计算机体系结构课程实验

实验环境

硬件
Intel i7-8750H CPU（6Cores 12Threads 9 MB Intel® Smart Cache）
16GB MEMORY

软件版本

Windows10
spark2.3.3
scala2.11
hadoop2.6.0

实验 1(20%)：graphX API 练习

GraphX

GraphX 是一个 Spark API，它用于图和并行图(graph-parallel)的计算。GraphX 通过引入 **Resilient Distributed Property Graph**：带有顶点和边属性的有向多重图，来扩展 Spark RDD(Resilient Distributed Dataset，一个可并行操作的有容错机制的数据集合)。

GraphX 项目的目的就是将 `graph-parallel` 和 `data-parallel` 统一到一个系统中，这个系统拥有一个唯一的组合 API。GraphX 允许用户将数据当做一个图和一个集合（RDD），而不需要而不需要数据移动或者复杂操作，优化图操作的执行。

GraphLoader.edgeListFile

GraphLoader.edgeListFile 提供了一个方式从磁盘上的边列表中加载一个图。它解析如下形式（源顶点 ID，目标顶点 ID）的连接表，跳过以#开头的注释行。它从指定的边创建一个图，自动地创建边提及的所有顶点。

```
# This is a comment
2 1
4 1
1 2
```

```
val  path = ".//file//Wiki-Vote.txt"// 数据集文件地址
val loadedGraph = GraphLoader.edgeListFile(sc,path)
```

本次 lab 从 Wiki-Vote.txt 加载一个图，生成 `class Graph[VD, ED]` 的对象，并赋值给常量 loadedGraph。

Pregel 图计算模型

Pregel 采用迭代的计算模型：在每一轮，每个顶点处理上一轮收到的消息，并发出消息给其它顶点，并更新自身状态和拓扑结构等。算法是否能够结束取决于是否所有的顶点都已经 vote 标识其自身已经达到 halt 状态了。顶点通过将其自身的状态设置成 halt 来表示它已经不再 active。

Pregel API

在 GraphX 中，更高级的 Pregel 操作是一个约束到图拓扑的批量同步（bulk-synchronous）并行消息抽象。与更标准的 Pregel 实现不同的是，GraphX 中的顶点仅仅能发送信息给邻居顶点，并利用用户自定义的消息函数构造消息。这些限制允许在 GraphX 进行额外的优化。

pregel 有两个参数列表（`graph.pregel(list1)(list2)`）。第一个参数列表包含配置参数初始消息、最大迭代数、发送消息的边的方向（默认是沿边方向出）。第二个参数列表包含用户自定义的函数用来接收消息（vprog）、计算消息（sendMsg）、合并消息（mergeMsg）。

```
def pregel[A: ClassTag](
initialMsg: A,
maxIterations: Int = Int.MaxValue,
activeDirection: EdgeDirection = EdgeDirection.Either)(
vprog: (VertexId, VD, A) => VD,
sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
mergeMsg: (A, A) => A)
: Graph[VD, ED] = {
Pregel(graph, initialMsg, maxIterations, activeDirection)(vprog, sendMsg, mergeMsg)
}
```

API 中的参数：

1. initialMsg：第一次迭代的时候每个顶点获取的初始值。

2. maxIterations：最大迭代的次数

3. activeDirection：在 pregel 函数中，该参数的默认值设置为 EdgeDirection.Either。

4. vprog：用户定义的 vertex program，在每个收到 inbound message 的顶点上进行并行计算（也就是该顶点的计算结果不受其他顶点的计算结果的影响），计算出顶点的一个新的值。

用 Pregel 操作表达计算 SSSP (single source shortest path)

```
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  // Vertex Program，节点处理消息的函数，dist为原节点属性（Double），newDist为消息类型（Double）
  (id, dist, newDist) => math.min(dist, newDist),

  // Send Message，发送消息函数，返回结果为（目标节点id，消息（即最短距离））
  triplet => {
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  //Merge Message，对消息进行合并的操作，类似于Hadoop中的combiner
  (a, b) => math.min(a, b)
)
println(sssp.vertices.collect.mkString("\n"))
```

计算单源最短路径的方法： Dijkstra 算法:

最短路径的最优子结构性质：如果 `P(i, j)={Vi, ..Vk, ..Vs,....Vj}`是从 i 到 j 的最短路径，k 和 s 是这条路径中的中间顶点，那么 P(k, s)必定是从 k 到 s 的最短路径。

