doi:10.3772/j.issn.1002-0470.2021.01.003

# 细粒度的流计算执行效率优化方法①

魏 波②******** 刘晓东* 阎雨庭* 赵晓芳* 袁雨馨**** 唐宏伟*

(*中国科学院计算技术研究所 北京 100190)

(**公安部第一研究所 北京 100044)

(***中国科学院大学 北京 100049)

**摘 要** 流计算引擎如Storm被广泛地应用于大数据实时处理中,以提高数据处理的执行效率,但该引擎因统计信息粒度粗、无法准确定位性能瓶颈,造成了难以提高数据处理的执行效率问题。为此,本文提出了一种细粒度流计算执行效率优化方法,该方法以元组为中心进行细粒度的性能分析,包含性能瓶颈识别算法和性能瓶颈缓解算法,支持量化地选择最优参数配置以提高执行效率。实验结果表明,在3个标准程序32个不同配置场景下,该方法能够准确地识别流应用的性能瓶颈,识别率为100%,应用的执行效率提高了70%。

**关键词** 流计算引擎;执行效率优化;细粒度;元组

## 0 引 言

大数据时代,传统的批处理计算MapReduc[1]已经不能满足数据实时处理的需求,流计算应运而生[2],广泛地应用于物联网、日志分析、网络监控、金融等领域。其中,主流流计算引擎[3-4]包括Storm[5-6]、Heron[7]、Spark Streaming[8]、Cloud Dataflow[9]、Apache Flink[10]等。

本文以Storm计算引擎为基础,重点研究如何提高流计算应用效率的问题。典型的Storm应用的执行效率受到诸多方面因素的影响,如可用的硬件资源、配置参数、应用的逻辑结构等,这使得定位应用的性能瓶颈成为难题。

截止目前,已有研究[11-15]主要是围绕Storm的性能测试,但没有对Storm的性能影响因素进行深入分析。例如,Lu等人[11]提出了流计算领域的benchmark,并采用默认的参数配置,简单地对Storm等一些流计算引擎进行了比较,没有深入地探讨Storm的性能瓶颈。Lopez等人[13]的实验结果说明了并行度对Storm应用吞吐率的影响,但没有具体研究性能瓶颈产生的原因及识别方法。Kulkarni等人[7]分析了Storm的不足,但侧重于Storm的扩展性、调试性、可控性和集群资源共享效率上的不足。Chintapalli等人[16]指出Storm集群扩展性存在瓶颈,并给出了优化方案。

本文以流计算引擎Storm应用执行过程中的性能瓶颈为研究点,提出以元组为中心的细粒度的流计算执行效率优化方法,解决该引擎因统计信息粒度粗、无法准确定位性能瓶颈从而造成执行效率难以提高的问题。研究和解决问题的思路包括以下3方面。

(1) 首先提出并实现了以元组为中心的性能分析器(Tuple-centered Profiler,简称T-Profiler)。

(2) 在此基础上,提出了细粒度流计算执行效率优化方法,该方法以元组为中心进行细粒度的性能分析,包含性能瓶颈识别算法和性能瓶颈缓解算法。

(3) 最后,使用Intel Storm benchmark的3个标

① 国家重点研发计划(2018YFB0904503)资助项目。

② 男,1983年生,博士,副研究员;研究方向:监控视频编解码,集成电路设计与测试,云计算与大数据,网络与信息安全;联系人,E-mail:weibo018@163.com

(收稿日期:2020-02-06)

准程序在 32 个不同的配置参数下验证了提出方法的有效性。

## 1　Storm 相关概念及执行过程

### 1.1　Topology、Spout 及 Bolt

Storm 是一款分布式流计算引擎，用于实时、高效地处理流数据。Storm 的数据处理过程被抽象成一个 DAG（directed acyclic graph）结构 Topology，如图 1 所示，包括 Spout 和 Bolt 两种类型的节点；各节点间通过订阅关系相连接；将数据流看作无界的元组序列，将元组作为数据处理的最小单元。

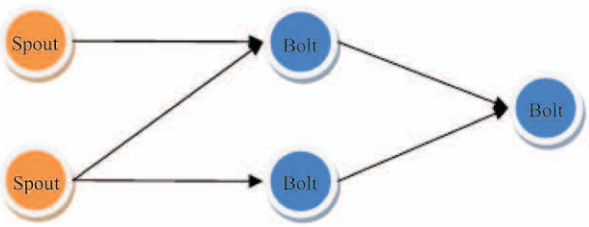



**图 1　Topology 的结构示意图**

Spout 节点是整个 Topology 的输入源，负责接收外界的输入流，并发送给对应的 Bolt 节点。Bolt 节点是 Topology 的处理单元，负责接收所订阅的 Spout 或 Bolt 的流数据，并进行加工处理，再转发给其他 Bolt 节点。

此外，一个 Topology 还包含一类叫做 Acker 的特殊 Bolt，它们的作用是跟踪每个元组的执行，当元组处理完毕时，Acker 会告知 Spout 节点元组已处理完成；当元组处理失败时，Acker 会通知 Spout 进行重传。

### 1.2　分组策略

Spout 和 Bolt 是一个或多个 Task 并发处理数据流，由此，Storm 规定了分组策略，用来确定数据流发送给 Bolt 的哪个 Task 处理。Storm 提供的分组策略有随机分组（Shuffle）、字段分组（Field）、部分键值分组（Partial Key）、全分组（All）、全局分组（Global）、无分组（None）、直接分组（Direct）、局部或随机分组（Local or Shuffle）。其中，Shuffle 和 Field 分组较为常用。

