并行计算框架 Spark 的自适应缓存管理策略

卞 琛[1,2],于 炯[1],英昌甜[1],修位蓉[1]
(1. 新疆大学信息科学与工程学院,新疆乌鲁木齐 830046; 2. 乌鲁木齐职业大学信息工程学院,新疆乌鲁木齐 830002)

摘 要: 并行计算框架 Spark 缺乏有效缓存选择机制,不能自动识别并缓存高重用度数据;缓存替换算法采用 LRU,度量方法不够细致,影响任务的执行效率. 本文提出一种 Spark 框架自适应缓存管理策略(Self-Adaptive Cache Management,SACM),包括缓存自动选择算法(Selection)、并行缓存清理算法(Parallel Cache Cleanup,PCC)和权重缓存替换算法(Lowest Weight Replacement,LWR). 其中,缓存自动选择算法通过分析任务的 DAG(Directed Acyclic Graph)结构,识别重用的 RDD 并自动缓存. 并行缓存清理算法异步清理无价值的 RDD,提高集群内存利用率. 权重替换算法通过权重值判定替换目标,避免重新计算复杂 RDD 产生的任务延时,保障资源瓶颈下的计算效率. 实验表明:我们的策略提高了 Spark 的任务执行效率,并使内存资源得到有效利用.

关键词: 并行计算; 缓存管理策略; Spark; 弹性分布式数据集
中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2017)02-0278-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2017.02.003

Self-Adaptive Strategy for Cache Management in Spark

BIAN Chen[1,2], YU Jiong[1], YING Chang-tian[1], XIU Wei-rong[1]
(1. *School of Information Science and Engineering, Xinjiang University, Urumqi, Xinjiang* 830046, *China*;
2. *School of Information and Engineering, Urumqi Vocational University, Urumqi, Xinjiang* 830002, *China*)

Abstract: As a parallel computation framework, Spark does not have a good strategy to select valuable RDD to cache in limited memory. When memory has been full load, Spark will discard the least recently used RDD while ignoring other factors such as the computation cost and so on. This paper proposed a self-adaptive cache management strategy (SACM), which comprised of automatic selection algorithm (Selection), parallel cache cleanup algorithm (PCC) and lowest weight replacement algorithm (LWR). Selection algorithm can seek valuable RDDs and cache their partitions to speed up data intensive computations. PCC clean-up the valueless RDD sasynchronously to improve memory utilization. LWR takes comprehensive consideration of the usage frequency of RDD, the RDD's computation cost, and the size of RDD. Experiment results show that Spark with our selection algorithm calculates faster than traditional Spark, parallel cleanup algorithm contributes to the improvement of memory utilization, and LWR shows better performance in limited memory.

Key words: parallel computing; cache management strategy; Spark; resilient distribution datasets

1 引言

利用内存的低延迟特性改进系统性能成为并行计算新的研究方向. Spark[1,2] 是继 Hadoop 之后出现的通用高性能并行计算框架,采用弹性分布式数据集(Resilient Distributed Datasets, RDD)[3] 作为数据结构. Spark 缓存管理策略中,程序员掌握缓存对象的选择权,增加了缓存策略的不确定性. 缓存替换算法采用 LRU,未考虑 RDD 计算代价及容量等影响应用程序执行效率的重要因素,度量方法不够细致. 因此,研究 Spark 框架自适应缓存策略具有一定的现实意义.

典型的缓存替换算法包括: FIFO、LRU、LFU、LRFU、MIN 等. 这些算法在并行计算框架得到广泛应用,但性能表现并不理想. 另外的一些研究成果则在缓存替换算法中加入了不同的参数,文献[4]在 FIFO 和 LRU 算法的基础上进行改进,引入附加参数进行置换目标的计算,但其参数选择不适用于 Spark. 文献[5]提出的 AWRP(Adaptive Weight Ranking Policy)算法为每

收稿日期:2015-09-02;修回日期:2015-11-16;责任编辑:蓝红杰
基金项目:国家自然科学基金(No.61262088, No.61462079)

个缓存对象计算权重,并优先转换权重值最低的缓存对象,但权重计算方法不具有普适性. 其他一些研究成果则考虑的是不同的系统特性和应用场景. 文献[6]针对多核处理器的高速缓存,设计了基于分区结构缓存替换算法;文献[7]提出了针对 websearch 应用的缓存替换算法;文献[8]则利用闪存的低延迟优势,设计了基于闪存的多级缓存管理系统.

近年来,一些研究人员致力于内存文件系统的研发. 文献[9]提出了 Tachyon,一种基于内存的分布式文件系统. 但在 Tachyon 的实现中,替换算法仍然采用 LRU. 文献[10]提出了内存文件系统 RAMCloud,但 RAMCloud 和 Spark 都属于高内存占用型系统,无法相互兼容.