迪科斯彻算法应用了贪心算法思想。贪心算法是指：再对问题求解的时候，总是做当前看来是最好的选择。也就是不从整体最优上加以考虑，但这却往往能得到最优解。贪心策略的选择必须具备无后效性，也就是整个状态中以前的过程不会影响以后的状态，只跟当前状态有关。

使用 pregel 来实现 Dijkstra 算法

从原点出发，依次计算其每一层的子节点到原点的最短距离，直到迭代到了所有的连通的顶点。此时就可以计算出每个顶点跟原点之间的最短距离。

第一次迭代时，每个顶点获取的值是最大值
每个顶点收集到的临界顶点发送过来消息汇总，取出发来的最小值。
一个 triplet 里，如果 src 中存储的到原点的距离加上当前边权小于 dst 顶点到原点的距离，就说明 dst 中保存的到原点的距离并不是最小的。就需要向 dst 顶点发送消息。

实验 2(40%)：使用 SSSP 和 PageRank 处理 Wikipedia 图并测量内存使用率和 cache 命中率等内存参数

GraphX 算法模型：PageRank

Pagerank 算法在 `org.apache.spark.graphx.lib` 包中有实现。

该 PageRank 模型提供了两种调用方式：

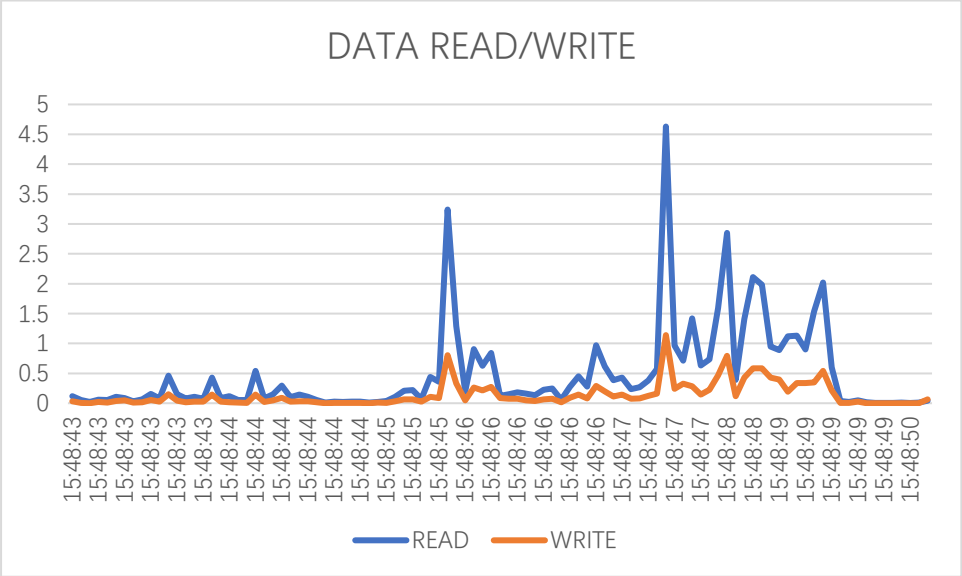
第一种：静态调用：staticPageRank(int)　在调用时提供一个参数 number，用于指定迭代次数，即无论结果如何，该算法在迭代 number 次后停止计算，返回图结果。

