计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(12):181−186 [doi: 10.15888/j.cnki.csa.006697]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Storm 集群下基于性能感知的负载均衡策略①

冯馨锐, 谢 彬, 唐 鹏, 秦 健

(中国电子科技集团公司第三十二研究所 信息服务平台室, 上海 201808)
通讯作者: 冯馨锐, E-mail: xiaohang317@126.com

摘 要: Storm 计算框架具有为多源异构大数据提供高效、快速、实时处理的能力. 然而因 Storm 默认的调度策略使用了简单的轮询方法, 无法根据集群动态的负载状态调整其任务的分配. 针对该问题, 提出了基于性能感知的负载均衡策略, 根据节点的处理效率计算其性能感知值, 并通过贪心调度保证节点的任务量与节点处理能力相匹配, 以达到负载均衡的目的. 通过与默认调度算法实验比较, 结果表明该算法能够有效降低 Storm 处理时延, 提高吞吐量和实现集群负载均衡.

关键词: Storm; 实时计算; 性能感知; 负载均衡

引用格式: 冯馨锐,谢彬,唐鹏,秦健.Storm 集群下基于性能感知的负载均衡策略.计算机系统应用,2018,27(12):181–186. http://www.c-s-a.org.cn/1003-3254/6697.html

Load Balancing Strategy Based on Performance-Aware in Storm Clusters

FENG Xin-Rui, XIE Bin, TANG Peng, QIN Jian

(Information Service Laboratory, The Thirty-second Institute of Chinese Electronic Technology Group Corporation, Shanghai 201808, China)

Abstract: As a real-time computing framework, Storm has provided an efficient, fast, and real-time processing ability for multi-source heterogeneous data processing. However, Storm's default scheduler uses a simple Round-robin method and unable to adjust assignments of its task according to cluster's dynamic load status. To solve this problem, this study proposes a load-balancing strategy based on performance-aware. It could calculate Performance-Aware Value (PAV) according to node's processing ability, then greedy scheduling to achieve load balancing, which assigns the amount of computation match with node current processing capacity to achieve load balancing. Compared with the default scheduling algorithm, the results show that this algorithm can effectively reduce the Storm processing delay and improve the throughput, finally achieve cluster's load balance.

Key words: Storm; real-time computing; performance-aware; load balance

随着互联网和大数据技术的快速发展, 全球的数据正在以惊人的速度增长, 因此数据的高时效性, 可操作性需求已经成为研究热点[1]. Storm 作为 Twitter 开源的分布式实时流计算系统, 具有编程接口简单[2], 可拓展性良好和容错机制可靠[3]的优点, 因此在实时推荐, 金融分析和在线机器学习[4]等方面发挥着重要作用.

实时流计算系统及负载均衡、调度技术的研究向来是大数据处理领域不可回避的研究热点. 面对企业生产对流式数据实时处理快速增长的需求, 如何根据运行时环境动态调度流算子任务, 满足低处理时延并最有效利用计算资源具有巨大的研究价值.

本文阐述了 Storm 自带的调度器调度策略, 并对其现存的负载均衡问题进行了分析. 同时, 本文针对流处理系统数据输入率和数据处理延迟等不足, 以集群

① 收稿时间: 2018-05-31; 修改时间: 2018-06-19; 采用时间: 2018-07-03; csa 在线出版时间: 2018-12-03

© 中国科学院软件研究所 http://www.c-s-a.org.cn

的视角提出了一种基于性能感知的负载均衡策略并对其算法进行了实验验证. 结果表明, 该算法能够有效的提高集群资源的利用率, 进而达到动态的负载均衡.

1 Storm 现有调度机制在负载均衡问题分析

Storm 通常应用在计算密集型场景中, 负载均衡技术则是保证实时计算高性能和高吞吐量的有效手段, 而任务调度策略又会直接影响负载均衡的性能.

在 Storm 集群中, 用户提交的作业是一个 topology, 其表现为有向无环图, 会被调度器进行任务的划分. Storm 默认调度策略采用 sort-slot 将任务 (Task) 轮流分配给节点, 如图 1 所示.

