云计算技术课程实验报告

陆徐东
519021910372
上海交通大学计算机科学与工程系
luxudong2001@sjtu.edu.cn

摘要—随着计算机技术的进步与分布式系统的发展，云计算的发展过程中衍生出一批分布式框架。这些分布式框架具有高性能、低延时和高弹性的特点，方便了系统的管理，降低了系统之间的耦合度，提高了系统的容错性，为云计算的开展带来了很大的便利。本次实验中我系统学习了 Hadoop 与 Spark 框架，学习并编写了 MapReduce、Scala 代码，并且引入 GraphX API 解决了一些简单的实际问题。

关键词—Hadoop 与 Spark 框架, MapReduce, GraphX API, PageRank 算法，PySpark 接口

I. 简介

本次实验中我完成了以下四个方面的内容：搭建华为云虚拟机 Hadoop 集群、运行 Hadoop 自带 wordcount 样例、在 Spark 中引入 GraphX API 运行 Connected Component 样例、使用 GraphX API 执行 PageRank 算法。

此外，我自己利用 MapReduce 方案学习并实现了 WordCount 程序，利用 Python 的 networkx 库可视化候选人数据，将单纯统计候选人结点入度的算法与 PageRank 算法进行比较，利用 networkx 库的 PageRank 算法与 GraphX API 对比，利用 Scala 语言手动实现了 PageRank 算法。我还学习了 spark 的 Python 接口 PySpark 的用法，利用 PySpark 运行了 networkx 库编写的程序。

II. 搭建华为云虚拟机 Hadoop 集群

A. 购买弹性云服务器与 *Java* 环境安装

此处按照实验手册需求购置了三台虚拟机，虚拟机配置如下图1所示。本次实验中选取termius软件进行远程连接。

图 1. 虚拟机配置

在 JAVA 官网下载 JDK 并解压到三台虚拟机指定位置。添加环境变量后 JAVA 生效。

```
root@master:/usr/lib/jvm/jdk1.8.0_311# java -version
java version "1.8.0_311"
Java(TM) SE Runtime Environment (build 1.8.0_311-b11)
Java HotSpot(TM) 64Bit Server VM (build 25.311-b11, mixed mode)
root@master:/usr/lib/jvm/jdk1.8.0_311#
```

B. 配置 *Hadoop* 框架

Hadoop2.7.3 框架按照教程配置 [1]。配置过程难度不大，但是按照教程配置出现两处问题。实验过程中我尝试解决。

1) 启动 hadoop，报错 Error JAVA_HOME is not set。对于这个问题，仅仅在系统 ~/.bashrc 文件添加 JAVA 环境变量还不够，需要在 config 文件中手动指定 JAVA_HOME，即在 hadoop-env.sh 文件下手动添加

```
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_311
```

2) 在 HFS 系统创建文件时报错 master:9000 failed on connection。并且此时 jps 指令在 master 机器下没有输出 NameNode。注意到教程修改 core-site.xml 过程中指定 value 为：⟨value⟩ file:/usr/local/hadoop/tmp⟨/value⟩。将其中的 “file:” 删去，即可顺利启动 NameNode。

C. *Spark* 配置

Spark 配置过程也采用教程方法 [2]。配置过程难度不大。

Spark 已经更新到 3 开头的版本，但是按照我的实验发现教程只能支持 2 开头各版本。于是选择最新的 2 开头版本 Spark2.4.8 版本进行配置。

配置过程中也出现了 JAVA 环境变量的问题，与上述 Hadoop 配置的解决方式一样，在 spark-config.sh 添加 JAVA_HOME 环境，即可解决问题。

D. 配置 *SBT*

SBT 的配置参考实验所给教程 [3]，后续实验中 Scala 独立应用程序的编写采用教程中 “Scala 独立应用程序” 这一板块。

E. 配置结果

下方代码块展示配置结果。在 master 结点下输入 jps 指令，得到如下输出：

```
root@master:/usr/local/spark# jps
1939 NameNode
2196 SecondaryNameNode
2356 ResourceManager
981 WrapperSimpleApp
2702 Jps
2655 Master
```

下方代码块展示配置结果。在 slave 结点下输入 jps 指令，得到如下输出：

```
root@slave02:~# jps
1973 NodeManager
1799 DataNode
939 WrapperSimpleApp
2190 Jps
2013 Worker
```

至此，本次实验的实验环境全部搭建完成。

III. 运行 Hadoop 自带 wordcount 样例

A. 编译 *wordcount* 样例

在本次云计算课程实验中我安装的 hadoop 框架版本为 2.7.3，所以我在执行测试样例的过程中调用的 mapreduce 工具为：/usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar。