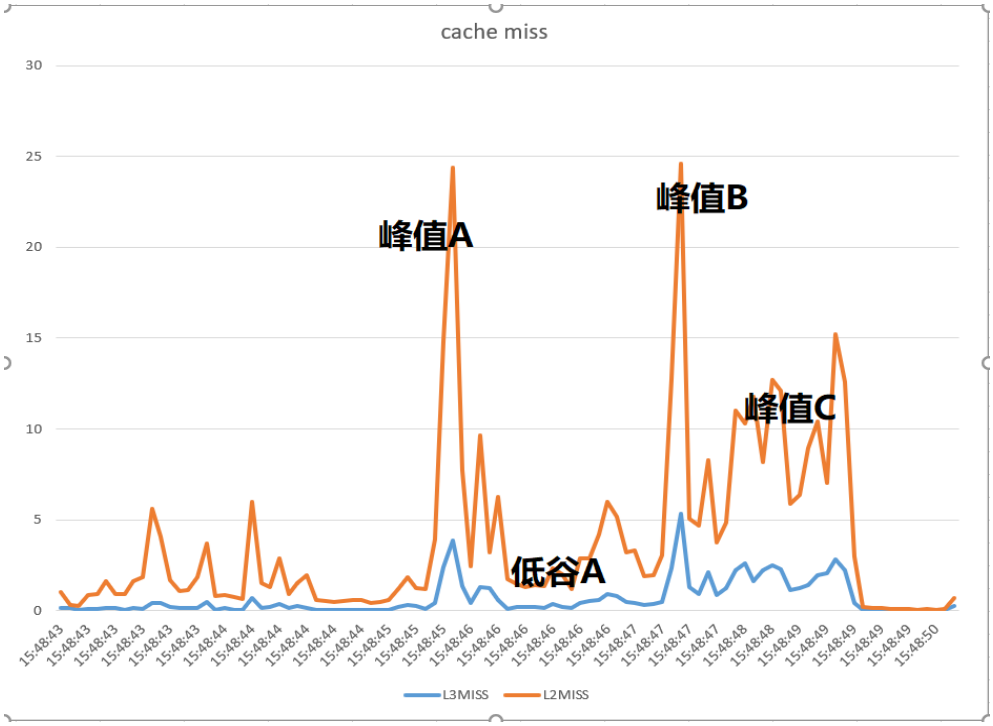
第二种：　动态调用：pageRank(double)　在调用时提供一个参数 tol，用于指定前后两次迭代的结果差值应小于 tol，以达到最终收敛的效果时才停止计算，返回图结果。所以 tol 参数值越小得到的结果越有说服力。本次 lab 中 tol 参数设为 0.01。

