Spark 大数据系统 Graphx 应用 垃圾回收性能分析

测试方案

1. 测试用例：
   选取 Graphx 典型的四个例子，Comprehensive Example, PageRank Example, TriangleCounting Example, ConnectedComponents Example.
2. 测试数据：
   数据来源：http://snap.stanford.edu/data/

Name	Type	Nodes	Edges	Description
soc-LiveJournal1	Directed	4,847,571	68,993,773	LiveJournal online social network

3. 测试环境

Application	Cores	Memory Per Node	Job Amount	Stage Amount	Task Amount
TriangleCounting	64	3.0GB	2	11	视情况而定

Comprehensive Exmaple 和 Connected Components 在本身应用执行时间比较短（1min 左右），此处不做统计分析

测试发现概况

1. 对于 Graphx 类型的应用，如果选取不合适的 GC 算法，则会频繁出现 OOM 和 GC 次数过多的问题
2. 之前以 LogisticsRegressionWithLBFGS 为例测试 Mllib 类的应用 GC 情况时，G1 算法并没有体现出明显的优越性，相反，Parallel GC 和 CMS 的性能要比 G1 好（在未改变 GC 算法相关参数的情况下）；但是本次 Graphx 的测试过程中，在同样不改变 GC 算法其他相关参数的情况下，G1 算法的优越性非常明显，以 TriangleCounting 为例，在使用 Parallel GC 时，会频繁出现 OOM 和 GC 次数过得问题，导致应用执行时间过长。因此，此处合理怀疑，对于不同类型的应用，甚至可能数据集不同，均有不同的适用其的 GC 算法
3. Graphx 类型的应用，如果没有采用合适的 GC 算法，会出现 task retry 的现象，由此造成的时间浪费非常严重，因此，对于同一个应用，即使使用相同的数据集，也只是会有相同的 Job 数和 Stage 数，但是不能保证其 Task 数量是一致的。
4. 对于 Graphx 类型的应用，如果没有选取合适的 GC 算法，极有可能出现应用耗时过长占用资源，导致集群崩溃.（非常明显的例子就是 PageRank，如果不采用 CMS 或者 G1，应用可以执行超过 3 小时依然不能执行结束，但是如果采用 CMS 或者 G1，应用在 10min 内就可以执行出结果）
5. 在所有外界环境均相同的情况下，改变 GC 算法，应用 GC 相对严重的 Stage 并不相同。
6. 就目前的数据反馈来看，G1 和 CMS 不相上下，但是此处存疑，因为还未涉及到 CMS 和 G1 的一些性能参数调优，不知道后续调优以后是否会有改进。

7. 当使用 CMS 时，应用执行过程中并不会出现 stage retry 的现象，但是 G1 和 Parallel GC 会出现 Stage Retry 的现象，如果在 G1 解决 Stage Retry 现象以后，其优越性会更明显。

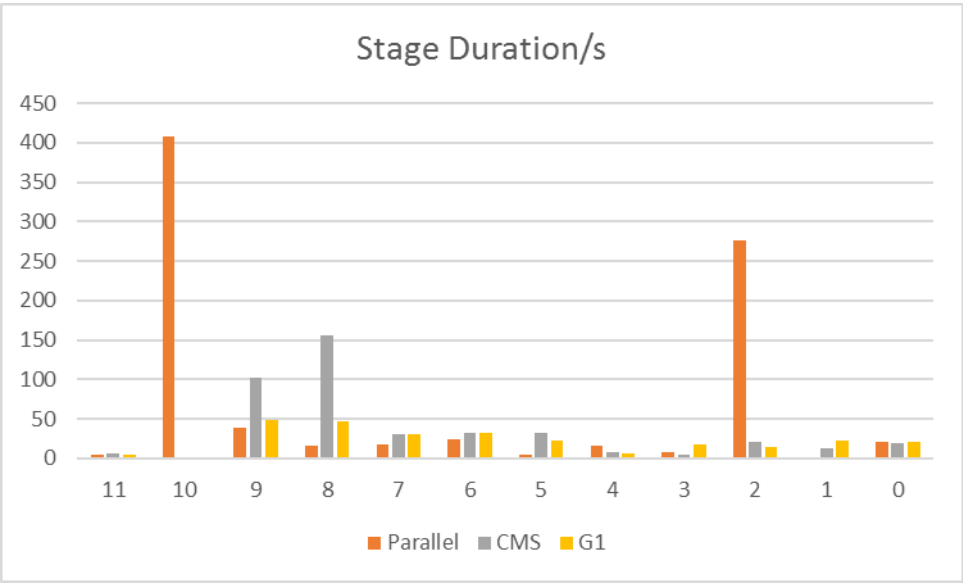
数据统计（统计 TriangleCountingExample）

1. 应用采用不同 GC 算法，执行情况统计

GC Type	Job	Stage	Task	Duration
G1	2	11	103	6.2min
CMS	2	11	90	6.6min
Parallel GC	2	11	118	25min

2. 因为 TriangleCounting 只有两个 Job，所以我们此处从 Stage 层次开始对比

Stage Id	GC Type Stage Duration/s		
	Parallel	CMS	G1
11	5	6	4
10	408	1	1
9	38	102	49
8	16	156	47
7	18	31	31
6	24	32	32
5	5	33	22
4	16	8	6
3	8	4	18
2	276	20	14
1	2	13	23
0	21	19	21



Stage Retry 现象目前只出现在当应用使用 G1 算法和 Parallel GC 算法当中，此处需要质疑的问题是，因为当使用 Parallel GC 时，stage retry 了多次，造成其运行时间的因素到底是 GC 时间过长还是 stage retry 的影响，但是从之前各个 Stage 的横纵向对比来看，主要矛盾还是 Stage 执行时间太长。