### 1.3　Storm 的执行过程

Storm 集群包含主节点和工作节点，主节点和工作节点之间通过 Zookeeper 集群协调。主节点上运行 Nimbus 服务，负责接收用户提交的 Topology，并分配任务给各工作节点；工作节点上运行 Supervisor 服务，负责接收 Nimbus 的任务分配，并启动 Worker 进程执行任务。

Storm 有 3 个实际执行 Topology 的实体。

（1）Worker 进程是 1 个 JVM 进程，其中运行多个 Spout 和 Bolt 的多个 Executor。

（2）Executor 运行 1 个 Spout 或 Bolt 的多个 Task。1 个 Executor 中的所有 Task 顺序执行，因而 Executor 是并行的基本单元。

（3）Task 是执行实际任务的 Spout、Bolt 实例。

Storm 为每个 Worker 进程维护一个传输队列，为每个 Executor 维护一个发送队列和一个接收队列，这些队列用于传递数据流。数据流传递过程的一个示例如图 2 所示。（1）Spout 的 Task 产生数据流进入到发送队列；（2）目标 Bolt 和 Spout 在同一 Worker 进程中的数据流直接传入 Bolt 的接收队列，而目标 Bolt 在其他 Worker 进程中的数据流暂时传入传输队列；（3）Bolt 的 Task 从接收队列获取数据流进行处理；（4）Bolt 的 Task 将加工后的数据流传入发送队列；（5）同过程（2）；（6）传输队列中的数据流将被转发给其他 Worker 进程的 Bolt 的接收队列。

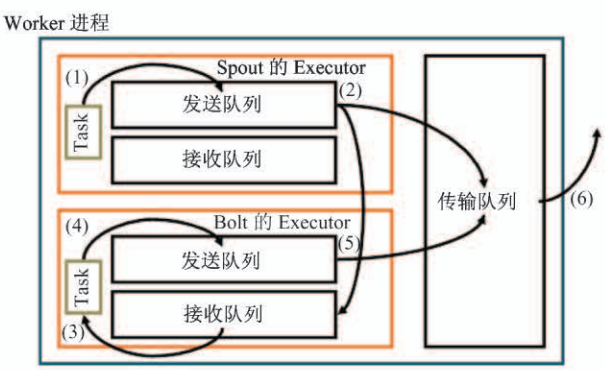



**图 2　数据流传输过程示例图**

## 2　细粒度的流计算执行效率优化

首先设计以元组为中心的性能分析器 T-Profi-

ler,然后阐述利用分析器获得的信息识别瓶颈算法,最后提出缓解瓶颈的算法。

### 2.1　T-Profiler 是以元组为中心的性能分析器

Storm 只支持提供粗粒度的指标。本文提出了一个以元组为中心的性能分析器 T-Profiler,并给出细粒度的指标,即 Spout 的发射间隔、每个元组在各阶段的执行时间,同时分析器可根据这些细粒度的指标识别性能瓶颈,能进一步缓解瓶颈。分析器概况如表 1 所示。

**表 1　Storm 性能分析器概况**

| 步骤 | (1)记录元组执行过程中经历的所有状态 | (2)处理 log 文件,根据各时间点求出元组执行过程的各段时间 | (3)运用瓶颈识别算法识别瓶颈,并使用缓解算法进行缓解 |
|---|---|---|---|
| 输出 | log 文件 | 收集结果 | 分析结果 |

性能分析器的第(1)步是记录元组状态,状态信息包括当前时间、元组的 ID、目标 Task 的 ID、源 Task 的 ID。根据 1.3 小节的描述,一个元组的处理需要经过发送、传输、接收 3 种队列,因此性能分析器不仅要能统计出 Bolt 的执行时间,还要统计出各队列的等待时间。以图 3 为例,Spout 发送元组到 Bolt 需要经过 3 个队列(当 Spout 和 Bolt 属于同一 Worker 进程时,无传输队列),在第 1、3、5 个圆圈处,记录元组的入队状态(包括发送者以及接收者的 Task ID);在第 2-4 个圆圈处,记录元组的出队状态;在第 1 个三角处,记录元组处于 Spout 发射状态;在第 2、3 个三角处,分别记录元组的处理开始状态与结束状态。根据 1.1 小节的说明,Acker 可以确定元组的完成情况,于是,我们在 Acker 确定元组完成时,记录元组处于完成状态。通过状态信息中的 Task ID 和元组 ID 我们可以很清楚地了解元组的整个执行过程。上述所有的状态信息将被写入到 log 文件中,便于之后的处理。

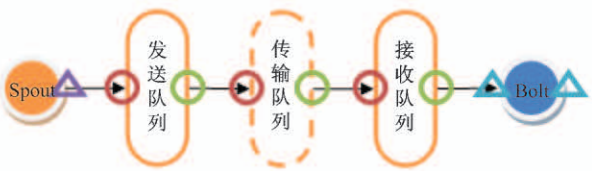



**图 3　状态记录示意图**

性能分析器的第(2)步是统计各元组的执行情况。通过识别元组的 ID,可以将元组各自的状态信息划分出来;Task ID 可以将状态信息与元组的实际执行过程对应起来。于是,利用状态信息中的时间点,可以统计出各元组的总执行时间、队列等待时间、各 Bolt 处理时间,以及 Spout 向某个 Bolt 发射元组的间隔。