本文提出一种基于 Spark 框架的自适应缓存管理策略(Self-Adaptive Cache Management,SACM),包括缓存自动化、有效缓存替换和提高内存利用率等方面的多项改进措施,最大限度消除内存资源瓶颈影响,使集群发挥最佳效能. 相比于已有的研究工作,自适应缓存管理策略更适宜于 Spark 框架的性能优化.

2　问题的建模与分析

本节首先分析 Spark 任务的执行机制,建立内存资源模型、任务执行效率模型和 RDD 权重模型,最后提出自适应缓存管理策略的问题定义.

2.1　Spark 任务执行机制

Spark 的任务执行采用了延时调度机制,即当用户对一个 RDD 执行 Action 操作时,调度器会根据 RDD 的 lineage 来构建一个 DAG,然后为工作结点分配子任务执行程序. Spark 任务 DAG 的典型示例如图 1 所示. 其中实线圆角方框表示 RDD,填充矩形表示分区,虚线框为 Stage. Spark 根据宽依赖划分 Stage,每个 Stage 都包含尽可能多的连续窄依赖,Stage 内部的窄依赖前后连接构成流水线,而各 Stage 则同步顺序执行,直到最终得出目标 RDD.

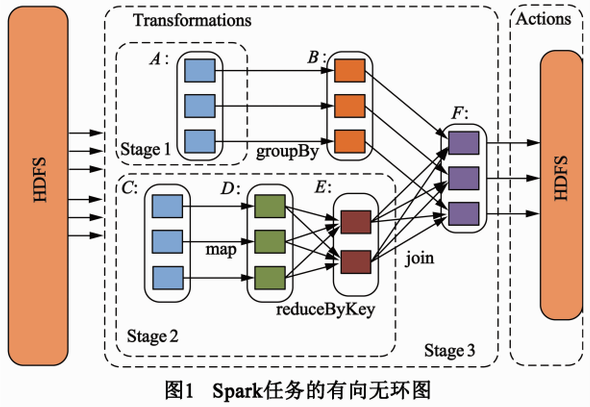

图1　Spark任务的有向无环图

2.2　内存资源模型

在并行计算集群中,资源池由一系列工作节点构成,定义工作节点集合 $W=\{1,2,\cdots,m\}$,集群内存资源集合 $R=\{r_1,r_2,\cdots,r_m\}$. 记 $Tasks=\{1,2,\cdots,n\}$ 为同一时间段内运行的任务,A_{im} 为任务 i 在工作节点 m 上的内存分配量,则任务的内存资源总量可表示为:

$$A_i=\sum_{m\in W}A_{im},i\in Tasks \tag{1}$$

由于 Spark 保证所有任务的并发执行,当且仅当每个工作节点的内存资源都不会溢出,即:

$$\sum_{i\in Tasks}A_{im}<r_m,m\in W \tag{2}$$

根据任务对资源需求的贪婪效应,当内存资源充足时,集群为任务 i 分配的内存资源应与任务所有 RDD 大小之和相等,而当内存资源不足时,分配内存小于 RDD 大小之和. 即:

$$\sum_{j\in Task_i}RDD_{ij}\geqslant A_i \tag{3}$$

定理 1　内存有效利用原则. 在不影响任务执行效率的前提下,任务的内存占用率越低,集群内存资源的利用率越高.

证明　记任务 i 调度时集群的空闲内存容量为 S_{vacant},则任务 i 在分配方案 A_i 下成功调度的概率为:

$$P=\frac{S_{vacant}}{A_i} \tag{4}$$

设任务 i 的两种内存分配方案为 A_x、A_y,且 $A_x>A_y$. 两种分配方案下任务的执行时间均为 t,由于 S_{vacant} 为常数,因此 $P_x<P_y$,即内存占用率越低的分配方案调度成功的概率越高,集群内存的利用率也越高.

2.3　任务执行效率模型

Spark 将 RDD 划分成多个分区,交由工作节点并行计算. 因此,对于每一个任务 i,记其 RDD 集合为 $Task_i=\{RDD_{i1},RDD_{i2},\cdots,RDD_{ij}\}$,这里 RDD_{ij} 表示任务 i 中第 j 个 RDD. 对于每个 RDD,记其分区集合为 $RDD_{ij}=\{P_{ij1},P_{ij2},\cdots,P_{ijk}\}$,其中 P_{ijk} 表示 RDD_{ij} 中的第 k 个分区.