B. 执行结果

执行 “hadoop fs -cat /output/part-r-00000” 指令打印结果。

测试样例的原始输入和执行结果如下所示：

```
root@master:~# cat test_file.txt
I am XudongLu from Shanghai Jiao Tong University. Nice to meet you!
root@master:~# hadoop fs -cat /output/part-r-00000
I 1
Jiao 1
Nice 1
Shanghai 1
Tong 1
University. 1
XudongLu 1
am 1
from 1
meet 1
to 1
you! 1
root@master:~#
```

可以看到，hadoop 自动统计了输入文本的各个单词个数，并且按照大写在前小写在后的字母顺序打印。

C. 自行学习并编写 *MapReduce* 函数

样例中直接调用了 Hadoop 自带的 WordCount 样例，虽然呈现了输出，但是对其中的工作原理我并不清楚。于是我对 MapReduce 框架进行了系统的学习，并结合相关样例利用 Java 实现了 Hadoop 自带的 WordCount 样例。

MapReduce 的简单思想就是把程序 “映射” 到不同的服务器上执行，最终在一台服务器 “规约”。实现 MapReduce 需要自定义 map 函数和 reduce 函数。在 WordCount 中定义 TokenizerMapper() 和 IntSumReducer() 两个类进行 MapReduce。

编写源代码文件 “WordCount.java”，利用如下命令执行编译、提交运行。

```
javac WordCount.java −cp $(hadoop classpath)
jar −cvf WordCount.jar −C ./ .
hadoop jar WordCount.jar WordCount /input /output
```

得到的输出将保存在 hfs 的 “/output” 文件夹下，输出结果与自带样例保持一致。

IV. 在 Spark 中引入 GraphX API 运行 Connected Component 样例

A. 编译过程

需要打包的样例为 ConnectedComponentsExample.scala 代码，是求图中的连通分量的算法。

按照实验手册中 “编写 Scala 独立应用程序” 模块进行编译。但是运行过程中报错找不到 GraphX 模块。

经过排查发现为 simple.sbt 文件出错。教程给的参考样例中不需要 GraphX API，所以没有引用相关依赖项。

修改 simple.sbt 为如下版本后，打包 jar 文件提交执行。

```
root@master:/usr/local/spark/mycode/ConnectedComponents# ls
project simple.sbt src target
root@master:/usr/local/spark/mycode/ConnectedComponents# cat simple.sbt
name:=“Simple Project”
version:=“1.0”
scalaVersion:=“2.11.8”
libraryDependencies+=“org.apache.spark”%%“spark−core”%”2.1.0”
libraryDependencies+=“org.apache.spark”%%“spark−graphx”%”2.1.0”
libraryDependencies+=“org.apache.spark”%%“spark−sql”%”2.1.0”
```

可以看到，为方便编译，引入 graphx 和 sql 两个依赖项。

B. 执行结果

通过以下两条指令打包可执行文件并提交执行。由于输出内容较长，这里只截取了部分有价值的输出以供参考。

```
root@master:/usr/local/spark/mycode/ConnectedComponents# /usr/local/sbt/
    sbt package
root@master:/usr/local/spark/mycode/ConnectedComponents# spark−submit
    ./target/scala−2.11/simple−project_2.11−1.0.jar
21/12/29 15:42:59INFO scheduler.DAGScheduler: Job 7 finished: collect at
    ConnectedComponentsExample.scala:67,took 0.209016 s
(justinbieber,1)
(matei_zaharia,3)
(ladygaga,1)
(BarackObama,1)
(jeresig,3)
(odersky,3)
```

V. 使用 GraphX API 执行 PageRank 算法

这一项工作为本次实验的重点与难点。在该块的实验中我也进行了一些简单的拓展，具体工作如下所示。

A. PageRank 算法原理

PageRank 的计算充分利用了两个假设：数量假设和质量假设 [4]。步骤如下：

1) 在初始阶段：网页通过链接关系构建起 Web 图，每个页面设置相同的 PageRank 值，通过若干轮的计算，会得到每个页面所获得的最终 PageRank 值。随着每一轮的计算进行，网页当前的 PageRank 值会不断得到更新。
2) 在一轮中更新页面 PageRank 得分的计算方法：在一轮更新页面 PageRank 得分的计算中，每个页面将其当前的 PageRank 值平均分配到本页面包含的出链上，这样每个链接即获得了相应的权值。而每个页面将所有指向本页面的入链所传入的权值求和，即可得到新的 PageRank 得分。

