Spark框架并行度推断算法

卞 琛[1,2*]，于 炯[2]，修位蓉[2]，廖 彬[3]，英昌甜[2]，鲁 亮[2]

(1. 广东金融学院互联网金融与信息工程学院 广州 510521；
2. 新疆大学信息科学与工程学院 乌鲁木齐 830046；3. 新疆财经大学统计与信息学院 乌鲁木齐 830012)

【摘要】分布式计算集群Spark宽依赖并行度取决于用户设定参数，对于不同的作业类型或数据集，硬编码的并行度参数设定难以发挥集群的最大计算能效。针对这一问题，首先对Spark作业执行方式进行深入分析，建立作业调度模型，提出宽依赖计算代价、资源空置率和溢写概率的定义；然后分析任务并行度对作业执行时间的影响，证明并行度取值具有合理区间，提出并行度推断算法的优化目标。最后根据模型定义进行目标求解，设计批处理内存计算框架的并行度推断算法(parallelism deduction algorithm, PDA)，通过构建的数据总量、执行区预留比、操作闭包集合、资源表等多个基础数据，计算符合资源需求表且具有最大资源利用率和最小开销的任务并行度；PDA算法在作业的各个Stage中迭代执行，根据计算环境优化调度方案提高性能。实验表明，PDA算法提高了Spark框架的作业执行效率，针对不同类型作业均具有良好的普适性。

关 键 词 内存计算；并行度推断；性能优化；Spark；溢写概率
中图分类号 TP393.09 **文献标志码** A doi:10.3969/j.issn.1001-0548.2019.04.014

Parallelism Deduction Algorithm for Spark

BIAN Chen[1,2*], YU Jiong[2], XIU Wei-rong[2], LIAO Bin[3], YING Chang-tian[2], and LU Liang[2]

(1. College of Internet Finance and Information Engineering, Guangdong University of Finance Guangzhou 510521;
2. College of Information Science and Engineering, Xinjiang University Urumqi 830046;
3. College of Statistics and Information, Xinjiang University of Finance and Economics Urumqi 830012)

Abstract Inappropriate parallelism parameter may result in the performance degradation on in-memory computing framework. For this issue, we analyze the execution mechanism of Spark jobs, establish job scheduling model, and give the definition of the computing cost, resource idle rate and spill probability. Based on the analysis of the relationship between parallelism parameter and job execution efficiency, the optimization objective of algorithm is given. To solve the problem of optimizing, a parallelism deduction algorithm (PDA) for in-memory computing framework is proposed. Firstly, PDA calculates the best parallelism of job execution by size of input data, worker computing resource and additional overhead of spill, and thus enhances the resource utilization of cluster and speeds up the state synchronization of job execution. The algorithm optimizes the task scheduling for each Stage, accelerates the job execution and improves the calculation efficiency. Experiment results demonstrate that the proposed algorithm can improve the computational efficiency of in-memory computing framework and accelerate data-intensive and compute-intensive applications.

Key words in-memory computing; parallelism deduction; performance optimization; Spark; spill probability

为进一步提高内存计算框架的数据处理效率和集群资源利用率，优化作业执行性能，本文选取开源内存计算框架Spark[1]为研究对象。Spark中宽依赖Stage的任务并行度取决于用户设定参数，系统读取并行度参数后发送相应的任务到worker执行，因此用户参数设定的合理性将直接决定作业执行效率和计算时长[2]。然而，程序员仅能通过经验设定并行度参数，难以最大化发挥计算集群的计算能力，因为注册worker数量、可用资源量、作业类型和数据分布等因素都将影响并行度合理性。即使优秀的程序员能够将并行度参数定位到最优区间，也不得不为不同的作业频繁修改参数，重复性工作既无法保证精准度，也降低了生产效率。为解决以上问题，本文主要工作有以下3个方面：

收稿日期：2017－04－24；修回日期：2019－03－19
基金项目：新疆维吾尔自治区自然科学基金(2017D01A20)
作者简介：卞琛(19 81－)，男，博士，副教授，主要从事内存计算、分布式系统、边缘计算等方面的研究. E-mail: bianchen0720@126.com