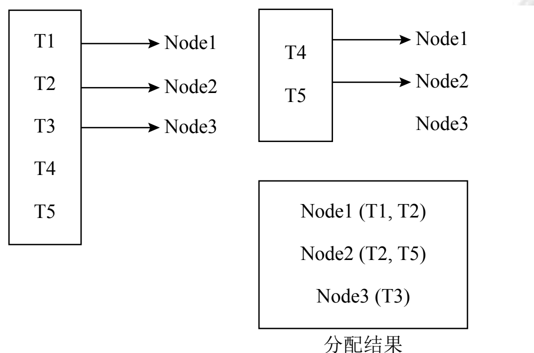

图 1 Storm 默认任务调度示意图

每次新提交的 topology 均会从第一个节点开始分配所划分的任务. 但因每个 topology 的业务逻辑不同, 其任务划分数量也是不定的. 如果仅根据节点剩余 slot 数量进行简单负载均衡会存在以下三点问题:

(1) 这种 sort-slots 会导致整个集群资源分配不均衡, 排序靠前的 supervisor 分配出去的 slot 数量多, 而后面某些很少.

(2) 在集群中添加、删除节点或 worker 进程后, 集群的计算资源会发生变化, 现有调度策略无法根据改变后的可用资源做出有效的调整策略.

(3) Storm 默认的调度机制未充分考虑节点负载情况, 计算容器的固定配置使其不能根据数据输入率实现有效的负载均衡.

针对 Storm 默认调度算法进行改进的相关研究吸引了越来越多研究者的关注. Ding[5]等采用了改变 worker 内部并行度方式应对负载的波动, Navalho[6]等提出了依据数据输入率动态变化调整 worker 等方法. 这些技术仍然是基于计算资源分配量固定的情况下进行的改进, 并未考虑 worker 本身的资源需求和分配不匹配的问题, 不一定适用于 Storm 集群运行时环境. Xiong[7]等以 Task 之间的 Tuple 传输速率作为判定条件, 提出热边调度算法, 将高频热边关联的 Task 作为整体来调度, 以最大程度减轻节点之间通过网络传输 Tuple 的数量. 这种算法能够在降低通信量和负载均衡之间有一个很好的权衡. 然而在真实的生产环境中, 往往是高速的局域网连接的计算机集群, 并不会将带宽视为一种限制资源. 此外, Zhang[8]等采用权值分配策略为每个 slot 定义并且计算优先级, 优化了 Slot 排序的问题, 实现了节点之间的负载均衡. 但是该策略并未考虑负载波动对计算资源的影响以及集群中不同节点之间的性能差异, 只适合应用于某些需要区分任务紧急度的领域中.

2 基于性能感知的负载均衡策略

本文提出了一种基于性能感知的负载均衡策略. 该策略可及时感知节点负载波动, 并可根据节点负载能力分配任务, 在充分利用系统的资源同时, 达到负载均衡的目的.

2.1 算法改进分析及思想

在 storm 集群中, 一个 topology 的任务是分布到多个节点上执行的, 且负载会随着数据输入率的波动而波动. 因各个节点的资源可用性和数据输入率存在一定的差异[9], 会导致任务的处理速度不一. 其实, 对于一个流应用来说, 计算资源供给的不稳定性, 数据输入率不确定性等情况, 终将反应到节点对于任务的处理性能上[10]. 如果将数据输入率, 平均处理延迟等参数结合应用在改进的算法中, 能够更好地反映节点的具体负载情况, 以此为依据来评估节点的处理能力, 量化的重新分配任务, 则可以达到改进负载均衡的目的.

算法思想如下: 首先对 Storm 的监控模块进行重用, 使其可以采集任务部署后的运行信息. 接着工作节点 (supervisor) 每隔一个周期向监控模块发送一个信息采集请求, 监控模块会及时反馈当前拓扑中各个任务的流输入率和平均处理时延. 任务调度器会根据每个节点采集的信息来定义该节点的优先级, 以达到及时的性能感知. 最后, 调度器以集群的视角将任务的计算量与节点处理能力相匹配, 产生新的任务映射方案, 达到负载均衡的目的.