定义 1　RDD 计算代价. Spark 任务中,分区是以一个或多个父节点为输入数据计算生成,设 $Parents_{ijk}$ 为分区 P_{ijk} 的父节点集合. 分区的计算要读取所有的输入数据,然后根据闭包和操作类型进行计算. 因此分区 P_{ijk} 的计算代价为数据读取代价与数据处理代价之和,我们以分区计算时间作为衡量计算代价的唯一指标,即:

$$T_{P_{ijk}}=\mathrm{read}(Parents_{ijk})+\mathrm{proc}(Parents_{ijk}) \tag{5}$$

若 $Parents_{ijk}$ 集合所有分区都存储在内存中,则数据读取代价可以忽略,即 $\mathrm{read}(Parents_{ijk})=0$. RDD 的所有分区由集群工作节点并行计算生成,因此其计算代价为所有分区计算代价的最大值,即:

$$T_{RDD_{ij}}=\max(T_{P_{ij1}},T_{P_{ij2}},\cdots,T_{P_{ijk}}) \tag{6}$$

定理 2 最小代价原则. 对于重复使用的 RDD,其重新计算代价高于缓存命中代价.

证明 若 RDD_{ij}缓存命中且需重复使用,则其每个分区读取时间 read(P_{ijk}) = 0;整个 RDD_{ij}的重新计算代价也为 0. 而若 RDD_{ij}缓存未命中,其生成代价为 $T_{RDD_{ij}}$,因此有:read(RDD_{ij}) < $T_{RDD_{ij}}$.

定义 2 任务执行时间. 如图 1 所示,Spark 以宽依赖为分界点,将任务划分为多个 Stage 执行,记 $stage_i$ 的 RDD 集合为{1,2,…,m},每个 RDD 有多个分区,那么 Stage 的执行时间应为所有 RDD 计算总时长,即:

$$T_{stage_i} = \sum_{i=1}^{m} T_{RDD_{ij}} \tag{7}$$

若 Spark 将任务划分为 n 个 Stage,则任务的执行时间为:

$$T_{task_i} = \sum_{i=1}^{n} T_{stage_i} \tag{8}$$

定理 3 最佳执行效率原则. 对于任务执行过程中多次使用的 RDD,缓存命中对任务执行具有加速作用.

证明 设 RDD_{ix}在任务 i 中重复使用,记 $f_{RDD_{ix}}$为使用次数,则 $f_{RDD_{ix}} > 1$. 对于任务 i 的分区集合 $Task_i$ = {RDD_{i1}, RDD_{i2}, …, RDD_{ij}},必然存在 $f_{RDD_{ix}}$ 个相同的 RDD_{ix},根据定理 2,若 RDD_{ix}缓存命中,后续 $f_{RDD_{ix}} - 1$ 个相等 RDD 的计算时间为 0,则任务执行加速比为:

$$Sp_{RDD_{ix}} = \frac{(f_{RDD_{ix}} - 1) \times T_{RDD_{ix}}}{T_{task_i}} \tag{9}$$

定义 3 RDD 生命周期. 对于任务 i 的 RDD_{ij},其生命周期的最大值为 RDD_{ij}开始计算时间到任务完成时间,记 $ST_{RDD_{ij}}$为 RDD_{ij}开始计算时间,FT_{Task_i}为任务完成时间,则:

$$maxLC_{RDD_{ij}} = FT_{task_i} - ST_{RDD_{ij}} \tag{10}$$

在不影响任务执行效率的前提下,RDD_{ij}生命周期的最小值为开始计算时间到最后一次使用完成的时间,记 $UT_{RDD_{ij}}$为 RDD_{ij}最后一次使用完成的时间,则有:

$$minLC_{RDD_{ij}} = UT_{RDD_{ij}} - ST_{RDD_{ij}} \tag{11}$$

因此 RDD 生命周期的值域可表示为:

$$minLC_{RDD_{ij}} \leqslant LC_{RDD_{ij}} \leqslant maxLC_{RDD_{ij}} \tag{12}$$

定义 4 内存资源熵. 用于衡量任务的内存需求与分配方案的匹配度. 对于任务 i,其内存资源熵为任务周期内单位时间内存占用量的最大值与内存分配总量的比值. 记 S_i = {S_{i1}, S_{i2}, …, S_{ih}}整个任务执行周期内单位时间内存占用量,则任务 i 的内存资源熵可表示为:

$$Q_i = \frac{A_i}{\max(S_{ih})} \tag{13}$$

内存资源熵精确刻画了任务内存需求与分配方案的契合度. 熵值越大,表示集群对任务内存需求的分配适应度越高,任务受内存瓶颈的影响也越小.

定理 4 资源最佳匹配原则. 在不影响任务执行效率的前提下,当且仅当任务所有 RDD 的生命周期取最小值时,任务的内存资源熵最大.