性能分析器的第(3)步是进行瓶颈分析。分析器根据之前收集的元组执行信息,运用瓶颈识别算法定位瓶颈,并运用瓶颈缓解算法给出最优参数配置。

### 2.2　瓶颈识别算法

一般地,Topology 中某个 Bolt 执行速度过慢,将会导致该 Bolt 的接收队列阻塞,使得后续元组在接收队列的等待时间呈上升趋势,这个规律同样可推广到发送队列和传输队列。因此,为了确定瓶颈所在,提出了瓶颈识别算法,通过识别队列等待时间序列是否为上升趋势来判断,如算法 1 所示。

算法 1 包含 3 个输入参数,分别是一段时间内所有元组在某队列中的等待时间序列、波动因子和上升趋势的最小跨度。算法返回值为该队列是否为瓶颈。其中,波动因子用于确定值的波动范围,算法会将波动范围内的值看作一个值,其缺省值为 0.5 ms;上升趋势的最小跨度默认值为时间队列长度的 1/3。

算法 1 的 2 ~ 4 行分别初始化上升趋势判断标志、上升趋势起始位置和当前判断到的位置;6、7 行,当前位置高于前一位置,判断标志加 1;8 ~ 18 行,当前位置比起始位置低时,重新设定起始位置,并将判断标志清 0,其中 10 ~ 16 行为查找新起始位置的部分,即向后查找到高于旧起始位置的地方停止查找,其前一位置即为新起始位置;19 ~ 21 行判断是否大于最小跨度,若小于,则终止算法;24 ~ 26 行判断是否含有上升趋势。

### 2.3　瓶颈缓解算法

缓解瓶颈 Bolt 的方式有多种,比如增加 Bolt 的并行度、通过硬件加速 Bolt 的执行等,但是它们的中心思路都是加强瓶颈 Bolt 的处理能力,本文仅讨论增加并行度的方法。

**算法 1** 瓶颈识别算法

输入：

1. 队列等待时间序列 *tSeq*
2. 波动因子 *thres*
3. 存在上升趋势的值的最小数量 *num*

输出：true 或 false，分别表示队列是否阻塞

```
1  function IdentifyBottleneck (tSeq, thres, num):
2   count = 0 // 判断上升趋势的标志
3   start = 0 // 趋势判断的起始 tSeq 的 index
4   cur = 1 // 当前判断到的 tSeq 的 index
5   while cur < tSeq.size()
6   if tSeq.get(cur) > tSeq.get(cur - 1) + thres
7   count + +
8   else if tSeq.get(cur) < = tSeq.get(start) + thres
9   count = 0 // 重新判断趋势
10   while cur < tSeq.size()
11     if tSeq.get(cur) > tSeq.get(start) + thres
12       cur--; // 避免将 GC 的位置作为起始 index
13       break
14      end if
15      cur + +
16     end while
17     start = cur // 重设趋势判断的新的起始 index
18    end if
19    if tSeq.size() - start < num
20     break
21    end if
22    cur + +
23  end while
24  if tSeq.size()-start > = num and count > 0
25    return true
26  else return false
27  end if
28 end function
```

**算法 2** 瓶颈缓解算法

输入：

1. 时间序列 *tSeq*
2. 是否为 Spout 发射间隔 *flag*
3. 区间划分因子 *factor*，区间数为 2 × *factor* + 1

输出：

1. 时间序列的代表值 *value*

```
1  function GetTypicalValue(tSeq, flag, factor):
2   min = min(tSeq) // 求序列最小值
3   avg = avg(tSeq) // 求序列平均值
4   spc = ((avg-min) / (factor + 0.5)( // 区间间隔
5   len = 2 × factor + 1
6   初始化 segSum[len], segNum[len]
7   record(tSeq, segNum, segSum, min, spc)
8   cnt = 0
9   if flag
10    cur = 0
11  else cur = seqNum.size()-1
12  end if
13  while cnt/tSeq.size() < 0.1
14    cnt + = segNum[cur]
15    if flag
16     cur ++
17    else cur--
18    end if
19  end while
20  if flag
21    value = segSum[cur - 1]/ segNum1[cur-1]
22  else
23    value = segSum[cur + 1]/segNum[cur + 1]
24  end if
25  return value
26 end function
```

经过 2.2 节算法的计算，可能会得到某个接收队列产生阻塞的结论，此时根据对应 Bolt 的执行时间和元组到来的间隔，可以得出较为合适的 Bolt 并行度，通过调节并行度可以缓解瓶颈。元组到来的时间间隔不会超过 Spout 发射元组过来的间隔，因此用 Spout 发射间隔近似代替它。由于性能分析器得到的 Bolt 执行时间和 Spout 发射间隔均是时间序列，为此需要获取该时间序列的代表值。代表值的获取算法如算法 2 所示。

算法 2 的 2 ~ 7 行将时间序列划分成多个区间；8 ~ 24 行求取代表值，对于 Spout 发射间隔序列，算法会舍弃最小的包含时间数不到总数的 0.1 的区间，将剩下的最小区间的平均值作为代表值，而对于 Bolt 执行时间序列，则采取相反的操作，舍弃最大的包含时间数不到总数的 0.1 的区间，将剩下的最大区间的平均值作为代表值。这里 Spout 选择最小区间的平均值，Bolt 选择最大区间，是为了让计算出来的 Bolt 并行度尽可能大。其中，record 函数的功能是将时间序列划分到 2 × *factor* + 1 个区间内，分别用两个数组保存区间内的时间总数和时间总和，如算法 3 所示。

**算法 3**　时间数算法

输入:

1. 时间序列 *tSeq*
2. 保存每个区间内数据数量的数组 *segNum*
3. 保存每个区间内数据总和的数组 *segSum*
4. 序列 *tSeq* 的最小值 *min*
5. 区间间隔 *spc*

```
1  function record(tSeq, segNum, segSum, min, spc):
2   for time in tSeq
3     index = ((time-min) / spc
4     if index < segNum.size()
5      segNum[index] ++
6      segSum[index] + =time
7     end if
8   end for
9  end function
```

在得到 Spout 发射间隔和 Bolt 执行时间的代表值之后,就可以计算出 Bolt 的合适并行度。根据1.2小节对分组策略的介绍,分组策略会规定数据流如何划分给并行度不为 1 的 Bolt 的某个 Task。因而,如何计算并行度与所采用的分组策略有关,本文仅以 Shuffle 分组策略为例,给出并行度的确定方法。Shuffle 分组策略如算法 4 所示。

**算法 4**　分组策略算法

输出:数据流所要送往的 Task 编号

```
1  function ShuffleGrouping():
2  targetTask = 目标 Bolt 的所有 Task 的编号集合
3   打乱集合 targetTask 中编号的顺序
4   index =0// targetTask 的索引
5   size  =targetTask.size()
6   while true
7    current = index ++ //原子操作
8    if current < size
9     return targetTask.get(current)
10    else if current = =size
11     index =0
12     打乱集合 targetTask 中编号的顺序
13     return targetTask.get(0)
14    //current > size 时发生线程冲突
15    end if
16  end while
17  end function
```

从中可以得知,Shuffle 分组策略会将数据流中的元组均匀地分发给 Bolt 的所有 Task,因此当 Bolt 的并行度增加时,其执行时间会等比例地减少。于是,可以得到如下结论,Bolt 的合适的并行度等于 Bolt 的执行时间除以 Spout 向 Bolt 发射元组的时间间隔。

## 3　算法验证

为验证瓶颈识别算法以及瓶颈缓解算法的有效性,本文采用了 Intel 的 benchmark[17] 进行算法验证。选取其中典型的 SOL、WordCount 和 UniqueVisitor 3 个 benchmark 进行测试。

### 3.1　实验方法

通过在 Apache Storm 1.0.2 版本[6] 上实现性能分析器进行实验。表 2 给出了实验中所用节点的配置信息。为了实验结果的稳定性,关闭了 Intel turbo boost[18]。使用一个 *A* 类节点配置了 Standalone 模式的 Zookeeper 服务。

**表 2**　节点配置信息

| 节点类型名 | CPU | 内存 |
|---|---|---|
| *A* | Xeon E5620@2.40 GHz,16cores,开启超线程 | 47 GB |
| *B* | Xeon E7-4820@2.00 GHz,64cores,开启超线程,关闭 Intel turbo boost 服务 | 504 GB |

Storm 集群共使用 5 个节点,其配置信息如表 3 所示。

**表 3**　集群配置信息

| 角色 | 描述(节点配置信息见表 2) |
|---|---|
| 主节点 | 使用一个 *A* 类节点。 |
| 工作节点集群 | 使用两个 *B* 类节点作为工作节点集群。 |
| Zookeeper 节点 | 使用一个 *A* 类节点运行 Zookeeper 服务。 |
| Kafka 节点 | 使用一个 *A* 类节点运行 Kafka[19-20] 服务。 |

在 Topology 的执行方式方面,首先让 Spout 等待 3 s,以保证所有的 Bolt 都初始化完毕;再按10 ms

间隔发射前 300 个元组，以保证元组执行时间达到稳定状态；最后按正常时间间隔发射之后的元组。Topology 可配置的运行参数有 Worker 进程数、每个 Spout/Bolt 的 Executor 数和 Task 数。实验中，为每个 benchmark 随机配置了多组参数验证瓶颈识别算法的有效性，本文仅展示其中几个有代表性的实例。

### 3.2　Topology 的结构

SOL 的 Topology 结构如图 4 所示，其中 Spout 随机发射长度为 100 的字符串；Bolt1 到 Bolt5 都只负责将上一个节点发来的字符串向后发送出去。

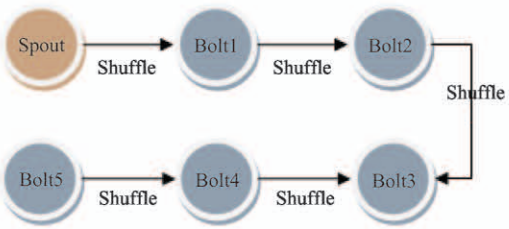



**图 4　SOL 的 Topology 的结构图**

WordCount 的 Topology 结构如图 5 所示，其中 Spout 从 Kafka 服务器接收数据流，并发送给 Split，数据流中的每个元组都是一个句子；Split 是 Bolt 节点，负责将接收到的每个句子元组划分为单词元组，并发送给 Count；Count 也是 Bolt 节点，负责计数单词元组。

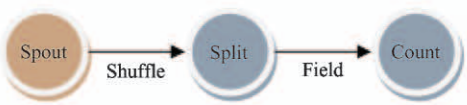



**图 5　WordCount 的 Topology 的结构图**

UniqueVisitor 的 Topology 结构如图 6 所示，Spout 负责从 Kafka 服务器接收数据流，并发送给 View，数据流中的每个元组都是一次页面访问信息；View 是 Bolt 节点，负责从元组中提取出页面的 URL 和访问者形成新的元组，并发送给 Uniquer；Uniquer 也是 Bolt 节点，负责统计访问页面的用户集。