2.2 性能感知方法

在过去的研究中, 系统负载的评价标准一般是

© 中国科学院软件研究所 http://www.c-s-a.org.cn

CPU 利用率, 但是对于一个任务来说, 该任务的 CPU 利用率仅仅表示了其占用了节点计算时间的比重, 并不能体现节点的处理性能[11]. 如表一 所示, 两个负载相同的任务, 一个任务输入率为 1000 tuples/s, 在平均处理延迟为 2 ms 的 node1 上执行, 另一个任务输入率为 2000 tuples/s, 在平均处理延迟为 1 ms 的 node2 上执行. 虽然两个任务的负载相同, 但其所对应的节点处理数据效率是明显不同的.

表 1　负载与 PAV 的对比

node	$r(t)$(tuples/s)	$\tau(t)$(ms)	负载 $r(t)*\tau(t)$	PAV $r(t)/\tau(t)$
1	1000	2	2000	500
2	2000	1	2000	2000

基于上述发现, 本文将$r(t)/\tau(t)$的值称为性能感知值 (PAV). PAV 与任务在某段时间内的输入率成正比, 与相应时间段内的节点平均处理延迟成反比.

性能的计算首先要获取监控拓扑任务的运行数据, 这个方法通过监控模块来完成. 为了避免监控的偶然因素, 调度器将采用最近连续三次监控的平均值. 单个节点的 PAV 可以是其上各任务的 PAV 之和, 也可是节点总数据输入率和与平均处理延迟的商. 本文从集群的视角来考虑, 采用后者来计算, 并对结果降序排列, 为之后的负载均衡做准备. 具体的 PAV 计算方法如下, 流程图如图 2 所示.

算法 1. 计算节点 PAV

输入: 任务运行时信息
输出: PAV_node[N]

Step1: receive task runtime information
Step2: **FOR** node belongs to cluster **DO**
get the total rate: sum all task rate on node
get the average latency
END FOR
Step3: PAV_ node[i] total rate/average latency
Step4: Sort_down(PAV_node)

合理配置采样周期对于调度性能的提高具有重要意义. 如果采样周期过短, 集群信息的采集会带来很大开销, 对任务实时性会有一定的影响, 而如果周期过长, 对于波动较大的数据流, 会影响其性能感知准确性. 根据以往经验, 数据平均到达率为 3000 tuples/s 且服从泊松概率分布时, 采样周期设置在 10 s~15 s 为佳[12]. 本文实验环境设置的采样周期为 10 s.

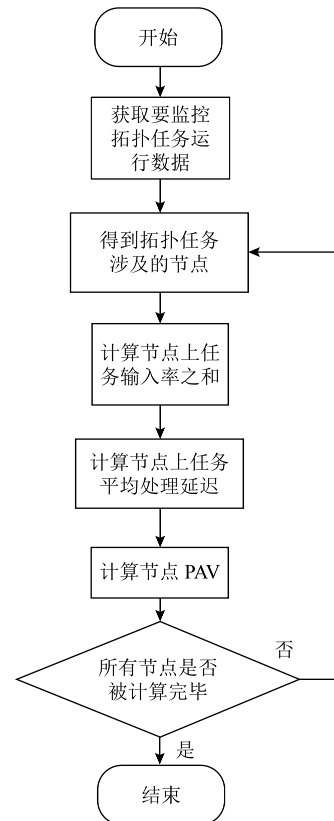

图 2　节点 PAV 计算流程图

我们可以根据 PAV 来评估一个节点当前的处理性能, PAV 高的节点应该被分配更多任务. 一般而言, 每个节点的计算资源是固定的, 随着提交任务的增多, 节点处理性能会变差, 即 PAV 会降低, 此时需将某些任务提交给 PAV 高的节点去处理, 可减小性能差节点的计算量.

