Big Data, 2018, 4(2): 191-202.

[43] Qian Wenjun, Shen Qingni, Qin Jia, *et al*. S-Storm: a slot-aware scheduling strategy for even scheduler in Storm[C]//Proc of the 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems. Piscataway, NJ: IEEE Press, 2016: 623-630.

[44] Li Chunlin, Zhang Jing, Luo Youlong. 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(6): 100-115.

[45] Shieh C K, Huang Shengwei, Sun Lida, *et al*. A topology-based scaling mechanism for Apache Storm[J]. International Journal of Network Management, 2016, 27(3).

[46] Zang Zhida, Rao R N. DBalancer: a tool for dynamic changing of workers number in Storm[C]//Proc of the 4th International Conference on Computer Science and Network Technology. Piscataway, NJ: IEEE Press, 2015: 142-145.

[47] Li J, Pu C, Chen Yuan, *et al*. Enabling elastic stream processing in shared clusters[C]//Proc of the 9th IEEE International Conference on Cloud Computing. Piscataway, NJ: IEEE Press, 2017: 108-115.

[48] Evans R. Apache Storm, a hands on tutorial[C]//Proc of IEEE International Conference on Cloud Engineering. Piscataway, NJ: IEEE Press, 2015: 2.

[49] Cardellini V, Nardelli M, Luzi D. Elastic stateful stream processing in Storm[C]//Proc of International Conference on High Performance Computing & Simulation. 2016: 583-590.

[50] Gulisano V, Jiménez-Peris R, Patiño-Martínez M, *et al*. StreamCloud: an elastic and scalable data streaming system[J]. IEEE Trans on Parallel & Distributed Systems, 2012, 23(12): 2351-2365.

[51] Fernandez R C, Migliavacca M, Kalyvianaki E, *et al*. Integrating scale out and fault tolerance in stream processing using operator state management[C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2013: 725-736.

[52] Heinze T, Pappalardo V, Jerzak Z, *et al*. Auto-scaling techniques for elastic data stream processing[C]//Proc of the 8th ACM International Conference on Distributed Event-Based Systems, IEEE International Conference on Data Engineering Workshops. New York: ACM Press, 2014: 318-321.

[53] Heinze T, Jerzak Z, Hackenbroich G, *et al*. Latency-aware elastic scaling for distributed data stream processing systems[C]//Proc of the 8th ACM International Conference on Distributed Event-Based Systems. New York: ACM Press, 2014: 13-22.