**图 6　UniqueVisitor 的 Topology 的结构图**

### 3.3　验证瓶颈识别算法

实验中为每个 benchmark 随机配置了 10 ~ 12 组参数，共 32 组。本文仅选出几个有代表性的参数配置进行展示，如表 4 所示。

**表 4　参数配置**

| | 配置 1 | 配置 2 | 配置 3 | 配置 4 | 配置 5 |
|---|---|---|---|---|---|
| | | | SOL | | |
| Worker 进程数 | 7 | 2 | 10 | 7 | 12 |
| Spout(Executor 数\|Task 数) | 3 \| 3 | 1 \| 1 | 2 \| 4 | 3 \| 3 | 3 \| 3 |
| Bolt1(Executor 数\|Task 数) | 1 \| 1 | 1 \| 1 | 3 \| 3 | 1 \| 5 | 5 \| 5 |
| Bolt2(Executor 数\|Task 数) | 2 \| 2 | 1 \| 1 | 3 \| 4 | 1 \| 3 | 2 \| 3 |
| Bolt3(Executor 数\|Task 数) | 1 \| 4 | 1 \| 1 | 5 \| 5 | 3 \| 4 | 1 \| 2 |
| Bolt4(Executor 数\|Task 数) | 2 \| 5 | 1 \| 1 | 1 \| 1 | 2 \| 2 | 1 \| 1 |
| Bolt5(Executor 数\|Task 数) | 1 \| 1 | 1 \| 1 | 1 \| 1 | 2 \| 2 | 4 \| 4 |
| | | | WordCount | | |
| Worker 进程数 | 3 | 7 | 13 | - | - |
| Spout(Executor 数\|Task 数) | 1 \| 1 | 1 \| 1 | 1 \| 1 | - | - |
| Split(Executor 数\|Task 数) | 1 \| 1 | 5 \| 5 | 6 \| 6 | - | - |
| Count(Executor 数\|Task 数) | 1 \| 1 | 2 \| 2 | 7 \| 8 | - | - |
| | | | UniqueVisitor | | |
| Worker 进程数 | 5 | 3 | 6 | 9 | - |
| Spout(Executor 数\|Task 数) | 3 \| 3 | 1 \| 1 | 3 \| 4 | 3 \| 3 | - |
| View(Executor 数\|Task 数) | 1 \| 2 | 1 \| 1 | 6 \| 6 | 5 \| 6 | - |
| Uniquer(Executor 数\|Task 数) | 2 \| 2 | 1 \| 1 | 1 \| 3 | 1 \| 1 | - |

对上述每组配置，都采用瓶颈识别算法对分析器收集的性能统计信息进行分析，识别结果如表 5 所示。

下面将分别绘制元组的队列等待时间曲线验证结果的正确性。

#### 3.3.1　SOL

把表 5 中 SOL 的各组配置的瓶颈分析结果对应队列的等待时间绘制为图 7。

**表 5　对于表 4 配置的识别结果**

| Benchmark | 配置编号 | 瓶颈分析结果（哪些队列阻塞） |
|---|---|---|
| SOL | 1 | Bolt1 的接收队列 |
| | 2 | Spout、Bolt2 和 Bolt4 的传输队列 |
| | 3 | Bolt4 的接收队列 |
| | 4 | Bolt1 的接收队列 |
| | 5 | Bolt2 和 Bolt3 的接收队列 |
| WordCount | 1 | Split 和 Count 的接收队列 |
| | 2 | Count 的接收队列 |
| | 3 | 无 |
| UniqueVisitor | 1 | View 的接收队列 |
| | 2 | View 的接收队列 |
| | 3 | 无 |
| | 4 | Uniquer 的接收队列 |

其中,图 7(a)对应配置 1 的 Bolt1 的接收队列,其等待时间线性上升,说明该队列阻塞 Bolt1 成为瓶颈;图7(b) ~ (d)分别对应配置2的Spout、Bolt2和 Bolt4 的传输队列,基本上呈上升趋势,这说明该配置下 Worker 进程数过少,导致传输队列阻塞;图 7(e)对应配置 3 的 Bolt4 的接收队列,该队列阻塞,Bolt4 是瓶颈;图 7(f)对应配置 4 的 Bolt1 的接收队列,图 7(g)和(h)对应配置 5 的 Bolt2、Bolt3 的接收队列,它们都是阻塞的,因而对应的 Bolt 为瓶颈。对比配置 4 和配置 5 的 Bolt1 的参数配置,可以得知,影响 Bolt 执行速度的主要参数是 Executor 数而不是 Task 数。