2.3　基于 PAV 的负载均衡算法实现

结对交换算法是解决负载均衡的常用算法, 如图 3 所示, 其基本思路是将 N 个节点分为 N/2 组, 负载最小的节点与负载最大的在一组, 负载次小的与负载次大的在一组, 依此类推, 若还剩中间节点, 则忽略处理. 在组内通过任务的迁移, 来减少节点之间的负载不平衡.

结对交换算法对性能有差异的节点任务交换, 在一定程度上缓解了性能差节点上的处理压力, 但是不能以集群视角来分配任务, 且对有些情形不起作用. 因此, 本文以结对交换算法为启发式优化算法, 提出了基于 PAV 的负载均衡算法.

负载均衡问题本质上是一个 NP 完全问题[13], 到目前为止还没有完全有效的解法. 对于这类问题, 用贪心选择策略有时候会设计出一个比较好的近似算法. 本

© 中国科学院软件研究所 http://www.c-s-a.org.cn

文在贪心算法的基础上调用结对交换算法, 会使负载进一步平衡. 本文关键部分算法 2 如下, 用到的相关符号参考表 2.

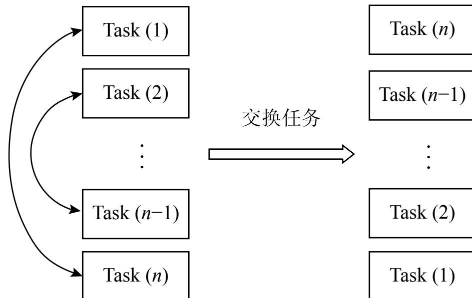

图 3 结对交换算法示意图

表 2 符号含义表

符号	含义
N	Strom 集群 slave 数目
M	任务数目
PAV_node	节点的性能感知数组
PAV_task	任务的性能感知数组
Storm_load	任务在集群中的分布二维数组
T	Topology 被分割后的任务

算法 2. 基于 PAV 的负载均衡算法

输入: PAV_node[N], PAV_task[M], N, M
输出: Strom_load[N][M]

```
Step1: Sort_up(PAV_task)
Step2: FOR i=0 TO M DO
Sort_down(PAV_node)
Insert(Storm_load[N][M])
Update(PAV_node)
END FOR
Step3: Sort_up(PAV_node)
DivideGroup(PAV_node,N/2)
Changed=false
Step4: FOR i=0 TO N/2 DO
Step4.1 REPEAT
InPair(T_i,T_j)
UNTIL(没有可以考察的任务对)
Step4.2 NumofSelectCand = SelectCand()
Step4.3 IF NumofSelectCand ==0 THEN
GOTO 4.1
ELSE
SelectBest() ExchangeBest()
Update(Storm_load[N][M])
Changed=true
END IF
END FOR
```

算法说明: Step1 在通过 2.2 节的方法获取节点和任务的性能感知数组后, 对 PAV_task 按照 PAV 值升序排列. Step2 通过贪心算法, 按照 PAV 高的节点应该被分配更多任务的原则, 将负载较重任务安排在 PAV 值高的节点上, 每放置一次后因为节点性能会因为任务数量发生变化, 所以要进行一次更新, 来对新的计算资源进行评估. Step3 和 Step4 是在此基础上调用结对交换算法, 以求进一步的负载均衡. Step4.2 用于选定候选的任务对, 本算法的候选条件为:

$$(\text{PAV_task}[T_i] > \text{PAV_task}[T_j])\&\&(\text{PAV_task}[T_i] - \text{PAV_task}[T_j] < \text{PAV_node}[S_i] - \text{PAV_node}[S_{N-i-1}])$$

Step4.3 在选定任务对后, 以得分之差的绝对值最小任务对为最优任务对:

$$\text{Score}(T_i,T_j)=\min\{\text{abs}(\text{PAV_task}[T_i]-\text{PAV_task}[T_j])\}$$

这样做主要是为了兼顾系统在调度过程中付出的开销. 最后在任务迁移同时对 Storm_load 进行更新.