证明 根据定义 4,max(S_{ih})表示任务周期内单位时间内存占用量的最大值,从任务调度的角度来看,max(S_{ih})也表示任务周期内所有 RDD 生命周期的最大交集,因此,当任务 i 所有 RDD 都满足条件:

$$LC_{RDD_{ij}} = minLC_{RDD_{ij}}$$

则:
$$Q_i = \frac{A_i}{\min(\max(S_{ih}))}$$

Q_i 为任务 i 在分配方案 A_i 下内存资源熵的最大值,即当任务所有 RDD 的生命周期取最小值时,任务需求与分配方案达到最优匹配.

2.4 RDD 权重模型

内存资源熵 Q_i < 1 时,任务执行过程中需要缓存替换,我们建立 RDD 权重模型,作为替换算法的设计依据. 通过对 Spark 任务的分析,RDD 权重有以下 4 个重要因素:RDD 使用频率、输入数据大小、RDD 计算复杂度和 RDD 容量大小.

定义 5 RDD 权重. 记 $f_{RDD_{ij}}$为 RDD_{ij}的使用频率,O(RDD_{ij})表示计算复杂度,S 表示大小,μ 作为调整系数. RDD_{ij}的权重表示如下:

$$V_{RDD_{ij}} = \frac{\mu \times (f_{RDD_{ij}} - 1) \times O(RDD_{ij}) \times S_{Parents_{ij}}}{S_{RDD_{ij}}} \tag{14}$$

在上述影响 RDD 权重的 4 个因素当中,计算复杂度是最难判定的因素,必须通过编译解析或预执行的方法来评估. 因此换一角度考虑,利用 2.3 节 RDD 计算代价的定义,记录 RDD_{ij}计算的起始时间和完成时间,以 RDD 生成时长作为计算代价的度量方法. 因此 RDD 计算代价表示为:

$$Cost_{RDD_{ij}} = FT_{RDD_{ij}} - ST_{RDD_{ij}} \tag{15}$$

计算代价包括了 RDD 权重的 2 个因素:计算复杂度和输入数据大小,因此可用于替换权重公式中的 $O(RDD_{ij}) \times S_{Parents_{ij}}$. 另外通过实验发现,任务出现两个使用频率、计算代价都相等 RDD 的几率很低,因此权重计算忽略了 RDD 大小这一因素. 最终的 RDD 权重计算公式表示为:

$$V_{RDD_{ij}} = \mu \times (f_{RDD_{ij}} - 1) \times Cost_{RDD_{ij}} \tag{16}$$

定理 5 任务效率最小牺牲原则. 在任务执行过程中进行缓存替换,优先替换权重小的 RDD 对任务执行效率的影响越小.

证明 设任务 i 执行缓存替换,对于任意重用的 RDD_{ix},根据式(14)、(15),RDD_{ix}缓存贡献的任务加速比可推导为:

$$Sp_{RDD_{ix}} = \frac{(f_{RDD_{ix}} - 1) \times T_{RDD_{ix}}}{T_{task_i}} = \frac{(f_{RDD_{ix}} - 1) \times Cost_{RDD_{ix}}}{T_{task_i}}$$

$$= \frac{V_{RDD_{ix}}}{\mu \times T_{task_i}}$$

对于替换目标 RDD_{iy}、RDD_{iz},若 $V_{RDD_{iy}} > V_{RDD_{iz}}$,则:

$$Sp_{RDD_{iy}} > Sp_{RDD_{iz}}$$

因此优先置换权重小的 RDD_{iz}对任务执行效率的影响较小.

2.5　自适应缓存管理策略问题定义

前面几节已经对 Spark 的内存分配、任务执行效率进行了详细的阐述,基于这些定义,我们的自适应缓存管理策略可形式化为:

object

$$\frac{A_i}{\min(\max(S_{ih}))} \tag{17}$$

$$\max(Sp_{task_i}) \tag{18}$$

s. t.

$$\sum_{j \in Task_i} RDD_{ij} \geqslant A_i \tag{19}$$

目标是最大化内存资源熵和任务加速比,约束条件是内存分配量小于等于任务生成的 RDD 容量之和. 很显然,通过定理 3 的证明,缓存重用的 RDD 可以获得最大任务加速比. 定理 4 的目标则是最大化内存资源熵,降低内存占用率,提高利用率. 当内存空间不足时,系统通过定理 5 获得内存瓶颈下的最大任务加速比.

3　自适应缓存管理策略

本节基于模型的相关定义以及定理证明,提出缓存自动选择算法、并行缓存清理算法和权重缓存替换算法.