1) 首先分析Spark框架作业宽、窄依赖Stage划分及执行过程，建立模型对作业计算代价、资源利用率、作业执行附加开销等指标进行评估。

2) 在作业调度模型的基础上，对并行度推断算法进行问题定义，为后期算法设计确定优化目标。

3) 划分问题的已知域与未知域，通过已知域构建算法基础数据，求解未知域提出并行度推断算法，并分析算法与优化目标的符合度。

1 相关工作

高效率、低延迟的内存计算框架得到工业界和学术界的一致认可，越来越多的企业级应用采用Spark作为底层框架，而研究人员也针对Spark提出了不同的性能优化方法。

内存计算框架的最大特点就是充分利用内存来提高计算效率，一些研究人员在提高Spark框架内存利用率的问题上提出了自己的优化策略。文献[3]提出Tungsten系统，由于Spark框架运行于JVM(java virtual machine)之上，而java对象需要占用源数据近2倍的内存空间，因此Tungsten提出数据堆外存储方式，避免java对象的空间浪费，提高内存空间利用率。此外，堆外存储直接存储二进制对象，无须序列化与反序列化的开销，还能够有效避免JVM的GC(garbage collection)回收效率问题。文献[4]提出内存分区动态划分方法，将Spark内存计算区与缓存区的固定比例划分(通过用户参数设定)修改为模糊边界的动态划分，对不同的作业对计算区与缓存区的空间需求量予以最大保障，从而加速作业执行。文献[5]提出Cache-friendly算法，建立三级存储的协调管理架构，提高内存数据命中率，有效避免缺页开销。文献[6]提出内存文件系统Alluxio(原名Tachyon)，将内存的存储功能计算框架独立出来，通过更细的分工达到更高的效率。Alluxio的兼容性更好，对上层计算框架的支持度也更高。

另外一些研究人员关注内存计算框架的性能优化方法，文献[7]提出Catalyst查询优化器，其目标是生成更高效的逻辑执行计划，Catalyst分析用户定义的所有操作，采用基于关系代数的等价变换方法，将高开销的操作替换为低成本操作，并在不影响作业执行结果的前提下调整操作执行顺序，从而生成更为高效的执行计划。文献[8]发现Shuffle过程是Spark框架的重要瓶颈，因此将Shuffle从作业执行中分离形成独立过程，开发优化I/O且更轻量级的服务组件，降低作业中的Shuffle过程开销。文献[9]同样关注Shuffle过程优化，将多对多的网络数据传输方式由推模式修改为拉取模式，优化调度器的负载均衡，使作业的整体执行时间缩短了20%。

本文从批处理作业执行机制入手，探索并行度与作业执行效率的关系，与已有研究工作的最大不同之处在于，建立作业调度模型分析硬编码并行度产生效率瓶颈的原因，研究符合集群硬件资源并具有普适性的自动化并行度推断算法，将多个stage使用相同并行度参数优化为每个阶段独立生成最优并行度，从而优化集群计算效率、缩减作业执行时长。

2 问题的建模与分析

本节首先建立作业调度模型，分析任务并行度与作业执行效率的关系，然后提出算法的优化目标。

2.1 作业调度模型

用户将作业提交至Spark系统后，调度器DAGScheduler生成作业的有向无环图，同时根据DAG的宽窄依赖关系划分Stage，交由TaskScheduler分配任务至各个Executor，各个Stage顺序执行。用户编程使用的基础数据结构为RDD (resilient distributed datasets)、Dataframe或Dataset，而DAGScheduler和TaskSheduler实际调度的粒度均为分区。设作业使用HDFS中文件作为计算源数据，文件在HDFS中占据4个Block，用户在Spark系统设置parallelism参数值为h，那么该作业的执行流程图可以表述为图1。

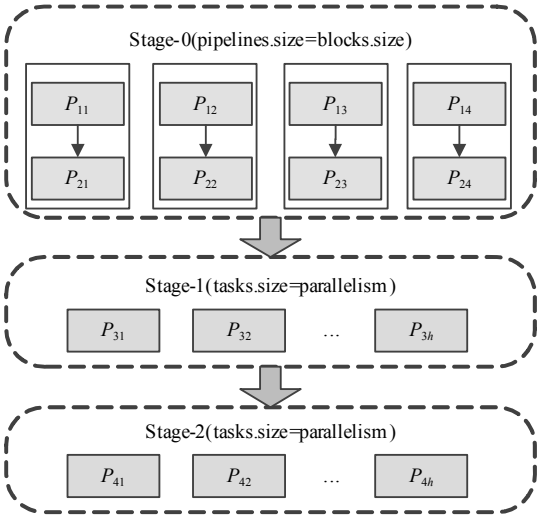


图1 Spark作业执行流程图

由图1可以看出，由于在HDFS中源数据占据4个Block，因此作业开始即有4个分区，在窄依赖Stage中，流水线的个数初始分区数相同；宽依赖Stage则读取用户设置Parallelism参数，确定每轮执行的任务

总数，分配给集群中的工作节点并行计算。

定义 1 计算代价：由于宽依赖Stage每个分区都需要从上一个RDD的所有分区读取数据，因此分区计算代价为数据拉取代价与数据计算代价构成。设分区$P_{i,h}$输入数据的桶集合为$\text{bks}_{i,h}$，以分区计算时间作为衡量计算代价的指标，则分区$P_{i,h}$的计算代价为：

$$T_{P_{i,h}} = \text{pull}(\text{bks}_{i,h}) + \text{compute}(\text{bks}_{i,h}) \tag{1}$$