此算法的可行性也容易证明. 由于在分配任务之前引入了对节点的性能感知, 保证了算法可以对整个集群的性能进行全面评估. 贪心调度算法可以依据任务和节点的 PAV, 使节点的负载与其计算能力相匹配, 避免了在一次 topology 提交过程中引起的节点过载问题. 结对交换法可以解决局部负载过重的问题, 优化系统的处理性能. 在算法复杂度方面, 由于算法在贪心调度的计算和结对交换任务上增加的时间复杂度都是 $O(n^2)$ 级别的, 所以整个调度算法的时间复杂度仍然为原来默认调度的复杂度级别, 即 $O(n^2)$.

3 实验部分

继 Storm0.8.0 版本后, 其资源调度器变为可插拔式的 (Pluggable Scheduler)[14], 可用于自定义任务的分配算法. 实验利用了 Pluggable Scheduler 实现了本文自定义的调度算法, 并且对 Storm 的 Trift 接口进行了二次封装, 使其可采集 topology 运行时的信息和 executor 的负载量, 以期实现共同辅助实验结果的分析. 实验环境为 OpenStack 下的 6 台 CentOS7.2 云主机组成, 配置见表 3 所示.

Master 配置相对于 Slaves 较高, 因为主节点需要承担任务的接受、部署和调度, Slaves 选择了资源异构的硬件配置, 可以在这基础上验证 PAV 不同节点上负载均衡策略的效果.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

表 3 实验环境配置

节点名称	配置
Master	四核 3.0 GHz CPU, 8 GB 内存, 50 GB 硬盘
Slave1	双核 3.2 GHz CPU, 4 GB 内存, 50 GB 硬盘
Slave2	双核 4.0 GHz CPU, 4 GB 内存, 50 GB 硬盘
Slave3	双核 3.0 GHz CPU ,4 GB 内存, 25 GB 硬盘
Slave4	双核 2.6 GHz CPU, 4 GB 内存, 25 GB 硬盘
Slave5	双核 2.6 GHz CPU, 2 GB 内存, 25 GB 硬盘

实验的目的在于对性能感知的改进方法做相关性测试, 并且以典型的大数据应用 WordCount 进行了 4 组实验. 4 组实验分别针对 Storm 优化的处理时延测试、系统吞吐量测试、固定数据输入率下 CPU 负载均衡性测试以及满足泊松概率分布的动态数据流下 CPU 资源利用率测试.

3.1 不同数据输入率下平均处理时延

流处理系统最主要的是保证应用的低处理时延, 处理时延是衡量一个调度系统是否有效的重要指标. 实验一包括了 4 组对比实验, 分别以 1000 条/s、2000 条/s、3000 条/s、4000 条/s 的速率向 Redis 中写入数据, 然后 Spout 从其中读取数据. 图 4 表示的是默认调度和本文调度在对应数据输入率下的平均处理时延的比较. 由实验结果可知, 默认调度算法的处理时延会随着发送端数据量的累积而增大, 当速率达到 4000 条/s 时, 时延增长约 40%, 而基于性能感知的调度算法可及时感知任务的计算量, 并将其分配到性能最优的节点上, 因此整体平均时延并不会有显著增大.