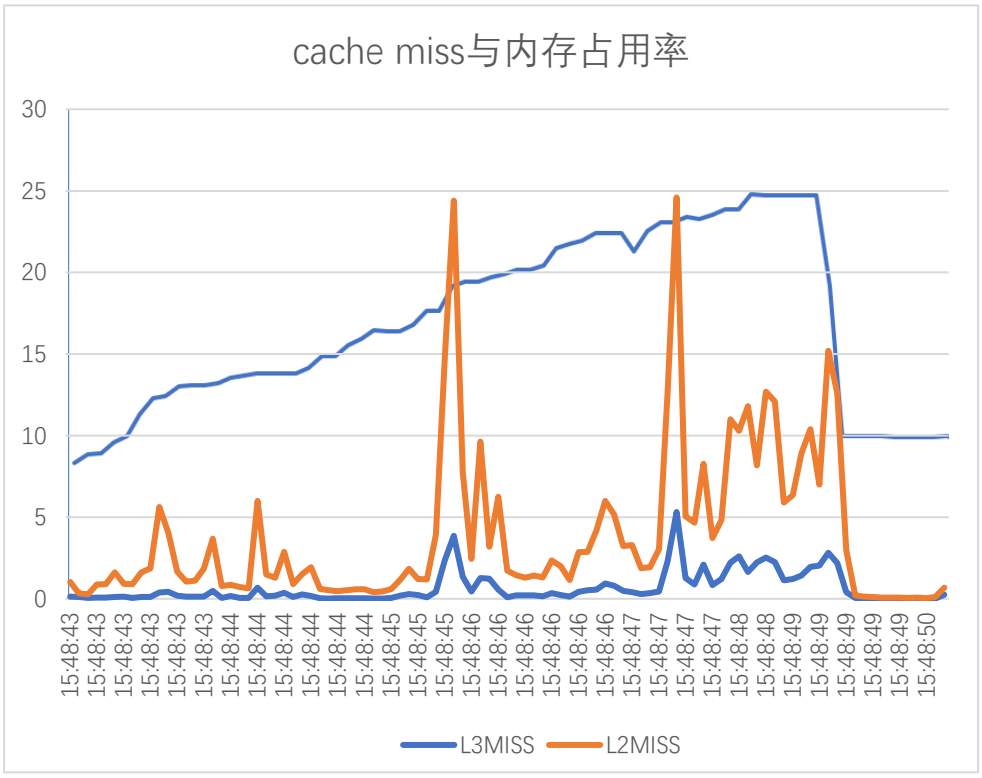
动态调用的伪代码如下

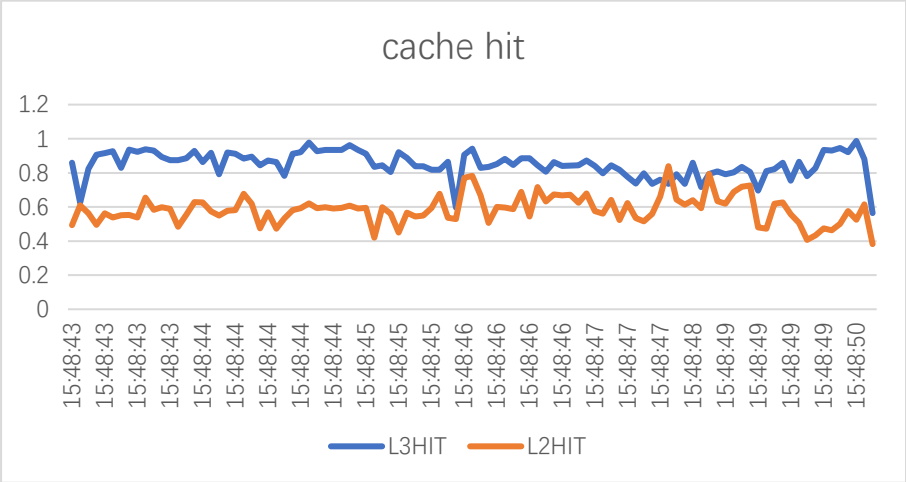
```
* {{{
 * var PR = Array.fill(n)( 1.0 )
 * val oldPR = Array.fill(n)( 0.0 )
 * while( max(abs(PR - oldPr)) > tol ) {
 *   swap(oldPR, PR)
 *   for( i <- 0 until n if abs(PR[i] - oldPR[i]) > tol ) {
 *     PR[i] = alpha + (1 - \alpha) * inNbrs[i].map(j => oldPR[j] / outDeg[j]).sum
 *   }
 * }
 * }}}
 *
 * `alpha` is the random reset probability (typically 0.15)`
```

PageRank 处理 Wikipedia 图的图表与分析




cache miss
30
25
20
15
10
5
0
峰值A
峰值B
峰值C
低谷A
15:48:43
15:48:44
15:48:45
15:48:46
15:48:47
15:48:48
15:48:49
15:48:50
L3MISS
L2MISS


cache miss与内存占用率
30
25
20
15
10
5
0
15:48:43
15:48:44
15:48:45
15:48:46
15:48:47
15:48:48
15:48:49
15:48:50
L3MISS
L2MISS



由于 CACHE MISS 的波动较为明显，选取 cache miss 图进行分析。

1. CACHE MISS 与读的负载相关性极强，当数据读的次数上升时，CACHE MISS 也显著上升。

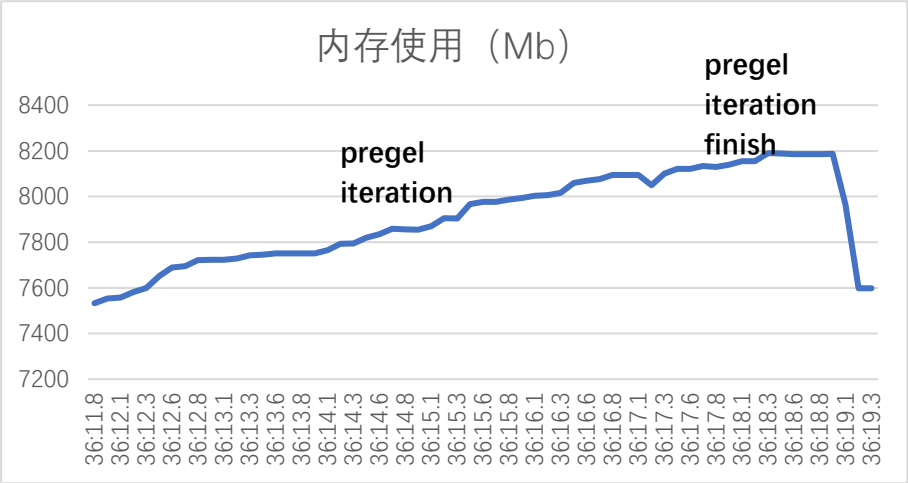
2. CACHE MISS 峰值分析

	该时刻程序运行内容	导致 cache miss 升高的原因	导致 L2 cache miss 大于 L3 cache miss 的原因
峰值 A	通过 RDD 之间的依赖关系构建 DAG	频繁读取 RDD,读取次数增大，CACHE MISS 相应增加	1. L3 cache 比 L2 cache 大 2. L3 cache 被多个核共享，而 L2 cache 的数据会经常变成 Invalid 状态
峰值 B	PREGEL iteration，同时调用 ContextCleaner 清理器, ContextCleaner 用于清理那些超出应用范围的 RDD、Shuffle 对应的 map 任务状态、Shuffle 元数据、Broadcast 对象以及 RDD 的 Checkpoint 数据。	内存的数据被清除，Cache 中的数据过时	
峰值 C	PREGEL iteration,但 Cleaned accumulator 没有频繁运行	内存使用率增大，L2 cache 出现 Capacity Miss 或者 Conflict Miss，这点从峰值 C 的 L3 MISS 相对于峰值 A 也增加可以见得	