任务的即时执行是一个理想化环境，但集群资源有限，若待分配任务数超过资源槽个数，则部分任务必须挂起等待，因此分区的计算代价还应包括等待调度的时间，即：

$$T_{P_{i,h}} = T_{\text{wait}_{i,h}} + \text{pull}(\text{bks}_{i,h}) + \text{compute}(\text{bks}_{i,h}) \tag{2}$$

记宽依赖Stage的计算任务为RDD_i。由于按照计算时间来评估代价，那么当前Stage的计算代价为RDD_i的所有分区计算代价的最大值，即：

$$T_{\text{stage}} = T_{\text{RDD}_i} = \max(T_{P_{i,1}}, T_{P_{i,2}}, \cdots, T_{P_{i,h}}) \tag{3}$$

定义 2 任务并行度：用于评价某一时刻并行执行的任务数。在Spark框架下，用户可以指定参数确定并行度，并行度参数与宽依赖Stage中RDD的分区个数相同，分区个数又决定了任务数。而由用户所指定的多个任务能否并发执行，还要依赖于Worker的数量及每个Worker上所分配的用于作业计算的CPU核心数。记当前执行作业的工作节点数为m，每个Worker上用于作业计算的CPU核心数为c，那么硬件资源支持的最大作业并发量为mc，称为物理并行度。设计算框架的并行度设置为h，对于h和mc应当小值优先，因此实际并行度可以表示为：

$$\text{dp} = \min(h, mc) \tag{4}$$

定义 3 空置率：用于表示因任务调度的不均衡性引发的硬件资源空闲比。根据定义2，若物理并行度大于用户并行度，表示资源槽个数多于等调度任务数，所有任务仅需1轮即可执行完毕。若物理并行度小于用户并行度，即$mc > h$时，表示资源槽个数小于待调度任务数，即部分任务需要等待，所有任务需要多轮执行才能完成。因此，只要h与mc不相等，则执行轮数即为：

$$\text{Round} = \text{ceiling}\left(\frac{h}{mc}\right) \tag{5}$$

可以看出，当且仅当h与mc为倍数关系时，每轮次都将有c个任务并发执行，所有工作节点获得均匀的任务分配；否则，必有工作节点在最后一轮执行时有资源空闲。因此，当前Stage计算时的资源空置率为：

$$V_{\text{stage}} = \frac{\text{Round} \times (mc) - h}{\text{Round} \times (mc)} \tag{6}$$

通过式(6)可知，仅当$h \bmod (mc) = 0$时，$\text{Round} \times (mc) = h$，而$V_{\text{stage}}$=0。由于$m$和$c$来源于资源需求表，均为常量，而对于用户并行度$h$，满足条件$V_{\text{stage}} \geqslant 0$的概率为：

$$Q = 1 - \frac{1}{mc} \tag{7}$$

若由50个节点执行计算任务，每个节点为2核CPU，则空置率为0.99，因此发生资源空闲的概率较大。

定理 1 宽依赖Stage的计算代价与空置率成正比。

证明 记执行计算的工作节点集合为w={1,2,⋯,m}，由于需要Round轮才能将所有任务执行完毕，当$h \bmod (mc) \neq 0$时，分配到第j($j \in w$)个工作节点上执行的任务数应表示为：

$$\text{Count}_j = \begin{cases} \text{Round} & j \leqslant (\text{h} \bmod (mc)) \\ \text{Round} - 1 & j > (\text{h} \bmod (mc)) \end{cases} \tag{8}$$

记$f(x)$表示单节点上任务执行的代价函数，参数x表示任务数量，那么当前Stage计算代价为：

$$T_{\text{stage}} = \max(f(\text{Count}_1), f(\text{Count}_2), \cdots, f(\text{Count}_j)) = f(\text{Round}) \tag{9}$$

通过式(6)求解Round可得：

$$\text{Round} = \frac{h}{(1 - V_{\text{stage}})mc} \tag{10}$$

式中，h，m，c均为常数，则V_{stage}越大，f(Round)也越大。因此宽依赖Stage计算代价与空置率成正比。

证毕。

由定理1可知，空置率越高则引发计算代价越大，若作业中有多个宽依赖Stage，则计算延迟会不断累积。

定义 4 宽依赖附加开销：用于表示shuffle过程因用户并行度设置引起的作业执行开销。宽依赖Stage的分区数量与用户并行度相同，当前续Stage执行完毕后，将数量划分到h个Bucket中，Bucket的写入工作由与其一一对应的Write Handler进程完成，Write Handler在内存中占用大小固定的缓冲区，记$\text{Cost}_{\text{spill}}$为每个Write Handler的内存占用量。此外，作业调度也会产生开销，记$\text{Cost}_{\text{scheduling}}$为宽依赖Stage中每个任务的调度开销，那么宽依赖附加开销为：

$$\text{Cost}_{\text{extra}} = h(\text{Cost}_{\text{spill}} + \text{Cost}_{\text{scheduling}}) \tag{11}$$