3.1　缓存自动选择算法

下面根据定理 3 求解 2.5 节定义的带约束条件优化问题,提出缓存自动选择算法,自动识别并缓存重用的 RDD,算法的主要思想为:

(1)生成 RDD 结构树. 根据 Spark 的延迟调度机制,可在计算开始前通过分析 DAG 生成 RDD 结构树 *treeRDDs*.

(2)计算每个 RDD 的使用次数. 算法遍历 RDD 结构树,生成键值对集合 $R < RDD, N >$,其中 N 表示 RDD 的使用次数.

(3)缓存自动化. 遍历键值对集合 R,将 $N > 1$ 的 RDD 放入待缓存集合,新 RDD 生成时,若缓存标志量为 True,则自动缓存.

(4)计算权重值. 通过 2.4 节的 RDD 权重公式,为缓存的 RDD 计算权重值并保存.

缓存自动选择算法的操作过程如算法 1 所示.

我们将渐近填充[11]的思想加入到算法设计中,尽最大可能缓存高重用度 *RDD*,加速任务执行. 算法中生成的权重集合将在 3.3 节的权重缓存替换算法中使用,以解决内存资源枯竭问题.

算法 1　缓存自动选择算法

```
输入:结构树 treeRDDs;
      空闲缓存容量 cachesize;
初始化:R←newHashmap;
        cachelist←new Array;
        weightlist←newHashmap;
1.      for i = 0 to treeRDDs. Length-1 do
2.        if R. contains(treeRDDs[i]) then
3.          N + + ;
4.          if N > 1 then
5.            cachelist. add(treeRDDs[i]);
6.          end if
7.        else
8.          R. add(treeRDDs[i],1);
9.        end if
10.     end for
11.     for j = 0 to cacheList. Length-1 do
12.       if RDD_j > cachesize then
13.         call Algorithm3;//调用缓存替换算法
14.       end if
15.       cacheAllPartition(RDD_j);//缓存 RDD
16.       v = calcWeight(RDD_j);//计算 RDD 权重
17.       weightlist. add(RDD_j,v);//加入权重集合
18.     end for
```

3.2　并行缓存清理算法

我们根据定理 4 提出并行缓存清理算法,其主要思想为:(1)获得缓存自动选择算法生成的结构树 *treeRDDs* 和键值对集合 R;(2)遍历以当前生成 RDD 为根节点的子树,将除根节点外的每个节点在集合 R 中的使用次数减 1. 若 RDD 的使用次数为 0,则清理该 RDD;(3)以当前生成的 RDD 为临界点,对结构树 *treeRDDs* 剪枝,去除所有使用次数为 0 的节点,降低后续清理的时间复杂度. 具体操作过程如算法 2 所示.

算法 2　并行缓存清理算法

```
输入:结构树 treeRDDs;
      键值对集合 R;
初始化:rdd←null;
        subtree←new Tree;
        cutlist←new List < RDD > ;
1.      rdd←getCurrentRDD();//取当前 RDD
2.      subtree←treeRDDs. sub(rdd);//取子树
3.      for i = 0 to subtree. Length-1 do
4.        node←R. find(subtree[i]);
5.        node. Num←node. Num-1
6.        if node. Num = = 0 then
7.          cleanup(node);//执行 RDD 清理
8.          cutlist. add(node);//添加到剪枝列表
```

```
9.          end if
10.       end for
11.       for i = 0 to cutlist. Length-1 do
12.         treeRDDs. remove( cutlist[ i ] );//剪枝
13.       end for
```

在并行清理算法中,剪枝操作使 RDD 结构树在多次清理中不断缩减,减少后续清理的遍历次数,降低算法复杂度. 从整体任务执行过程来看,清理算法的执行过程与任务计算过程是并发的,任务执行时间不受清理算法的影响. 同时由于清理算法的复杂度远低于 RDD 计算复杂度,可以保障清理算法自身不会出现多次并发的情况.

3.3 权重缓存替换算法

我们根据 2.4 节的 RDD 权重模型,提出权重缓存替换算法. 其主要思想如下:(1)获取需要缓存 RDD 的大小和权重;(2)对已有 RDD 权重集合进行条件过滤,将权重小于新 RDD 的对象放入候选替换列表;(3)将候选替换列表按权重值从小到大顺序排列;(4)搜索候选替换列表,若存在替换目标,其容量与空闲内存容量之和大于等于新 RDD 大小,则执行替换. 若不存在,则放弃替换. 具体操作如算法 3 所示.