3. CACHE MISS 低谷分析

	该时刻程序运行内容	导致 cache miss 降低的原因
低谷 A	PREGEL iteration,但 Cleaned accumulator 没有频繁运行	内存的数据尚且没有过时 而且数据读写的 Throughput 减小

系统内存使用(按 Mb 计算)



从 Pregel iteration start 处开始，到 Pregel finish iteration 为止，内存的使用率稳步上升。

原因一：Spark 通过 BlockManager 机制在内存以及磁盘上保存 Pregel 图计算处理的中间结果。

如下 log 所打印的情况

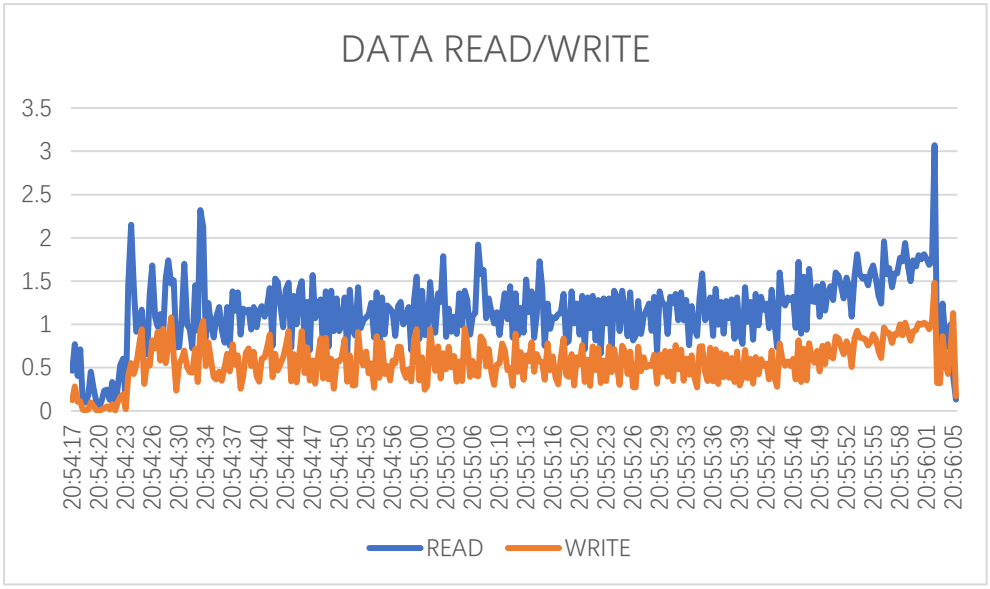
2020-06-08 14:36:14 INFO BlockManagerInfo:54 - Added broadcast_0_piece0 in memory on DESKTOP-B9R5OSG:56702 (size: 20.4 KB, free: 366.3 MB)

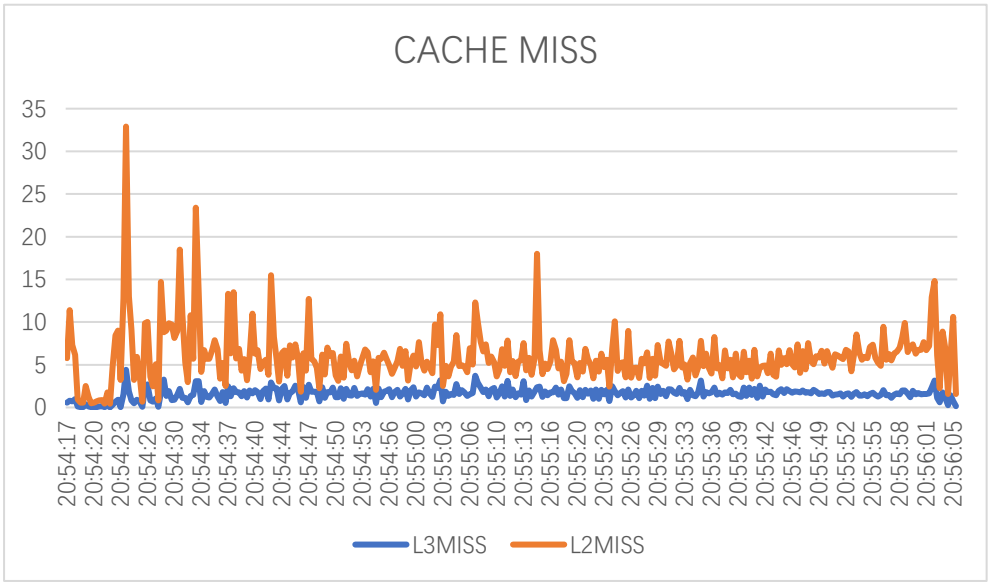
BlockManager 是管理整个 Spark 运行时数据的读写，包含数据存储本身，在数据存储的基础之上进行数据读写。使用 BlockManager 进行写操作时，比如说，RDD 运行过程中的一些中间数据，或者我们手动指定了 persist()，会优先将数据写入内存中，如果内存大小不够，会使用自己的算法，将内存中的部分数据写入磁盘。