定义 5 溢写概率：用于表示因拉取数据量超越可用内存容量导致溢写磁盘的可能性。Worker拉取的数据将优先存储于内存中，若内存无法存储所有数据，则启动磁盘溢写进程，由于磁盘I/O吞吐量远小于内存，溢写过程将降低作业执行效率，因此需要对溢写概率进行评估，保障后续算法设计的准确度。Spark系统内存执行区的比例由memory.Fraction参数确定，记其为φ(0<φ<1)。r为每个worker用于作业执行的内存容量，那么实际可用于数据拉取的内存容量为$r\varphi$。

设前续RDD有k个分区，则当前Stage每个任务需要拉取k个Bucket的数据，记$\mathrm{bks}_{i,h}$={$\mathrm{bk}_{i,h,1}$,$\mathrm{bk}_{i,h,2}$,…,$\mathrm{bk}_{i,h,k}$}为任务拉取的Bucket集合。若当前Stage每轮执行的c个任务，则避免溢写的必要条件是的输入数据均能放入内存，因此溢写概率可表示为：

$$P_{\mathrm{j}}=\begin{cases}0 & \left(\sum_{x=1}^{k}\mathrm{bks}_{i,h,x}\times c\right)\leqslant r\varphi\\ 1 & \left(\sum_{x=1}^{k}\mathrm{bks}_{i,h,x}\times c\right)>r\varphi\end{cases} \tag{12}$$

由定义4和定义5可以看出，一方面，宽依赖附加开销随并行度线性变化，并行度越大，附加开销也越大。另一方面，若宽依赖Stage各任务的输入数据量基本相同，则前续Stage输出数据量S与宽依赖Stage拉取数据量的关系可表示为：

$$S=\sum_{x=1}^{k}\mathrm{bks}_{i,h,x}\times h \tag{13}$$

由式13可知，前续Stage的输入数据量固定，因此h越小则$\sum_{x=1}^{k}\mathrm{bks}_{i,h,x}$越大，即每个任务的输入数据量越大，从而可能导致溢写概率为1。为避免溢写产生的磁盘I/O，任务并行度取值越大越好。因此，只有将任务并行度指定到合理区间，才能在硬件资源稳定的前提下加速作业执行。

2.2 并行度推断算法的问题定义

根据2.1节的定义和分析，并行度推断算法的优化目标可形式化为：

object

$$\min(V_{\mathrm{stage}}) \tag{14}$$

$$\min(\mathrm{Cost}_{\mathrm{extra}}) \tag{15}$$

s.t.

$$(\forall j\in w)P_j=0 \tag{16}$$

目标是最小化空置率和宽依赖附加开销，约束条件为所有Worker的溢写概率均为0。根据定义3和定义4，并行度的取值将决定空置率和宽依赖调度开销，同时对Worker的溢写概率也有重要影响。

3 并行度推断算法

本节基于模型的定义分析，提出Spark框架并行度推断算法。

3.1 环境定义

环境定义是算法验证准确性的的重要保障，Spark系统2.0版的参数设置共有167种，不同的参数设定将对系统效率产生不同的影响，因此本文方法采用固定的一套配置参数，以保障算法执行环境的一致性。此外，Shuffle过程的分区函数将决定每个宽依赖Stage任务的输入数据量，由于不了解真实的数据分布，数据倾斜将导致宽依赖Stage的执行延时增加，从而无法准确评估任务并行度与作业执行效率的关系。为避免数据倾斜对作业执行时间的影响，本文采用文献[10]提出的closer系统进行数据均衡分配，采用consolidation技术完成数据分区，以减少Bucket写入开销，最大程度减少数据倾斜和分区写入对作业执行时间的影响，从而为最优并行度的精确计算提供保障。

3.2 构建基础数据

基础数据分为系统变量和介入参数2类。系统变量是Spark框架的原生数据，由Spark控制台、资源需求表和配置参数提供，主要包括执行作业的工作节点总数m，每个Worker的CPU核数c和可用内存量r，执行区划分比φ。介入参数是并行度推断算法必需的附加数据，生成方式是在作业执行过程中统计或由用户自行定义。算法的介入参数主要包括：

1) 前续Stage的中间结果数据总量S。数据总量S是对前续RDD所有分区大小进行累加，由于Spark框架中作业的各Stage同步执行，因此具备精确统计的条件。

2) 执行区预留比u。参数u($u\in[0,0.5]$)表示在执行区内存中计算缓冲区所占的比例，是算法唯一依赖的用户参数。定义预留比的主要目的是防止拉取数据溢出，频繁引发GC影响执行效率。由于用户了解执行作业类型和输入数据量，因此能够为执行区预留比指派相对合理的值。需要说明的是，执行区预留比u仅是一个参数，并不进行内存空间划分，对Spark框架原生的内存管理策略不产生任何影响。