算法 3 权重缓存替换算法

```
输入:RDD 权重集合 weightlist;
      空闲缓存容量 cachesize;
      新缓存 RDD 权重 v;
      新缓存 RDD 大小 s;
初始化:    candidates←new List < RDD > ;
1.        for i = 0 to weightlist. Length-1 do
2.          if v > weightlist[ i ]. Weight then
3.            candidates. add( weightlist[ i ] );
4.          end if
5.        end for
6.        candidates. orderByWeight( ) ;//排序
7.        for i = 0 to candidates. Length-1 do
8.          if candidates[ i ]. size + cachesize > s then
9.            replace( candidates[ i ] ) ;//缓存替换
10.             return;
11.         end if
12.       end for
```

4 实验与评价

本节将通过实验进行比较和评价,验证缓存自动选择算法、并行缓存清理算法和权重缓存替换算法的有效性.

4.1 实验环境

实验环境用 1 台服务器和 8 个工作节点建立计算集群,服务器作为 Spark 的 Master 和 Hadoop 的 NameNode. 任务执行时间的数据来源于 Spark 的控制台,而内存使用状况的监控由 nmon 完成. 实验任务选用数据密集型算法 PageRank,数据选取 SNAP[12] 提供的 7 个标准数据集.

4.2 缓存自动选择算法

实验选择了不同大小的数据集,在充足内存条件下进行测试算法性能,通过多次采样取平均值的方法确保数据有效性,实验结果如图 2 所示,其中图 2(*a*)表示相同任务不同数据集的内存占用率;图 2(*b*)表示不同数据集的任务执行时间.

如图 2(*a*)可以看出,节点数和连线数越大的数据集内存占用率越高. 从对比结果来看,缓存选择算法的内存占用率低于传统 Spark,这并不符合我们的预期,理论上缓存高重用度的 RDD 应当带来额外的内存开销. 通过分析 Spark 源码,缓存过程只是将 RDD 的指针由计算区移动到缓存区,并不执行数据复制,因此不占用额外的内存空间.

由图 2(*b*)可以看出,在相同数据集条件下,缓存自动选择算法优化效果非常明显. 横向比较 4 组实验结果,数据集越大,缓存策略的影响越大,缓存自动选择算法的提升效果也越明显. 这符合一般意义上的用户预期,用户的任务量大时,更期望完成时间得到更大改善.

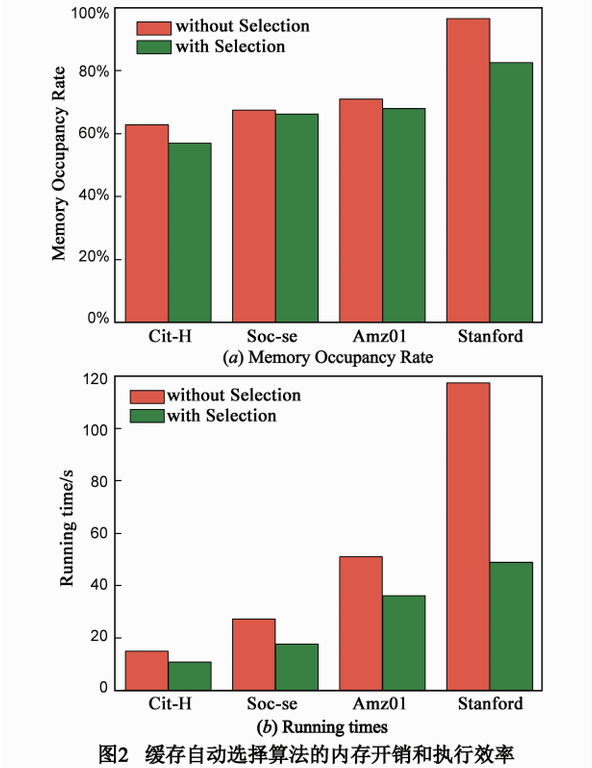

图2 缓存自动选择算法的内存开销和执行效率

4.3　并行缓存清理算法

实验选取节点和连接数差异较大的 2 个数据集,同样在充足内存条件下测试. 实验结果如图 3,图 3(*a*)和 3(*b*)分别为 Soc-sign-eponions 和 Amazon0302 的测试结果.

由实验结果可以看出,同一任务两次测试的执行时间非常相近,每个任务的内存使用均呈现梯度上升的趋势. 比较内存占用率变化曲线,并行缓存清理算法梯度上升较为缓和,峰值也低于传统 Spark. 因此,并行缓存清理算法的内存平均占用率和最高占用率都低于传统 Spark,证明了定理 4 的正确性.