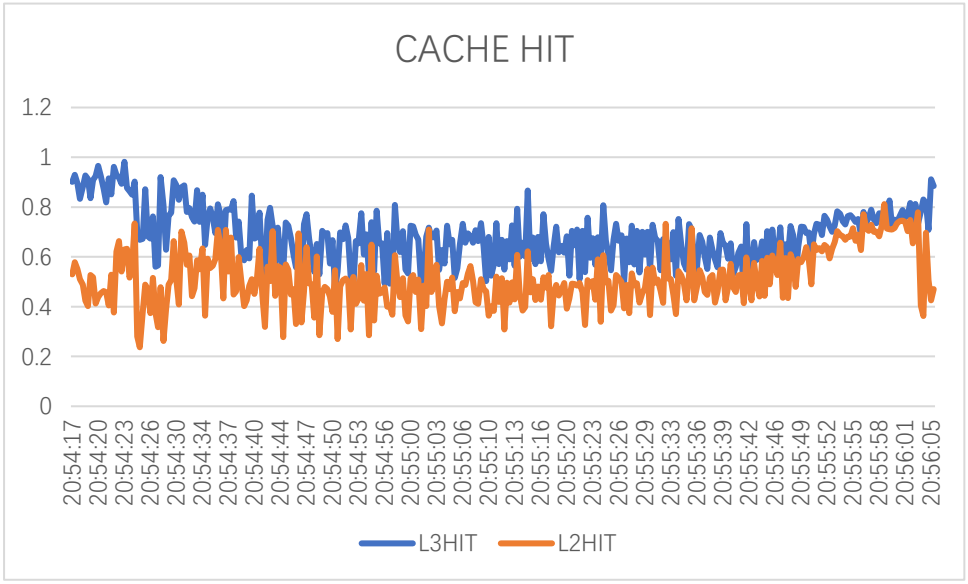
原因二：DAGScheduler, Executor 的内存占用

实验 3(20%): 处理 Google 图并测量内存使用率和 cache 命中率等内存参数。

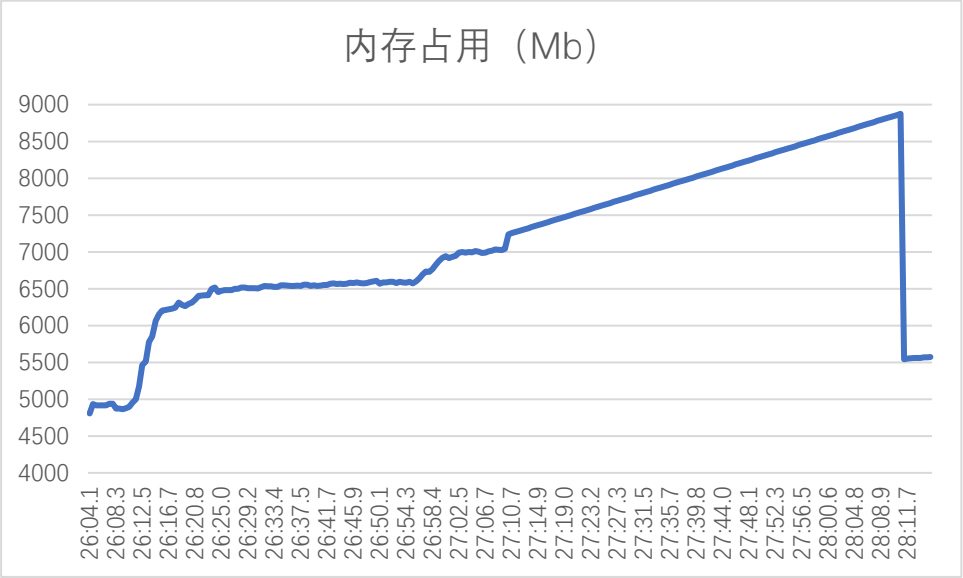
PageRank 处理 Google 图的图表与分析


DATA READ/WRITE
3.5
3
2.5
2
1.5
1
0.5
0
20:54:17
20:54:20
20:54:23
20:54:26
20:54:30
20:54:34
20:54:37
20:54:40
20:54:44
20:54:47
20:54:50
20:54:53
20:54:56
20:55:00
20:55:03
20:55:06
20:55:10
20:55:13
20:55:16
20:55:20
20:55:23
20:55:26
20:55:29
20:55:33
20:55:36
20:55:39
20:55:42
20:55:46
20:55:49
20:55:52
20:55:55
20:55:58
20:56:01
20:56:05
READ
WRITE


CACHE MISS
35
30
25
20
15
10
5
0
20:54:17
20:54:20
20:54:23
20:54:26
20:54:30
20:54:34
20:54:37
20:54:40
20:54:44
20:54:47
20:54:50
20:54:53
20:54:56
20:55:00
20:55:03
20:55:06
20:55:10
20:55:13
20:55:16
20:55:20
20:55:23
20:55:26
20:55:29
20:55:33
20:55:36
20:55:39
20:55:42
20:55:46
20:55:49
20:55:52
20:55:55
20:55:58
20:56:01
20:56:05
L3MISS
L2MISS


CACHE HIT
1.2
1
0.8
0.6
0.4
0.2
0
20:54:17
20:54:20
20:54:23
20:54:26
20:54:30
20:54:34
20:54:37
20:54:40
20:54:44
20:54:47
20:54:50
20:54:53
20:54:56
20:55:00
20:55:03
20:55:06
20:55:10
20:55:13
20:55:16
20:55:20
20:55:23
20:55:26
20:55:29
20:55:33
20:55:36
20:55:39
20:55:42
20:55:46
20:55:49
20:55:52
20:55:55
20:55:58
20:56:01
20:56:05