3) 操作闭包集合TC。定义二元组tc_i=<trans, closure>表示第i个Stage的操作和闭包。在作业调度时，从DAGScheduler中读取各Stage操作和闭包添加

到TC集合。TC集合的作用是判定作业是否存在迭代，以防止无效的并行度推断。由于大多数机器学习算法都要多轮迭代才能收敛，而每轮迭代的数据量变化不大，因此每轮推断的并行度也基本相同，即使并行度推断算法的时空复杂度较低，也会因多轮迭代的累积效应产生延迟，因此TC集合是并行度算法是否迭代执行的重要依据。

3.3　并行度推断算法

下面对2.2节定义的优化目标进行求解，提出并行度推断算法，以此在宽依赖Stage分配适当任务数量，算法的主要思想如下：

1) 迭代判定。将宽依赖Stage的操作和闭包生成二元组，在TC集合中进行检索，若有匹配项则直接使用上次推断的并行度h，跳转至步骤5)；若无匹配项则执行步骤2)。

2) 读取基础数据。将前续RDD的各分区大小累加求得数据总量S，从Spark控制台读取worker总数m，从资源需求表读取Worker的CPU核数c和内存量r，从系统参数中执行区划分比φ，在用户参数中读取空间预留比u。

3) 计算数据总量S与可用内存量的大小关系，若$S \leqslant (mr\varphi(1-u))$，则只需1轮任务分配即可计算完成，生成并行度$h=mc$，跳转至步骤5)。否则执行步骤4)。

4) 计算任务执行轮数，将数据总量S与可用内存量$(mr\varphi(1-u))$相除，取商并向上取整，得到任务执行轮数Round，生成并行度$h=mc\times\text{Round}$。

5) 任务调度。将h个任务分配至所有工作节点，算法结束。

PDA算法的执行过程并行度推断算法：

输入：工作节点个数m
　　单节点分配的核心数c
　　单节点的内存分配量r
　　内存划分比φ
　　空间预留比u
　　操作闭包集合TC
输出：任务并行度h
　　tc_i=getCurrentTrans(); /*获取当前Stage的操作和闭包生成二元组 */
　　left=TC.findLeftNeighbor(tc_i); /*查找与tc_i相同的最近左邻居*/
　　rigth=TC.findRightNeighbor(tc_i); /*查找与tc_i相同的最近右邻居*/
　　if TC.sub(left, tc_i)==TC.sub(tc_i,right) then /*如果TC从left取到tc_i的子集与从tc_i取到right的子集相同，判定为迭代*/
　　return; /*沿用已有的任务并行度*/
　　else
　　S=gatherInput();/*计算输入数据总量*/
　　if $S<=mr\varphi(1-u)$ then /*若数据总量小于等于可用拉取内存总量*/
　　$h=mc$; /*任务并行度等于工作节点数乘以CPU核心数*/
　　return h;
　　else
　　round=ceiling(S/($mr\varphi(1-u)$)); /*计算任务执行轮数*/
　　h= mc×round; /*任务并行度等于工作节点数与核心数、执行轮数的乘积*/
　　return h;
　　end if
　　end if

定理 2　算法求解的并行度h符合2.2节定义的优化目标。

证明：首先证明算法使作业的溢写概率为零，即$(\forall j \in w)P_j = 0$。由于使用closer系统进行数据均衡，则每个任务的输入数据量为：

$$\text{input}_{\text{task}_i} = \frac{S}{h} \tag{17}$$

所有输入数据均从前续Stage填充的Bucket拉取，因此$\text{input}_{\text{task}_i}$为当前Stage需要拉取的Bucket总容量。由算法步骤3)和步骤4)的执行过程可知：

$$h \geqslant \frac{S}{r\varphi(1-u)}c \tag{18}$$

将式(17)代入式(18)可得：

$$\text{input}_{\text{task}_i} \leqslant \frac{r\varphi(1-u)}{c} \tag{19}$$

由于$u \in [0, 0.5]$，因此可得符合$(\forall j \in w)P_j = 0$的表达式：

$$(\text{input}_{\text{task}_i} \times c) \leqslant r\varphi \tag{20}$$

对于优化目标1，算法步骤3)步骤4)所计算的并行度h均为$h=mc\times\text{Round}$(步骤3)的执行轮数Round=1)，因此集群的空置率为零，即V_{stage}=0。而对于优化目标2，步骤4)求解的执行轮数为大于等于数据总量与可用内存量相除所得商的最小整数，因此算法求解的并行度h为符合约束条件的最小值，附加开销也最小。证毕。