B. Python networkx 库

NetworkX 是一个 Python 包 [5]，用于创建，操作和研究复杂网络的结构，动态和功能。

C. 数据集可视化

实验中我们采用了 Wikipedia vote network [6] 数据集，进行投票统计。在进行实际统计之前，我用 Python 的 networkx 包对数据集的部分数据进行了可视化。可视化结果如图2所示。

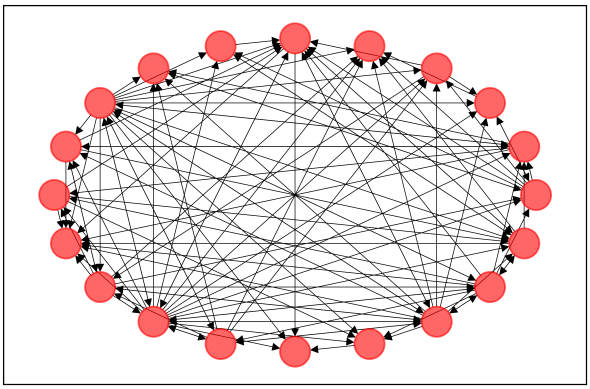

图 2. Wikipedia vote network

可以看到，Wikipedia vote network 为一个有向图，其中某个使用者会对另外一个使用者进行投票。

D. Scala 语言实现相关算法

当我们日常进行投票统计的过程中，对于每个节点我们只会把他的入度进行简单相加，来判断其有几票。但是 PageRank 算法提供了一个不一样的观点。该算法把每个人的票数进行权重加和，并考虑了候选人本身投票带来的影响力变化。

本次实验过程中分别对上述 PageRank 和入度相加两种算法进行了实现与比较。Scala 语言采用 graphx 自带的 PageRankExample.scala 进行改写。

```
// Load the edges as a graph
val graph = GraphLoader.edgeListFile(sc, "file:///usr/local/spark/mycode/
    ConnectedComponents/src/main/scala/data/graphx/followers.txt")
// Run PageRank
val ranks = graph.pageRank(0.0001)

ranks.vertices.sortBy(_._2, false).collect.foreach(println)
println("================indegrees==============")
// indegrees
graph.inDegrees.sortBy(_._2, false).collect.foreach(println(_))
```

此处在对 PageRank 算法排序统计完成之后，对每个结点的入度也进行了统计。考虑到 PageRank 算法可能会统计到一些不是候选人的结点，于是在输出后对结果进行筛选整理。但并没有发现这样的结果存在，这一定程度上也说明了 PageRank 算法的合理性。排序统计的结果如下。以下 $(\cdot,\cdot)$ 括号中左侧代表选出的候选人，右侧代表统计量的值。

```
// pagerank on the left and indegree on the right
(4037,32.78074239389385) (4037,457)
(15,26.18174657476919) (15,361)
(6634,25.518550140728546) (2398,340)
(2625,23.361004685170897) (2625,331)
(2398,18.559437057563535) (1297,309)
(2470,17.957604768297593) (2565,274)
(2237,17.76401205997604) (762,272)
(4191,16.135404511533686) (2328,266)
(7553,15.436932186579376) (5254,265)
(5254,15.297497713729927) (3352,264)
(2328,14.508487421589328) (4191,259)
(1186,14.483277672215047) (2066,254)
(1297,13.843623570397671) (1549,245)
(4335,13.778724193287324) (3089,244)
(7620,13.746244183899307) (2535,232)
(5412,13.651704171117718) (737,231)
(7632,13.572881154938925) (4335,228)
(4875,13.331509917939387) (3456,223)
(6946,12.866012365200149) (5412,219)
(3352,12.691867222435459) (3334,217)
```

可以看到，PageRank 和 Indegree 的统计结果大致相同但并不相等，说明 PageRank 考虑到了单纯入度相加之外的情况。而这种情况避免了直接将入度相加带来的统计偏差。