### 3.3.2　WordCount

对表 5 中 WordCount 的各组配置的瓶颈分析结果,将其对应的队列等待时间绘制为图 8。

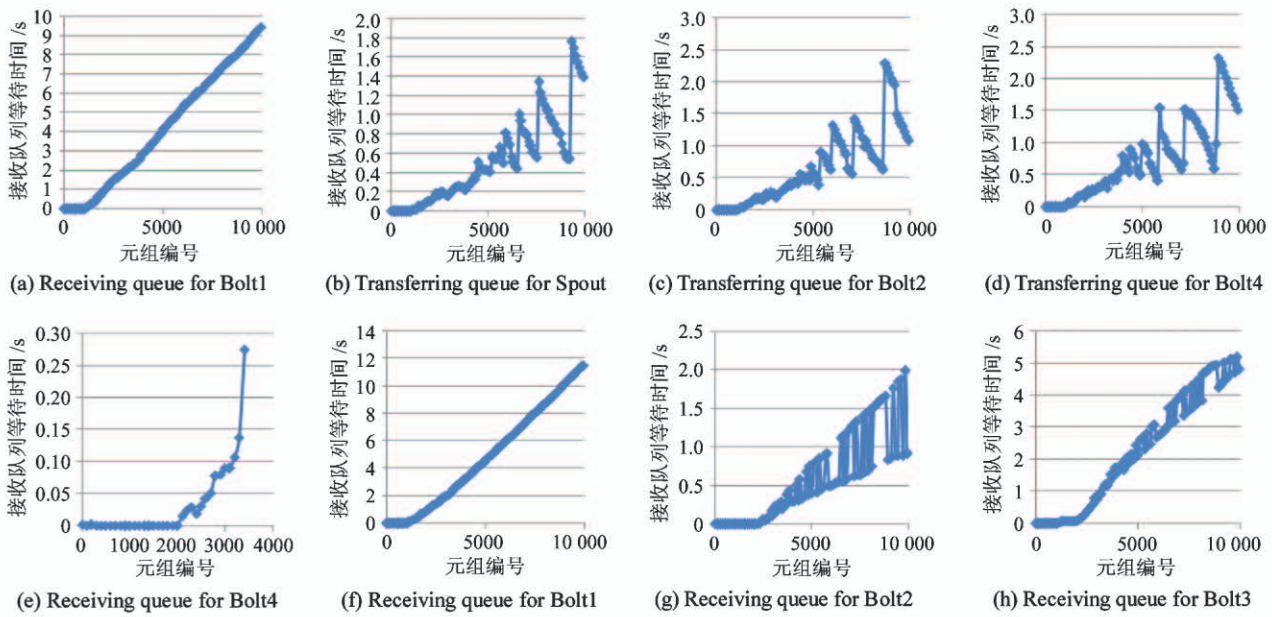



**图 7　表 5 中 SOL 所有配置的瓶颈队列等待时间**

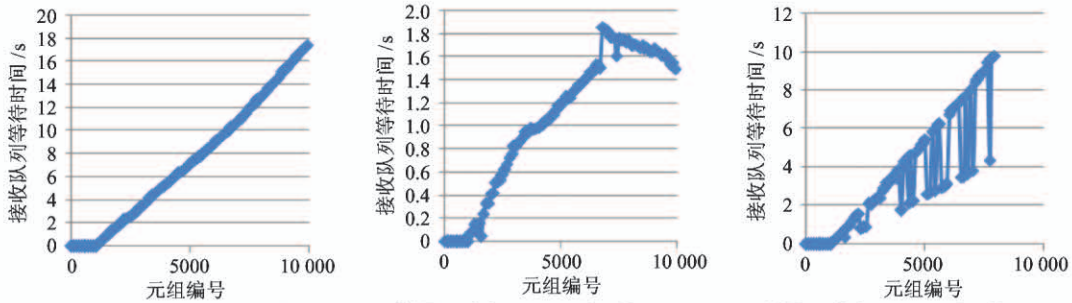
其中,图 8(a)和(b)分别对应配置 1 的 Split、Count 的接收队列,其等待时间有上升趋势,说明这些队列阻塞,对应的 Bolt 成为瓶颈;图 8(c)对应配置 2 的 Count 的接收队列,该队列阻塞,Count 是瓶颈。

### 3.3.3　UniqueVisitor

对表 5 中 UniqueVisitor 的各组配置的瓶颈分析结果,将其对应的队列等待时间绘制为图 9。

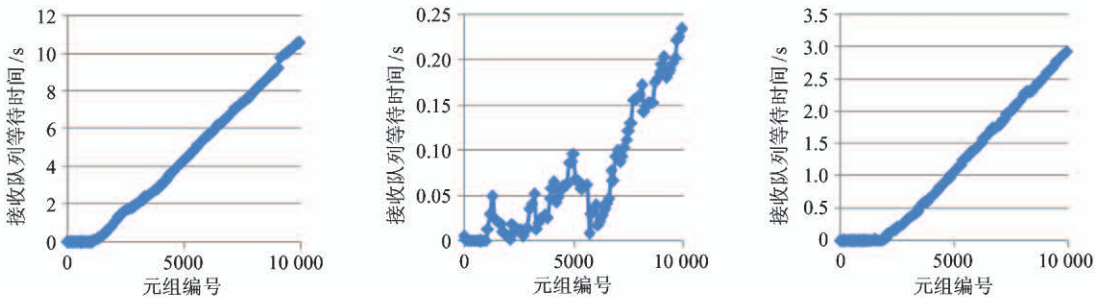
其中,图 9(a)对应配置 1 的 View 的接收队列,其等待时间线性上升,因而队列处于阻塞状态,View 是瓶颈;图 9(b)对应配置 2 的 View 的接收队列,其等待时间有上升趋势,说明该队列阻塞,对应的 View 成为瓶颈;图 9(c)对应配置 4 的 Uniquer 的接收队列,该队列阻塞,Uniquer 是瓶颈。

本实验所有配置的验证情况如表 6 所示。




(a) Receiving queue for Split　(b) Receiving queue for Count　(c) Receiving queue for Count

**图 8　表 5 中 WordCount 所有配置的瓶颈队列等待时间**




(a) Receiving queue for View　(b) Receiving queue for View　(c) Receiving queue for Uniquer