在Spark原生系统中，并行度由用户根据经验设

置，经验差异导致的性能差异不可规避。而通过引入并行度推断算法，提高的并行度设定的准确度和适应性，消除了由并行度设定误差导致的延时累积，提高了作业执行效率。PDA算法使作业中多个宽依赖Stage的并行度可以不同，每个Stage都由推断算法生成最优并行度，从而有效地提高计算性能。此外，算法全部过程均为简单算术运算且仅执行常数次，因此时间复杂度为$O(1)$。数据总量S的统计汇总是算法中唯一具有延时的操作，而根据TC集合所做的迭代判定，能够大大减少数据总量的统计次数，从而将算法的负面影响降至最低。

4 实验与评价

本节通过实验进行测试和评价，验证Spark框架并行度推断算法的有效性和普适性。

4.1 实验环境

实验集群由1台服务器和8个工作节点组成，服务器承担Spark的master和Hadoop的NameNode。实验通过Spark控制台采集任务执行时间，nmon完成资源利用率监测。

实验数据有两类，生成的Zipf测试数据集和SNAP(Stanford network analysis project)[11]真实数据集。Zipf测试数据集用于完成WordCount测试，总数据量为8.6 GB，包括9个子集，每个子集的Zipf分布指数为γ，γ取0.2～1.0之间的小数，增量为0.1，分布指数越大表示数据分布越倾斜。SNAP数据集为无向图，用于执行k-means和PageRank作业，具体信息如表1所示。

表1 测试数据集列表

数据集	作业	节点数	边数
Facebook Social Network	Facebook	4 039	88 234
web-Google	Google	875 713	5 105 039
Cit-Patents	Cit-Pts	3 774 768	16 518 948

4.2 并行度测试

实验首先选择Zipf的4个子集进行WordCount测试，观测不同并行度参数设定下的作业执行时间，验证并行度与计算性能的关系，证明通过并行度推断优化作业执行效率的合理性，实验结果如图2所示。

由图2可以看出，随着任务并行度的增大，作业执行时间不断减少，在并行度取值为32时趋于稳定，后期又有略微增加。由实验结果可以证实，一方面，并行度设置值是影响作业执行性能的一个重要因素，错误的并行度将导致延时增大，降低集群计算效率。另一方面，并行度具有稳定的收敛区间，并非越大越好。以上两点表明，设计推断算法求解并行度最优值的问题切入点合理，具备算法设计的必要性和可行性。

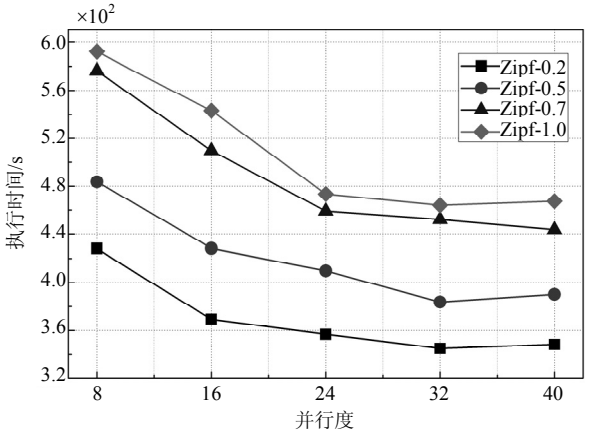
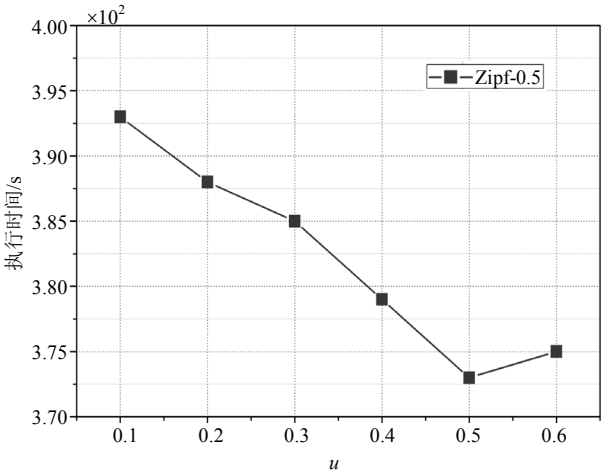
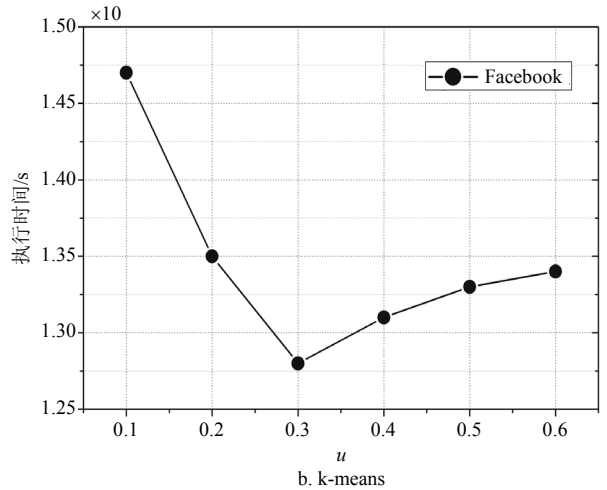