L3HIT
L2HIT

系统内存使用(按 Mb 计算)



问答题(20%):

代码运行流程

1. application 启动之后，会在本地启动一个 Driver 进程用于控制整个流程
2. 首先需要初始化的是 SparkContext, SparkContext 要构建出 DAGScheduler,TaskScheduler
3. 在初始化 TastScheduler 的时候,它会去连接 master,并向 master 注册 Application ,master 收到信息之后,会调用自己的资源调度算法,在 spark 集群的 work 上,启动 Executor,并进行资源的分配， 最后将 Executor 注册到 TaskScheduler，到这准备工作基本完成了
4. 通过 GraphLoader.edgeListFile(sc, "file") 去加载数据源，将数据转化为 RDD,
5. DagScheduer 先按照 action 将程序划分为一至多个 job(每一个 job 对应一个 Dag)，之后对 DagScheduer 按照是否进行 shuffer,将 job 划分为多个 Stage 每个 Stage 过程都是 taskset , dag 将 taskset 交给 taskScheduler,由 Work 中的 Executor 去执行， 至于 task 交给哪个 executor 去执行，由算法来决定。

算法设计

使用 pregel 来实现 SSSP 在 PART1 已经阐述，不再赘述。

使用 PREGEL 实现 pagerank 在 PART2 已经阐述，不再赘述。

比较实验 2 和实验 3 在运行时间、空间占用方面的差异

	实验 2 Pagerank	实验 3 Pagerank
运行时间	7s	2min
瓶颈	无明显瓶颈	driver-memory
运行时间优化方式		指定 driver-memory 3g
空间占用	700Mb	4Gb
Pregel Iteration 数	20	81
Second per Iteration	0.3	1.35
每个 partition 占用了多少内存	500KB~2MB 不等	20MB 左右

结合自己的实际作图进行综合阐述

1. CACHE HIT 的比较:

   实验 2 和实验 3 的 L3 HIT 均在 0.6 到 1 波动，L2 HIT 均在 0.4 到 0.6 波动，两者相差不大。

2. CACHE MISS 的比较:

   实验 2 和实验 3 都显示出 L2 CACHE MISS 在 pregel 计算开始时出现高达 25 的峰值，在 pregel 计算的中后期进入 5 到 10 之间波动的平稳期。

   这说明内存使用率并不会显著减少 CACHE MISS，CACHE MISS 更多与数据读的频率相关。

3. 运行时间的比较:

   实验 3 的运行时间显著比实验 2 长。

   这是因为

   1. pageRank 算法在 Google 图上需要更多迭代次数才能收敛。
   2. google 图生成的 RDD 更大，进行计算、序列化、反序列化所需时间更多，在每个 iteration 上所需时间更多。

4. 内存使用的比较:

实验 3 的内存使用显著比实验 2 大。

这是因为

1. 由于 google 图生成的 RDD 更大，所需更多 driver-memory 存储 partiition 以及中间结果，实验 2 每个 partition 占用了 500KB~2MB 不等内存，实验 3 每个 partition 占用了 20MB 左右内存。
2. 实验 3 的每个 Executor 进程的内存占用比实验 2 多，以及 executor-cores 比实验 2 多。

在实验过程中遇到的难题和解决方式

配置 cache 检测工具 pcm 遇到一些问题，解决方法来自于 https://github.com/opcm/pcm/blob/master/WINDOWS_HOWTO.md

Idea 把 Spark 代码打包成 Jar 的教程如下

https://blog.csdn.net/qq_36699423/article/details/92795821

WINDOWS 系统下部署 spark 应用也有一些问题，解决方法来自于

https://www.zhihu.com/question/35973656

附录

如果遇到内存不够的情况可以使用 PartitionStrategy，有以下四种参数可选

1. RandomVertexCut
2. CanonicalRandomVertexCut
3. EdgePartition1D
4. EdgePartition2D
   本次 lab 所用机器的内存足够，所以就不使用 PartitionStrategy 了

   以下为如果需要使用 PartitionStrategy 的代码例子

```
val graph = GraphLoader.edgeListFile(sc, path = ".//file//web-Google.txt" )
    .partitionBy(PartitionStrategy.RandomVertexCut)
```

测量 cache 命中率的环境

监测使用软件：Intel® Performance Counter Monitor

`Performance Counter Monitor`（PCM）是一个由英特尔开发的，也是基于 PMU(performance monitoring unit)一个性能检测工具。

部署使用：spark-submit

Spark 的 bin 目录中的 `spark-submit` 脚本用于启动集群上的应用程序。

使用 Performance Counter Monitor 的命令

Cache 监测命令

```
pcm 0.03 -ns -nsys -csv=test.csv -- spark-submit --master spark://macor:7077  --class PagerankWiki C:\\Users\chris\Desktop\GRAPHX\out\artifacts\GRAPHX_jar\GRAPHX.jar > testlog.txt
```

从而在 testlog.txt 中为 spark 程序的输出，配合 test.csv 画出的折线图即可分析 cache miss 和内存使用率。

测量内存占用率的环境

监测使用软件：依赖 psutil 包的 python 脚本

```python
#!/usr/bin/python3
import psutil
import time
import os
import csv
import _thread
import datetime
def startspark(threadName):
    os.system("spark-submit --master spark://macor:7077  --class PagerankWiki C:\\Users\chris\Desktop\GRAPHX\out\\artifacts\GRAPHX_jar\GRAPHX.jar")

def monitoemem(threadName):
    with open("test.csv","w",newline='') as csvfile:
        writer = csv.writer(csvfile)

        for x in range(100):
            time.sleep(0.1)
            writer.writerow(( datetime.datetime.now(),psutil.virtual_memory().used))

try:
    _thread.start_new_thread( startspark, ("Thread-1",))
    _thread.start_new_thread( monitoemem, ("Thread-2",))
except:
   print ("Error: unable to start thread")

while 1:
   pass
```

在 spark 集群（本次 lab 只有一个结点）提交 PagerankWiki 的任务的命令

```
spark-submit --master spark://macor:7077  --class PagerankWiki C:\\Users\chris\Desktop\GRAPHX\out\artifacts\GRAPHX_jar\GRAPHX.jar
```

`pagerank_google` Cache 监测命令

```
pcm 0.3 -ns -nsys -csv=test.csv -- spark-submit --master spark://macor:7077  --class pagerank_google  --driver-memory 3g  --executor-memory 2g  --num-executors 3  C:\\Users\chris\Desktop\GRAPHX\out\\artifacts\GRAPHX_jar\GRAPHX.jar > log.txt
```

在 spark 集群（本次 lab 只有一个结点）提交 `pagerank google` 的任务的命令

```
spark-submit --master spark://macor:7077  --class pagerank_google  --driver-memory 3g  --executor-memory 2g  --num-executors 3 C:\\Users\chris\Desktop\GRAPHX\out\\artifacts\GRAPHX_jar\GRAPHX.jar > log.txt
```