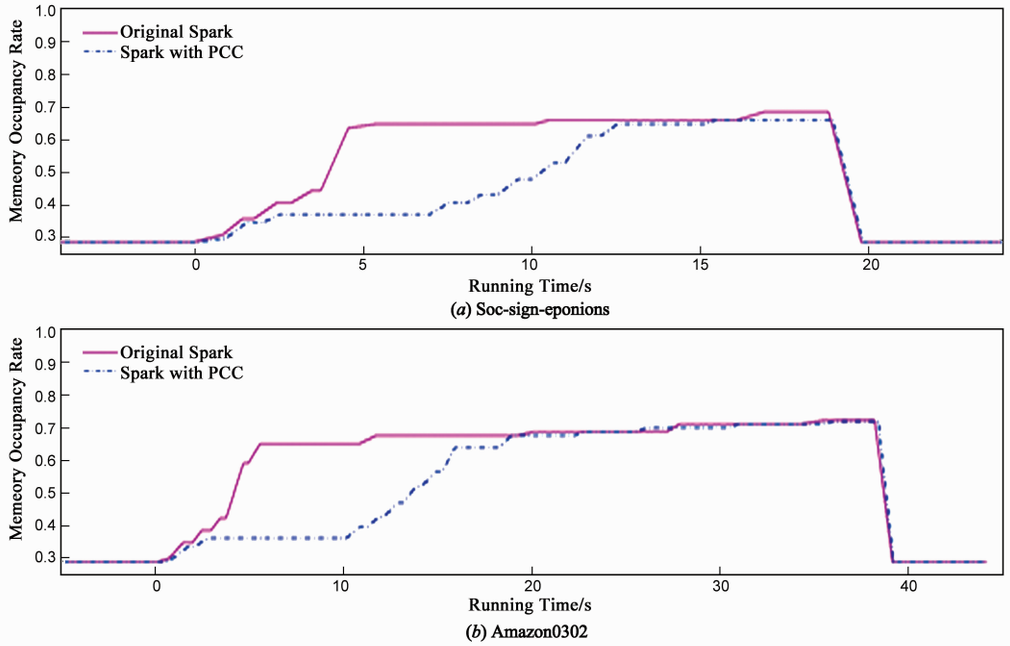

图3　并行清理算法的内存使用情况

4.4　权重缓存替换算法

实验选择了 2 个高内存占用率的数据集进行测试. 我们增加了任务的缓存数据量,一方面能够确保缓存替换算法的执行,另一方面可以将实验结果与预期结果进行对比. 实验监控了不同迭代次数任务的执行时间,记录数据取多次实验的平均值. 结果如图 4 所示,其中图 4(*a*)表示 web-Google 数据集不同迭代次数下的任务完成时间,图 4(*b*)则为 WikiTalk 数据集测试的性能表现.

由图 4(*a*)可以看出,权重缓存替换算法的任务执行时间呈线性增长的趋势,而在 LRU 算法测试结果中,2 个区间的任务时长急剧上升. 在图 4(*b*)中,权重替换算法和 LRU 在前两个监测点的数据差异不大,因为在内存充足时,权重替换算法并不能体现优势. 而在 6 次迭代时,LRU 算法出现较大任务延时,权重缓存替换算法的执行时间仍然保持稳步增长. 综合比较两次实验结果,我们的算法在频繁发生缓存替换时,相对于 LRU 的优化效果较为明显.

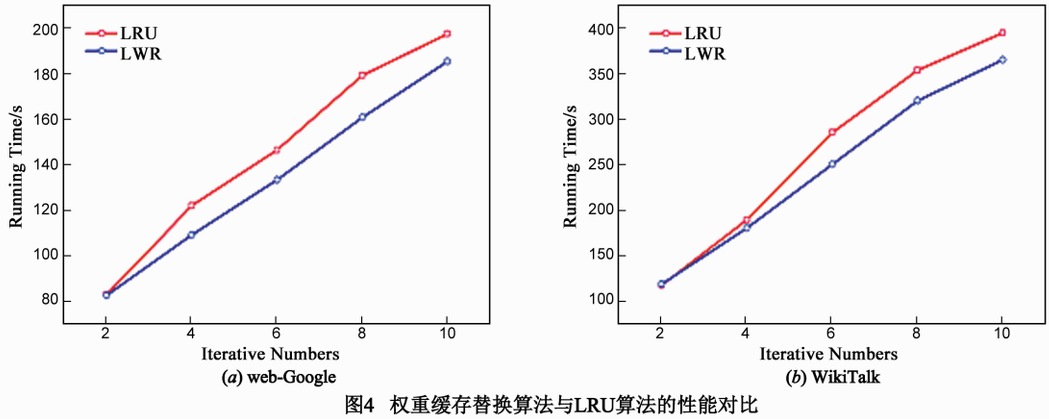

图4　权重缓存替换算法与LRU算法的性能对比

4.5 综合评估

前面的实验单独验证每个算法的执行性能,下面整体测试自适应缓存管理策略的有效性. 我们将多个任务合成为一个工作集并发执行. 实验记录不同迭代次数下,工作集任务的执行时间,如图 5 所示. 由实验结果很容易看出,我们的策略对 Spark 框架具有良好的优化效果.