图2 传统Spark的作业执行效率

4.3 参数评估实验



a. WordCount



b. k-means

图3 空间预留比取值测试

空间预留比u是并行度推断算法中唯一一个用户自定义参数，本节重点验证空间预留比的取值对并行度推断算法准确度的影响。实验选取数据密集

型的WordCount和计算密集性的*k*-means进行测试，WordCount的测试数据集是Zipf-0.5，*k*-means则使用Facebook数据集，迭代次数为10。在多次实验中，空间预留比*u*在区间[0.1,0.6]以0.1为步长递增，采集数据为作业执行时间，实验结果如图3所示。

由图3a可以看出，当*u*取值范围在0.1～0.4时，WordCount作业的执行时间随*u*取值的增大而递减，当*u*取值为0.6时计算时长又略有增加。由于WordCount为数据密集型作业，硬件资源特别是内存空间的匮乏是性能的主要瓶颈，因此空间预留比应取较大值。由图3b可以看出，空间预留比*u*以0.3为临界值，作业执行时间在临界值前后出现明显先减后加的趋势。*k*-means是计算密集型作业，对内存容量的敏感度不高，因此空间预留比不应设置较大。

综合比较来看，对不同的空间预留比取值，两种类型作业的执行时间变化率都很小。主要原因有以下两个方面：1) 空间预留比*u*仅是一个参数，既不做内存空间硬性划分，也不改动Spark原生内存管理策略，设计这一参数的唯一目的是为了降低溢写概率和GC开销；2) 即使空间预留比设置不合理，也只会影响调度任务总数，资源空置率始终衡定为零，因此空间预留比对计算性能的影响很小。由整体比较结果可以看出，对不同类型的作业，空间预留比*u*的最优取值不相同，不能硬编码直接指定，需要用户根据具体情况自行选择。

4.4　对比实验

本节重点测试并行度推断算法的有效性，实验采用WordCount和PageRank算法进行测试，由于两种作业均属数据密集型，空间预留比默认为0.5。

1) WordCount

实验选择5个不同分布指数的Zipf数据集执行WordCount作业，为避免数据倾斜影响算法的真实性判断，实验中加入closer系统数据均衡后的测试结果。Spark原生系统和closer的并行度为10，实验结果如图4所示。

由图4可以看出，不论测试数据集为何种分布，closer系统数据均衡的效果均优于原生Spark，再通过PDA算法的介入，作业执行有了一定程度的提升，从而证明了并行度推断算法的有效性。总体来看，PDA具有良好优化效果的原因有两个方面：1) PDA使任务数量与Reducer总数更加匹配，从而提高Worker的工作参与度，优化作业执行性能；2) 由于推断过程以减少内存溢出为目标，避免频繁GC和溢写开销，降低延时缩减作业执行时间。

2) PageRank

实验选取2个无向图数据集执行PageRank作业，以测试并行度推断算法的正确性。为避免数据倾斜的影响，Spark原生系统和PDA均采用closer数据均衡算法，实验结果如图5所示。

