Experiment 8/9

517030910356 谢知晖

目录

1 实验准备

1.1 实验环境

本次实验环境依旧为 Ubuntu 平台 (版本号 18.04) 下的 Python(版本号 2.7)。

本次实验中，我配合 1.8.0_181 版本的 java，搭建了 Linux 上的 Hadoop 环境 (版本号 2.2.0)。尽管对于较旧的 Hadoop 版本会时常出现"illegal access" 的 WARNING，但由于不影响实现效果，我没有更换 java 和 Hadoop 的版本。

1.2 实验目的

实验中，我们将接触到 Hadoop 这一分布式系统的基础架构，进而开发简单的分布式程序。

在实验 8 中，我们首先接触到了 Hadoop 本身，以及面向大数据并行处理的计算模型 MapReduce，尝试了简单地在 Hadoop 框架内使用 MapReduce 模型运行迭代运算。

而在实验 9 中，我们将自己实现 MapReduce 的模型，并使用 Hadoop Streaming 框架让我们的 Python 代码在 Hadoop 集群上运行。

两次实验都围绕 MapReduce 这个中心模型进行，让我们能够更清楚地了解 Hadoop 在支持高效并行计算的基础上，是如何将任务进行规划，以使其能够充分利用 Hadoop 的并行处理优势的。

1.3 实验原理

由于在之前的实验中我们已经接触到了并行计算并介绍了相关原理，在此我便不再复述。

由于本次的实验主要围绕 Hadoop 框架以及 MapReduce 和 Hadoop Streaming 两种实现方法，故我将主要介绍这三方面的内容。

另外，在 Exp9-2 中接触到了 PageRank 的相关算法，我也会加以介绍。

1.3.1 Hadoop 基础

Hadoop 是 Apache 下一个开源子项目，实现了一种分布式系统基础架构。它的设计思想来源于 Google 的 GFS 和 MapReduce 这两篇学术文章。

Hadoop 框架最核心的设计是 HDFS、MapReduce、YARN 和 Common。其中，HDFS 实现了分布式文件系统，让用户可以以流形式访问文件系统中，为海量的数据提供了存储。MapReduce 为海量的数据提供了计算方法。而 YARN 和 Common 分别为 Hadoop 的资源调度系统以及底层支撑组件，共同支持分布式系统的稳定运行。

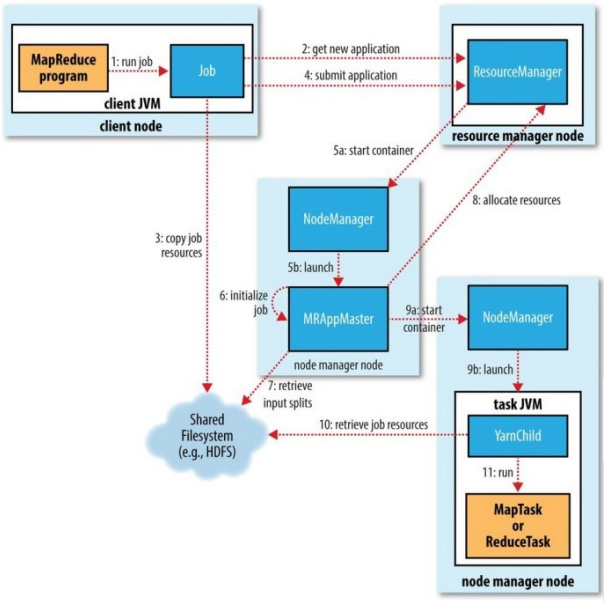
1.3.2 MapReduce

MapReduce 的最主要目的是将一个复杂计算分解成一个或者多个 Map-Reduce 步骤，以发配给并行处理系统执行。其大致过程可划分如下:

Map 从给定的一行中提取出关注的数据，输出相应格式的键值对

Sort&Shuffle 对 Map 的输出按照 key 进行排序

Reduce 在排好序的对中过滤、聚合等



MapReduce 工作流程

可以看出，实际的原理并不复杂。其核心在于 Shuffle，它链接了 Map 与 Reduce。关于 Shuffle 的知识比较繁杂，我仅能了解它描述了 Map task 的输出到 Reduce task 的输入这个过程，并在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗，且减少磁盘 IO 对 task 执行的影响。

1.3.3 Hadoop Streaming

Hadoop Streaming 是一种编程工具，它支持用任何编程语言来编写 mapreduce 的 map 函数和 reduce 函数。

由于 Hadoop Streaming 使用 Unix 标准流作为 Hadoop 和应用程序之间的接口，所以我们可以借助它使用任何编程语言通过标准输入/输出来写 MapReduce 程序，这无疑给我们带来了极大的便利。

1.3.4 PageRank

PageRank 是一种由根据网页之间相互的超链接计算的技术而作为网页排名的要素之一。它的实现基于以下简单的原理:

1. 如果一个网页的入链越多，它就越重要
2. 如果一个网页被越重要的网页所指向，它也就越重要

我们将这个重要程度定义为 PageRank。在随机游走模型中，它可以由以下公式得出:

$$R(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{R(v)}{N_v}$$

公式由两部分组成，一部分是直接随机选中的概率 $\frac{1-d}{N}$，另一部分是从只想它的网页顺着链接浏览的概率。

2 实验过程

由于本次实验中的大部分核心内容都被隐藏在 Hadoop 的 API 中，我尽可能地将实验的基本过程讲解的更为详细。

2.1 Exp8-1

本实验中，我们仅需执行 Hadoop 中 examples 里的 Pi，并尝试使用不同的 Map task 执行次数和在每次 Map 中投掷的 Samples 数量来计算 Pi 的值。