**图 9　表 5 中 UniqueVisitor 所有配置的瓶颈队列等待时间**

**表 6　实验中所有配置的验证情况**

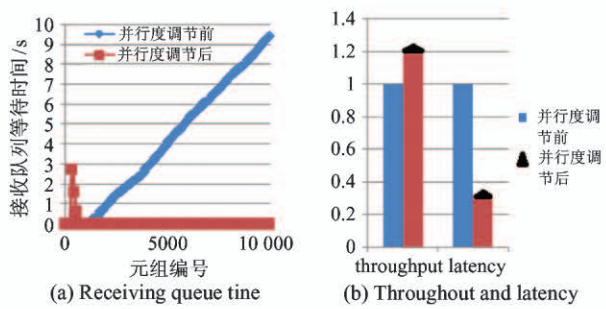
| benchmark | 配置数 | 瓶颈识别成功数 |
|---|---|---|
| SOL | 12 | 12 |
| WordCount | 10 | 10 |
| UniqueVisitor | 10 | 10 |

通过实验数据可知，本文提出的瓶颈识别算法可以有效地指出 Topology 的瓶颈所在。

### 3.4　验证瓶颈缓解算法

本小节以表 5 的 SOL、WordCount 和 UniqueVIsitor 的配置 1 为例，分别缓解它们的 Bolt1、Split 和 Uniquer 的接收队列的阻塞状态，来验证瓶颈缓解算法的有效性；同时，通过迭代使用瓶颈缓解算法，使得配置 1 的所有 Bolt 的瓶颈状态都被缓解，然后比较 Topology 运行过程中的 60 ~ 120 s 之间总吞吐率和总延迟的变化。为了避免传输队列阻塞，在提高并行度（即 Executor 数）的同时，将 Worker 进程数也增加。

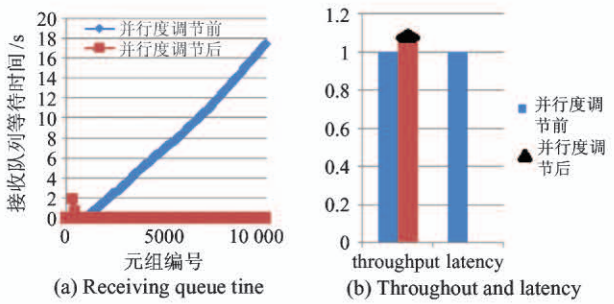
对于 SOL 配置 1，采用 2.3 节的算法，计算出 Spout 向 Bolt1 发射元组的时间间隔的代表值为 0.36，Bolt1 的执行时间的代表值为 0.61，由于 Spout 并行度为 3，所以 Bolt1 的并行度应设置为 5（3 × 0.61 ÷ 0.36）。调节配置 1 的参数，让 Bolt1 的 Executor 和 Task 数提高为 5，同时修改 Worker 进程数为 14，重新运行 SOL，得到 Bolt1 接收队列等待时间如图 10（a）所示，图中 2 条曲线分别代表并行度调节前和调节后的时间。对比图中 2 条线可知，该队列的阻塞状态被缓解。同时，迭代执行 2.3 节的算法，将所有 Bolt 的瓶颈状态都缓解，得到 60 ~ 120 s的总吞吐和总延迟见图 10（b）所示。从图中可以看到，吞吐率变化不大，而并行度调节后的总延迟降低到了原来的 1/3。




(a) Receiving queue tine　(b) Throughout and latency

**图 10　对于 SOL 的瓶颈缓解算法的验证结果**

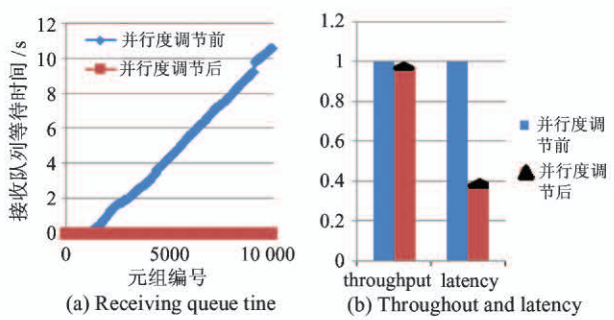
对于 WordCount 配置 1,采用 2.3 节的算法,计算出 Spout 向 Split 发射元组的时间间隔的代表值为 0.33,Split 的执行时间的代表值为 3.2,所以 Bolt1 的并行度应设置为 10(3.2 ÷0.33)。通过调节配置 1 的参数,让 Split 的 Executor 和 Task 数提高为 10,同时修改 Worker 进程数为 12,重新运行 WordCount,得到 Split 接收队列等待时间如图 11(a)所示。对比图中 2 条线可知,该队列的阻塞状态被缓解。同时,迭代执行 2.3 节的算法,将所有 Bolt 的瓶颈状态都缓解,得到 60 ~ 120 s 的总吞吐和总延迟如图 11(b)所示,从图中可以看到,吞吐率变化不大,而并行度调节后的总延迟不到原来的 1/10。



(a) Receiving queue tine　(b) Throughout and latency

**图 11　对于 WordCount 的瓶颈缓解算法的验证结果**

对于 UniqueVisitor 配置 1,同样采用 2.3 节的算法,计算出 Spout 向 View 发射元组的时间间隔的代表值为 0.37,View 的执行时间的代表值为 0.65,由于 Spout 并行度为 3,所以 View 的并行度应设置为 5(3 ×0.65 ÷0.37)。调节配置 1 的参数,让 View 的 Executor 和 Task 数提高为 5,同时修改 Worker 进程数为 10,重新运行 UniqueVisitor,得到 View 接收队列等待时间如图 12(a)所示。对比图中两条线可知,该队列的阻塞状态被缓解。同时,迭代执行 2.3 节的算法,将所有 Bolt 的瓶颈状态都缓解,得到 60 ~ 120 s 的总吞吐和总延迟见图 12(b)所示,从图中可以看到,吞吐率变化不大,而并行度调节后的总延迟约为原来的 1/3。