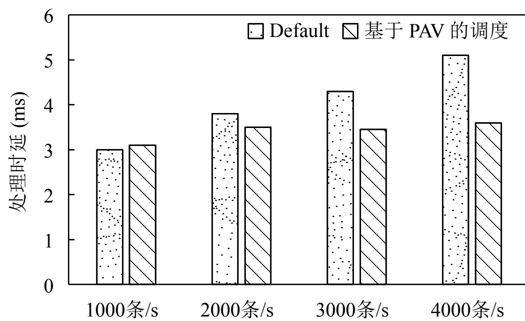

图 4 处理时延对比

3.2 系统吞吐量测试

实验二通过改变 topology 中的 task 数量, 对系统吞吐量进行测试. 从图 5 可以看出, 在 task 数量小于 6 时, 默认的算法效率略高于改进算法, 这主要是由于 task 数量较少时, 集群负载很低, 而默认算法的实现较为简单, 因此性能好于改进算法. 随着 task 数量的增加, 改进算法优势就体现出来了. 这是因为基于 PAV 的调度能有效感知集群中每个节点的负载情况, 将任务和节点处理能力相匹配, 从而避免个别节点的过载. 随着 task 数量的继续增加而超过 10 个时, 两种算法的吞吐量均出现了不同程度的下降. 这是由于这些 task 通信所需的频繁的序列化与反序列化操作及 task 所对应的 executor 间的切换, 均会消耗很多的计算资源, 最终导致吞吐量下降.

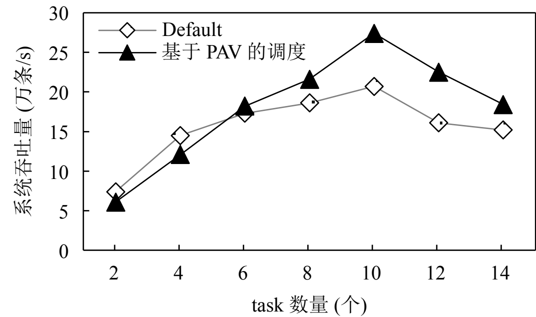

图 5 系统吞吐量比较

3.3 固定数据输入率下 CPU 负载均衡性测试

实验三采集了在固定数据输入率 (2000 条/s) 下流处理过程中集群 CPU 资源在异构节点上的负载分布, 进而考察基于性能感知负载均衡策略效果. 如图 6 所示, 当系统稳定后, 计算能力较好的节点 (Slave2 和 Slave3) 因其使用基于 PAV 的负载均衡调度, CPU 使用率相较默认调度有了显著上升, 而 Slave5 由于计算能力略低, 调度器在感知性能后会将其上任务根据结对交换算法进行迁移, 因此与默认调度策略相比, 其 CPU 使用率会有显著下降, 约 50% 左右.

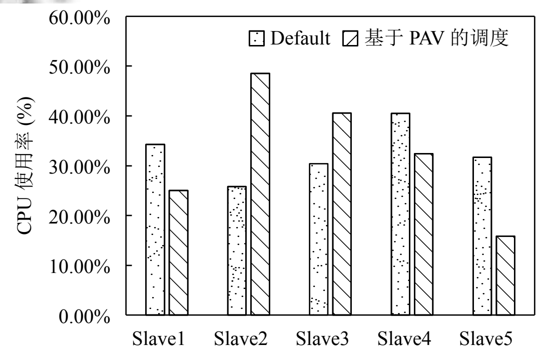

图 6 集群节点 CPU 负载分布

3.4 动态数据流下 CPU 资源利用率测试

为了能够满足在真实的生产环境下系统随着数据到达率的变化而处理数据, 实验四在实验三的基础上, 通过 Python 数据发生器作为数据源产生数据流, 且到

© 中国科学院软件研究所 http://www.c-s-a.org.cn

达的时间遵循泊松分布. 公式如下:

$$P(x=k)=\frac{e^{-\lambda}\lambda^{k}}{k!} \tag{1}$$

测试数据泊松分布参数 λ 从 1000 开始, 分别增至 2000、3000, 统计不同性能节点的 CPU 资源使用率.

表 4 给出了各节点在 λ 参数递增的动态数据流下, CPU 资源平均利用率在默认调度和基于性能感知的负载均衡调度下的对比. 重点考察 Slave2 和 Slave5. 随着 λ 的增大, 性能较好的 Slave2 其 PAV 调度 CPU 利用率相较默认调度会有显著增加, 当 λ=4000 时, 两者相差近 20%, 而 Slave5 性能较差 (见表 3), 其 PAV 会随着 λ 增加而下降, 依据本文贪心调度策略, 上面的任务会被调度到别的节点. 所以尽管输入率动态增加, CPU 资源利用率反而有所下降. 这是改进算法对节点性能感知后动态迁移任务的结果, 进一步体现了真实环境下动态的负载均衡.