GC Type	Retry Stage	Retry Times	Duration	Operation
G1	8	1	47 s	mapPartitions at VertexRDDImpl.
	7	1	31 s	mapPartitions at VertexRDD.
	6	1	32 s	mapPartitions at GraphImpl.
	5	1	22 s	map at GraphOps.
	4	1	6 s	mapPartitions at VertexRDDImpl.
	3	1	18 s	mapPartitions at VertexRDD.
	2	1	14 s	mapPartitions at VertexRDD.
	1	1	23 s	map at GraphImpl
GC Type	Retry Stage	Retry Times	Duration	Operation
Parallel GC	10	2	6.8 min	mapPartitions at GraphImpl.
	9	2	38 s	mapPartitions at VertexRDDImpl.
	8	1	16 s	mapPartitions at VertexRDD.
	7	3	18 s	mapPartitions at GraphImpl.
	6	2	24 s	map at GraphOps.
	5	2	5 s	mapPartitions at VertexRDDImpl.
	4	2	16 s	mapPartitions at VertexRDD.
	3	2	8 s	map at GraphImpl
	2	2	4.6 min	map at GraphImpl.scala

3. 为了体现 GC 时间的比重，我们将所有 Task 的执行时间求和，将所有 Task 的 GC 时间求和，看其比重情况

GC Type	Task Amount	Total GC Time/Total Time	Total Duration/Total Time
G1	103	0.089	0.91
CMS	90	0.094	0.905
Parallel GC	118	0.213	0.786

(Total Time = Total Task Duration + Total GC Time)

对于每一个 task，我们计算了 Rate = Task GC Time /Task Duration，做出如下统计

GC Type	Rate>0.5	0.5>Rate>0.2	0.2>Rate
	Task Amount/ Total Task Amount		
G1	0	6/103	95/103
CMS	1/90	12/90	77/90
Parallel GC	7/118	40/118	65/118

各种算法下，gc 相对严重的 task 所在的 stage
其中 gc 相对严重是 Rate = Task GC Time /Task Duration 相对较大，或者 GC Time 相对较长

GC Type	StageID	Operation
G1	6	mapPartitions at GraphImpl.
	7	mapPartitions at VertexRDD.
CMS	7	mapPartitions at VertexRDD.
	8	mapPartitions at VertexRDDImpl.
	9	mapPartitions at GraphImpl.
Parallel GC	7	mapPartitions at GraphImpl.
	8	mapPartitions at VertexRDD.
	10	mapPartitions at GraphImpl.

基本可以看出，对于 TriangleCounting 来说，gc 相对严重的阶段出现在 mapPartitions at GraphImpl. 以及 mapPartitions at VertexRDD.

测试中出现的问题

1. 出现 stage skipped 的现象
   Spark Job 的 ResultStage 的最后一个 task 成功执行以后，DAGScheduler.handleTasskCompletion 方法会发送 SparkListenerJobEnd 事件：

```
                // If the whole job has finished, remove it
                if (job.numFinished == job.numPartitions) {//ResultStage所有任务都执行完毕，发送
SparkListenerJobEnd事件
                  markStageAsFinished(resultStage)
                  cleanupStateForJobAndIndependentStages(job)
                  listenerBus.post(
                    SparkListenerJobEnd(job.jobId, clock.getTimeMillis(), JobSucceeded))
                }
```

   JobProgressListener.onJobEnd 方法负责处理 SparkListenerJobEnd 事件：

```
        if (stageInfo.submissionTime.isEmpty) {//Job的Stage没有提交执行，则这个Stage和它对应的Task
会标记为skipped stage和skipped task进行统计
          // if this stage is pending, it won't complete, so mark it as "skipped":
          skippedStages += stageInfo
          trimStagesIfNecessary(skippedStages)
          jobData.numSkippedStages += 1
          jobData.numSkippedTasks += stageInfo.numTasks
        }
```

判断条件 stageInfo.submissionTime 的设置是在 Stage 被分解成 TaskSet 之后，在 TaskSet 提交到 TaskSetManager 之前进行设置

```
    if (tasks.size > 0) {
      logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" + stage.rdd + ")"
      stage.pendingTasks ++= tasks
      logDebug("New pending tasks: " + stage.pendingTasks)
      taskScheduler.submitTasks(
        new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.firstJobId, properties))
      stage.latestInfo.submissionTime = Some(clock.getTimeMillis())//设置StageInfo的submissionTime
成员，表示这个TaskSet会被执行，不会被skipped
```

其中 tasks 是一个 list，表示每一个 stage 每个 task 的描述，描述信息包括：task 所在的 stage id, task 处理的 partition, partition 所在的主机地址和 Executor id

在将 Stage 分解成 TaskSet 的时候，如果一个 RDD 已经 Cache 到了 BlockManager，则这个 RDD 对应的所有祖宗 Stage 都不会分解成 TaskSet 进行执行，所以这些祖宗 Stage 对应的 StageInfo.submissionTime.isEmpty 就会返回 true，所以这些祖宗 Stage 和它们对应的 Task 就会在 Spark ui 上显示为 skipped

2. 出现 stage retry 现象
3. 出现 spark.shuffle.FetchFailedException:Failed to Connect to /133.133.10.12:39647

   或者出现 spark.shuffle.MetadataFetchFailedException:Missing an output for shuffle X

   Shuffle 分为 shuffle write 和 shuffle read 两部分，shuffle write 的分区数由上一阶段的 RDD 分区数控制，shuffle read 的数据分区数则是由 Spark 提供的一些参数控制。

   出现 failed to connect 就是 executor lost，因为 shuffle read 的量很大，但是其分区很小，导致一个 task 需要处理的数据非常大，从而导致 JVM crash 或者长时间 gc，从而导致 shuffle 数据失败，executor 丢失。