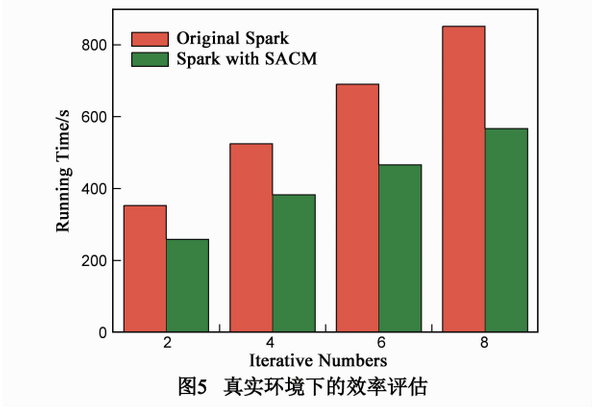

图5 真实环境下的效率评估

5 结论

本文建立了基于 Spark 框架的自适应缓存管理策略. 其中,缓存自动选择算法自动缓存任务中高重用度的 RDD,优化任务执行效率;并行缓存清理算法异步清理使用完成的 RDD,提高集群的内存利用率;权重缓存替换算法综合考虑使用频率、计算代价等因素执行缓存替换,优化内存瓶颈下的任务执行效率. 理论与实验结果证明,自适应缓存管理策略提高了 Spark 的任务执行效率,并使内存资源得到有效利用.

参考文献

[1] Zaharia M,Das T,Li H,et al. Discretized streams:An efficient and fault-tolerant model for stream processing on large clusters[A]. Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing[C]. Boston,MA:USENIX,2012. 1 –6.

[2] Apache Spark. Spark Overview[EB/OL]. http://spark.apache.org,2015-01-21/2015-03-18.

[3] Zaharia M,Chowdhury M,Das T,et al. Resilientdistributed datasets:A fault-tolerant abstraction for in-memory cluster computing[A]. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation [C]. Berkeley,CA:USENIX,2012. 1 –14.

[4] Young N. On-line file caching[A]. Proceedings of the 9th Annual ACM-SIAM Symposium on DiscreteAlgorithms [C]. Baltimore,MD:ACM,1999. 82 –86.

[5] Swain D,Paikaray B,Swain D. AWRP:Adaptive weight ranking policy for improving cache performance[EB/OL]. http://arxiv.org/abs/1107.4851,2011-07-25/2015-03-18.

[6] FANG Juan,WANG Jing,LI Chengyan,et al. Partition-based cache replacement to manage shared L2 caches[J]. Chinese Journal of Electronics,2014,23(3):464 –467.

[7] 司成祥,孟晓烜,许鲁. 一种针对 websearch 应用的缓存替换算法[J]. 电子学报,2011,39(5):1205 –1209.
SI Chengxiang,MENG Xiaoxuan,XU Lu. A novel replacement algorithm designed for websearchapplications[J]. Acta Electronica Sinica,2011,39(5):1205 –1209. (in Chinese)

[8] Byan S,Lentini J,Madan A,et al. Mercury:Host-side flash caching for the datacenter[A]. Proceedings of the 28th Symposium on Mass Storage Systems and Technologies [C]. New York,NY:ACM,2012. 1 –12.

[9] HaoyuanLi,Ghodsi A,Zaharia M,et al. Tachyon:Reliable,memory speed storage for cluster computing frameworks [A]. Proceedings of the 27th IEEE Conference on SYSTEM-ON-CHIP[C]. Las Vegas,NV:IEEE,2014. 1 –15.

[10] Ongaro D,Rumble S M,Stutsman R,et al. Fast crash recovery in RAMCloud[A]. Proceedings of 23th ACM Symposium on Operating Systems Principles[C]. New York,NY:ACM,2011. 29 –41.

[11] Ghodsi A,Zaharia M,Shenker S,et al. Choosy:Max-min fair sharing for datacenter jobs with constraints[A]. Proceedings of the 8th ACM European Conference on Computer Systems[C]. New York,NY:ACM,2013. 365 –378.

[12] Jure L. Stanford Network Analysis Project[EB/OL]. http://snap.stanford.edu/,2013-05-16/2015-03-18.

作者简介

卞 琛 男,1981 年出生. 博士研究生,CCF 会员. 主要研究方向包括:内存计算、高性能计算、分布式系统等.
E-mail:bianchen0720@126.com

于 炯 男,1964 年出生. 教授、博士生导师,CCF 高级会员. 主要研究方向包括:网格计算、并行计算、分布式系统等.