E. Python networkx 实现相关算法

随后我用 Python 自带的 networkx 库在本地 Windows 机器上进行了对比试验，对比不同 API 的差异。

networkx 库也自带了 PageRank 算法 API 可以直接调用，此外我自己编写相关函数统计了每个结点的入度情况。统计结果如下所示。

```
// pagerank on the left and indegree on the right
(4037, 0.00014064231339238245) (4037,457)
(2470, 0.00014060776588022602) (15,361)
(15, 0.00014060510166506124) (2398,340)
(2237, 0.00014060034980519932) (2625,331)
(1186, 0.00014059627279752385) (1297,309)
(2625, 0.00014058984322352 13) (2565,274)
(665, 0.00014058655518804143) (762,272)
(6774, 0.00014058363585707306) (2328,266)
(8293, 0.00014058167769766706) (5254,265)
(2654, 0.00014058155365484 72) (3352,264)
(4191, 0.00014058089420514 63) (4191,259)
(2285, 0.00014058025950313) (2066,254)
(5254, 0.00014057998786519895) (1549,245)
(214, 0.00014057968962860227) (3089,244)
(6634, 0.00014057855293489411) (2535,232)
(2328, 0.00014057732818488554) (737,231)
(4875, 0.00014057682501381893) (4335,228)
(28, 0.0001405766855372637) (3456,223)
(2398, 0.00014057560389559) (5412,219)
(4261, 0.00014057432176522633) (3334,217)
```

可以看到，由于实现方式差异，两个 API 统计得到的入度排序相同，但是 PageRank 结果有较大不同，前 20 个数据里只有 12 个数据完全相同。

F. Scala 手动实现 Scala 算法

Scala 语言为函数式编程，实现起来较为麻烦。此处依据 Spark 自带样例修改。

定义页面初始分为 1。为计算每条边权重权重，定义 *val* 为节点出度，边的权值设为 $\frac{1}{OutDegree}$，利用 *val* 的最终结果表示节点得分。

按照上述算法利用 aggregateMessage 进行消息传递和消息累加与整合，迭代 100 次获得最终结果。按照最终得分排序打印如下：

```
// pagerank
(4037,0.0019237982657767743)
(15,0.0015365855168042284)
(6634,0.0014977469730989847)
(2625,0.0013711426241683408)
(2398,0.0010892770092432108)
(2470,0.0010538408679927971)
(2237,0.001042506027571294)
(4191,9.469774364547136E−4)
(7553,9.060053264184156E−4)
(5254,8.97808539978934E−4)
(2328,8.515252441644098E−4)
(1186,8.499695101505235E−4)
(1297,8.12516620755299E−4)
(4335,8.08725830006545E−4)
(7620,8.067708375404808E−4)
(5412,8.012736604561829E−4)
(7632,7.966080827579903E−4)
(4875,7.824404938189794E−4)
(6946,7.551356668125029E−4)
(3352,7.44919203702809E−4)
```

可以看到，实验结果与 Spark 自带样例完全一致，证明代码逻辑的正确性。

VI. PySpark 接口使用

在以上的实验中，我们使用的都是 spark 的 JAVA 接口，spark 也存在 Python 接口，叫做 PySpark。

A. PySpark 介绍

PySpark [7] 是 Python 中 Apache Spark 的接口。它不仅允许用户使用 Python API 编写 Spark 应用程序，而且还提供了 PySpark shell，用于在分布式环境中以交互方式分析数据。

B. networkx 库的 PySpark 版本编写

我将使用 networkx 库的 Python 脚本修改为能运用 PySpark 的版本。并且在分布式环境下运行。

编写逻辑和在本地运行 Python 没有差异。只是需要将读写文件的代码改为 Spark 指定接口。

```
rdd_file = sc.textFile(
    "file:///usr/local/spark/mycode/ConnectedComponents/src/main/scala/
        data/graphx/followers.txt").collect()
```

这个指令将 txt 文本中的内容读取为一个字符串赋值给 rdd_file，代码的其余部分的实现与正常 Python 脚本无差别。

C. 实验结果展示

Python 脚本的执行依旧需要 spark-submit 提交。执行如下命令运行脚本：

```
root@master:~# spark−submit −−master local[2] −−num−executors 2 −−
    executor−memory 1G ~/spark−nx.py
```

脚本输出与本地执行的 Python 程序没有差别。

VII. 实验总结

在本次云计算技术课程实验中，我系统学习了 Hadoop 与 Spark 框架，学习了 MapReduce 原理，手动实现 Scala 算法，并且引入 GraphX API 解决了一些简单的实际问题。此外我可视化了数据集的部分内容，学习了一个新的 Python 图处理 API networkx，并且用其与 GraphX API 进行对比。此外我还学习了 Spark 的 Python 接口 PySpark，更进一步扩展了知识面。

参考文献

[1] http://dblab.xmu.edu.cn/blog/1177-2/

[2] http://dblab.xmu.edu.cn/blog/1714-2/

[3] http://dblab.xmu.edu.cn/blog/1307-2/

[4] https://blog.csdn.net/hguisu/article/details/7996185

[5] https://networkx.org/documentation/stable/index.html

[6] http://snap.stanford.edu/data/wiki-Vote.html

[7] https://spark.apache.org/docs/latest/api/python/index.html