表 4 动态数据流下集群 CPU 资源平均利用率 (%)

节点	λ=2000		λ=3000		λ=4000	
	Default	PAV	Default	PAV	Default	PAV
Slave1	25.5	29.5	30.6	44.8	37.5	51.7
Slave2	28.4	26.0	32.5	45.9	35.5	55.0
Slave3	34.25	40.0	39.0	44.2	49.6	46.7
Slave4	33.0	38.0	38.5	40.5	48.0	42.2
Slave5	37.1	41.0	42.5	36.6	53.2	34.7

综合以上 4 组实验结果可以看出, 通过对时延和流输入率的感知, 结合负载均衡迁移机制, 使任务可以在集群间依据处理的需要调度到适合的节点, 是提高 Storm 集群资源利用率和处理效率的有效途径.

4 结束语

本文基于 Storm 实时流处理调度机制, 针对默认调度策略资源分配不均衡和无法动态感知数据的输入率等问题, 提出了基于性能感知的负载均衡策略. 通过实验验证, 结果表明基于性能感知的负载均衡策略可有效促进 Storm 集群负载均衡, 提高吞吐量, 并降低处理时延. 本文的局限性在于, 对于性能感知的调度策略方面未考虑设置阈值和动态调整阈值以达到更好的调度性能, 这也是下一步的研究方向.

参考文献

1 张引, 陈敏, 廖小飞. 大数据应用的现状与展望. 计算机研究与发展, 2013, 50(S2): 216–233.
2 马可, 李玲娟. 分布式实时流数据聚类算法及其基于 Storm 的实现. 南京邮电大学学报 (自然科学版), 2016, 36(2): 104–110.
3 孙大为, 张广艳, 郑纬民大数据流式计算关键技术及系统实例. 大数据流式计算关键技术及系统实例. 软件学报, 2014, 25(4): 839–862.
4 陈敏敏, 王春新, 黄奉线. Storm 技术内幕与大数据实践. 北京: 人民邮电出版社, 2015. 1–3.
5 Fu TZJ, Ding JB, Ma RTB, *et al*. DRS: Dynamic resource scheduling for real-time analytics over fast streams. 2015 IEEE 35th International Conference on Distributed Computing Systems. Columbus, OH, USA. 2015. 411–420.
6 Navalho D, Duarte S, Preguiça N, *et al*. Incremental stream processing using computational conflict-free replicated data types. Proceedings of the 3rd International Workshop on Cloud Data and Platforms. Prague, Czech. 2013. 31–36.
7 Xiong AP, Wang XW, Zou Y. Scheduling algorithm based on storm topology hot-edge. Computer Engineering, 2017, 43(1): 37–42.
8 张楠, 柴小丽, 谢彬, 等. Storm 流处理平台中负载均衡机制的实现. 计算机与现代化, 2017, (12): 65–70. [doi: 10.3969/j.issn.1006-2475.2017.12.013]
9 Arasu A, Cherniack M, Galvez E, *et al*. Linear road: A stream data management benchmark. Proceedings of the Thirtieth International Conference on Very Large Data Bases. Toronto, Canada. 2004. 480–491.
10 樊明璐. 流式大数据处理平台中资源动态调度技术研究[硕士学位论文]. 北京: 北京工业大学, 2016.
11 Xu JL, Chen ZH, Tang J, *et al*. T-Storm: Traffic-aware online scheduling in storm. Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems. Washington, DC, USA. 2014. 535–544.
12 范嘉骅. Storm 中自适应任务调度的研究与优化[硕士学位论文]. 上海: 上海交通大学, 2016.
13 聂如云. 云服务系统任务调度负载均衡的研究[硕士学位论文]. 北京: 首都经济贸易大学, 2016.
14 Martin A, Knauth T, Creutz S, *et al*. Low-overhead fault tolerance for high-throughput data processing systems. Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS). Washington, DC, USA. 2011. 689–699.

© 中国科学院软件研究所 http://www.c-s-a.org.cn