(a) Receiving queue tine　(b) Throughout and latency

**图 12　对于 UniqueVisitor 的瓶颈缓解算法的验证结果**

## 4　结　论

本文对流计算引擎 Storm 的应用执行效率优化问题进行了研究,提出了以元组为中心的细粒度执行效率优化方法,解决了该引擎因统计信息粒度粗、无法准确定位性能瓶颈从而造成执行效率难以提高的问题。为验证本文提出方法的有效性,采用主流测试基准 Intel Storm benchmark 进行了测试,实验结果表明,在 3 个标准程序 32 个不同的配置参数下,该方法识别瓶颈的准确率为 100%,消除性能瓶颈后的执行效率提高了 70%。值得注意的是,执行效率还可进一步考虑吞吐率(每秒完成的 Tuple 个数)。吞吐率受 Worker 进程数量、Spout 并行度的影响。实验表明,吞吐率随 Worker 进程数的增大先上升后下降,本文对 Worker 进程数仅是简单调节,Spout 并行度决定最终可能达到的吞吐率上限。吞吐率的优化将作为未来的研究方向。

**参考文献**

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

[2] 程学旗,靳小龙,王元卓,等. 大数据系统和分析技术综述[J]. 软件学报,2014,25(9):1889-1908

[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[J]. 软件学报,2014,25(4):839-862

[4] 崔星灿,禹晓辉,刘洋,等. 分布式流处理技术综述[J]. 计算机研究与发展,2015,52(2):318-332

[5] Toshniwal A, Taneja S, Shukla A, et al. Storm@ twitter [C]// Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, New York, USA, 2014: 147-156

[6] The Apache Software Foundation. Apache Storm downloads [EB/OL]. http://storm. apache. org/downloads. html: the Apache Software Foundation, 2017

[7] Kulkarni S, Bhagat N, Fu M, et al. Twitter Heron: stream processing at scale[C]// Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, New York, USA, 2015: 239-250

[8] Zaharia M, Das T, Li H, et al. Discretized streams: fault-tolerant streaming computation at scale[C]//Proceedings of the 24th ACM Symposium on Operating Systems Principles, Farmington, USA, 2013: 423-438

[9] Akidau T, Schmidt E, Whittle S, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing[J]. *Proceedings of the VLDB Endowment*, 2015, 8(12): 1792-1803

[10] The Apache Software Foundation. Apache flink? — stateful computations over data streams[EB/OL]. https://flink. apache. org/: the Apache Software Foundation, 2017

[11] Lu R, Wu G, Xie B, et al. Stream Bench: towards benchmarking modern distributed stream computing frameworks[C]//Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, London, UK, 2014: 69-78

[12] Chintapalli S, Dagit D, Evans B, et al. Benchmarking streaming computation engines: storm, flink and spark streaming[C]//Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Orlando, USA, 2016: 1789-1792

[13] Lopez M A, Lobato A G P, Duarte O C M B. A performance comparison of open-source stream processing platforms[C]//Proceedings of the 2016 IEEE Global Communications Conference, Washington, USA, 2016: 1-6

[14] Fan J, Chen H, Hu F. Adaptive task scheduling in storm[C]//Proceedings of the 4th International Conference on Computer Science and Network Technology, Beijing, China, 2015: 309-314

[15] Shukla A, Simmhan Y. Benchmarking distributed stream processing platforms for IoT applications[C]//Proceedings of Technology Conference on Performance Evaluation and Benchmarking, New Delhi, India, 2016: 90-106

[16] Chintapalli S, Dagit D, Evans R, et al. PaceMaker: when zookeeper arteries get clogged in storm clusters[C]//Proceedings of IEEE 9th International Conference on Cloud Computing, Hong Kong, China, 2016: 448-455

[17] GitHub Inc. Storm benchmark[EB/OL]. https://github. com/intel-hadoop/storm-benchmark: GitHub, 2017

[18] Intel Corporation. Intel turbo boost technology[EB/OL]. https://www. intel. com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology. html: Intel Corporation, 2017

[19] Apache Software Foundation. APACHE Kafka a distributed streaming platform[EB/OL]. http://kafka. apache. org/: Apache Software Foundation, 2017

[20] Kreps J, Narkhede N, Rao J. Kafka: a distributed messaging system for log processing[C]//Proceedings of the 6th International Workshop on Networking Meets Databases, Linköping, Sweden, 2011: 1-7

# Fine-grained optimization method for execution efficiency of stream computing

Wei Bo[*,**,***], Liu Xiaodong[*], Yan Yuting[*], Zhao Xiaofang[*], Yuan Yuxin[*,***], Tang Hongwei[*]

([*] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**] The First Research Institute of Ministry of Public Security, Beijing 100044)

([***] University of Chinese Academy of Sciences, Beijing 100049)

**Abstract**

The stream computing engine Storm is widely used in the real-time processing of big data to improve the execution efficiency of data processing. However, due to the coarse granularity of statistical information, the engine cannot accurately locate the performance bottleneck, which makes it a big problem to improve the execution efficiency of data processing. To solve the problem, a fine-grained optimization method for execution efficiency of stream computing is proposed, which focuses on tuples for fine-grained performance analysis, including performance bottleneck identification algorithm and performance bottleneck mitigation algorithm, and supports quantitative selection of optimal parameter configuration to improve execution efficiency. The experimental results show that under 32 different configuration scenarios of three standard programs, the proposed method can accurately identify the performance bottleneck of flow applications, the recognition rate is 100%, and the application execution efficiency can be improved by 70%.

**Key words**: stream computing engine, optimization method of execution efficiency, fine-grained, tuple