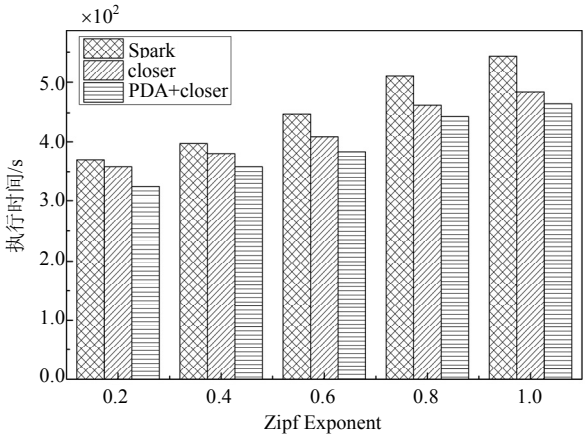


图4　WordCount对比实验

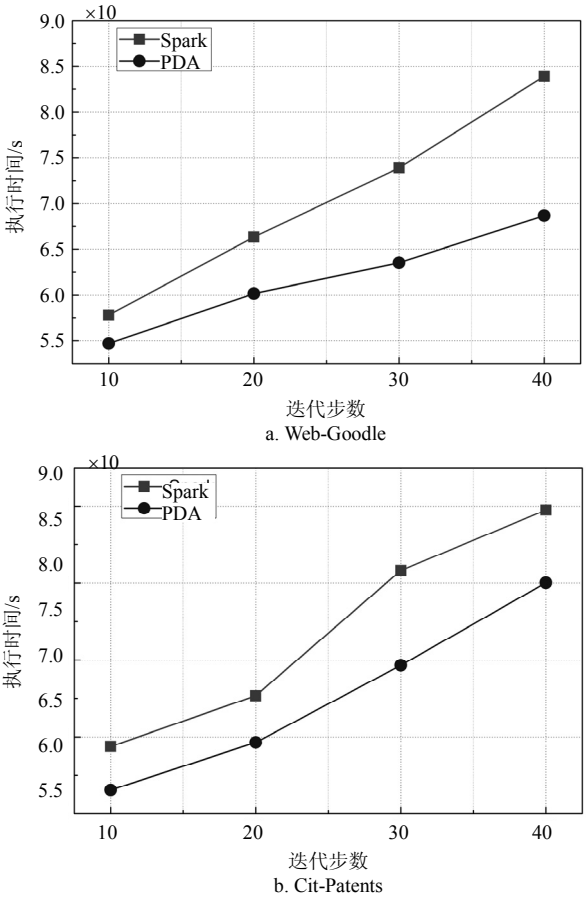


图5　PageRank对比实验

由图5可以看出，对于每一个数据集，PDA算法的作业执行时间均明显小于Spark原生系统，从而证明PDA算法对Spark框架具有性能优化效果。对于2个数据集，作业执行时间随着迭代次数的增加而线

性减少，这是因为PDA算法执行迭代判定，每轮迭代的优化比例也基本相同。从图5的整体对比结果来看，对于多轮迭代的机器学习算法，PDA可以根据输入数据量、可用内存和Worker总量等参数进行任务并行度的动态调整，提高节点参与度和资源利用率，优化作业执行效率，由此证明了并行度推断算法的有效性。

5 结 束 语

本文针对批处理内存计算框架中任务并行度的合理性问题，首先分析Spark框架作业宽、窄依赖Stage划分及执行过程，对作业计算代价、资源利用率、作业执行附加开销等指标进行评估。然后在模型定义和证明的基础上，提出并行度推断算法的优化目标。最后通过目标定义，构建算法基础数据，提出并行度推断算法，并证明了算法与优化目标的匹配度。

下一步工作集中在以下两个方面：

1) 对连续窄依赖的流水线机制进行分析，探索具有最小同步代价的作业执行机制。

2) 对宽窄依赖之间的Shuffle过程进行分析，设计异构环境下适应节点计算能力的Shuffle策略。

参 考 文 献

[1] ZAHARIA M, CHOWDHURY M, DAS T, et aI. Fast and interactive analytics over hadoop data with spark[J]. USENIX Login, 2012, 37(4): 45-51.

[2] ZAHARIA M, XIN R, WENDELL P, et al. Apache Spark: A unified engine for big data processing[J]. Communications of ACM, 2016, 59(11): 56-65.

[3] XIN R, ROSEN J. Project Tungsten: Bringing Apache Spark closer to bare metal[EB/OL]. [2016-03-21]. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[4] WENDELL P, ZAHARIA M, XIN R. Spark memory management overview[EB/OL]. [2016-05-21]. http://spark.apache.org/docs/latest/tuning.html#memory-management-overview.

[5] SENGUPTA B, DAS A. Use of SIMD-based data parallelism to speed up sieving in integer-factoring algorithms[J]. Applied Mathematics and Computation, 2017, 293(1): 204-217.

[6] LI Hao-yuan, GHODSI A, ZAHARIA M, et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks[C]//The 2014 ACM Symposium on Cloud Computing. New York: ACM, 2014: 1-15.

[7] ARMBRUST M, XIN R S, LIAN C, et al. Spark SQL: Relational data processing in spark[C]//The 2015 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2015: 1383-1394.

[8] LI Jin-gui, LIN Xue-lian, CUI Xiao-long, et al. Improving the shuffle of Hadoop mapreduce[C]//The 13th IEEE International Conference on Cloud Computing Technology and Science. Piscataway, NJ: IEEE, 2013: 266-273.

[9] GUO Yan-fei, RAO Jia, ZHOU Xiao-bo. IShuffle: Improving Hadoop performance with shuffle-on-write[C]//The 10th International Conference on Autonomic Computing. Berkley, CA: USENIX Association, 2013: 107-117.

[10] GUFLER B, AUGSTEN N, REISER A, et al. Load balancing in MapReduce based on scalable cardinality estimates[C]//The 28th IEEE International Conference on Data Engineering(ICDE). Piscataway, NJ: IEEE, 2012: 522-533.

[11] JURE L. Stanford network analysis project[EB/OL]. [2015-09-21]. http://snap.stanford.edu/.

编 辑 叶 芳