2.1.1 实验步骤

我们只需要执行命令：

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.2.0.jar pi
                                                    <nMaps>  <nSamples>
```

即可实现 Pi 值的估计。取不同的 nMaps 和 nSamples 值，将会得到不同的结果，同时所用时间也不相同。

2.1.2 实验结果

我得到的部分统计结果如下：

Number of Maps	Number of samples	Time(s)	π
2	10	25.923	3.8
5	10	27.84	3.28
10	10	38.238	3.2
2	100	22.574	3.12
10	100	37.981	3.148

Pi 值的计算 (1)

不难发现，Map task 执行次数和在每次 Map 中投掷的 Samples 数量增加时都会提高 Pi 值得准确度。

再对结果进行分析。比较第一行和第四行以及第三行和第五行，可以发现当 nSamples 的值变化时，计算用时并没有发生很大的变化，甚至当 nSamples 增大时，计算用时反而减小了。

针对上述现象，我再进行了如下几个实验，得到以下结果:

Number of Maps	Number of samples	Time(s)	π
10	1000	34.715	3.1408
10	10000	35.613	3.1412
10	100000	32.68	3.141552
10	1000000	37.968	3.1415844
10	10000000	45.081	3.14159256

Pi 值的计算 (2)

在 nSamples 的取值较小时，nSamples 的增大并不会增加太多的用时，而当 nSamples 达到 10^6 级别时，用时开始随着 nSamples 的增加而增加。这是可以理解的，因为应用 Hadoop 计算 Pi 的开销主要在于 Map 的分配。各个 Map 并行运行，而由于计算能力存在余量，在 nSamples 较小时不会达到饱和。

2.2 Exp8-2

2.2.1 实验步骤

通过 Exp8-1 中的分析，我发现 nMap 的增加是没有很大必要的，nSamples 的值对于 Pi 的精确度更为重要，我仅仅在 nMap=10 时便能计算到精确到小数点后 5 位的值。这是为什么呢？

查找资料可知，Hadoop 的 example 中的 Pi 计算是通过 Quasi-Monte Carlo 算法进行的，这是一种采用大量采样的统计学方法，属于数据密集型的工作。

值得注意的是 Hadoop 并不适合做计算密集型的工作[1]，尤其是下一步计算依赖于上一步的计算结果的时候。但在这里，我们的计算是并行的而非连续关联的，因而能够只借助 Map 的并行优势进行运算。

为了得到更为精确的 Pi 值，我在之后再进行了一系列实验。

2.2.2 实验结果

对于更大的 nSamples，计算得到的 Pi 值越来越接近于真实值。

Number of Maps	Number of samples	Time(s)	π
10	10000000	45.081	3.14159256
10	100000000	87.275	3.141592744
10	1000000000	312.448	3.1415926644

Pi 值的计算 (3)

得到的结果十分符合我的预期。在 nSamples 增大时，Pi 越来越精确，且用时并没有随其线性变化，说明 Map 起了作用。

2.3 Exp9-1

实验 9 中，我们开始进行 MapReduce 的实践。Exp9-1 便是这样一个简单的例子，让我们能够完成一个统计各个首字母的平均单词长度。

2.3.1 实验步骤

任务被分为 Mapper，Sort 和 Reducer 三部分。Mapper 负责将原始文本切分为一个个单词，在通过 Sort 后由 Reducer 处理得到计数结果。Sort 的实现并不需要我们完成。

Mapper Mapper 的实现非常简单。它仅仅实现了文本的切分，并输出键值对形式的 word-count。

```
import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()

    for word in words:
        print '%s\t%s' % (word, 1)
```

[1]进一步的讨论在实验感想中呈现

在之后的处理过程中，Sort 将基于键 (word) 来对键值对进行排序和分组。

Reducer 由于键值对已经得到分组，而首字母相同的 word 将排序在相邻的组中，故而在遍历时，我们能对所有出现的首字母进行统计。

首先，进行相关参数的初始化。

```
current_start = None
current_start_count = 0
total_length = 0
count = 0
word = None
```

其中 current_start 是当前统计的首字母，current_start_count 是当前统计的首字母的次数，total_length 是首字母为当前统计的首字母的所有单词的总长度，word 和 count 用来读取输入的数据。

然后，读入数据。当读到新的首字母时，输出上一个首字母的统计结果，否则将当前读到的 word 和 count 纳入当前首字母的统计当中。

```
for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)

    try:
        count = int(count)
    except ValueError:
        continue

    if current_start == word[0]:
        current_start_count += count
        total_length += len(word)
    else:
        if current_start:
            print '%s\t%s' % (current_start, total_length / float(current_start_count))
        current_start_count = count
        current_start = word[0]
        total_length = len(word)

if current_start == word[0]:
    print '%s\t%s' % (current_start, total_length / float(current_start_count))
```

最终得到的输出是每个首字母和其平均长度的键值对。

2.3.2 实验结果

测试输入"we become what we do"，得到了正确结果。

```
hduser@fffffarmer-virtual-machine:~/experiment/src$ echo "we become what we do"  | ~/experiment/src/mapper.py  | sort  -k1,1| ~/experiment/src/Myreducer.py
b       6.0
d       2.0
w       2.66666666667
```

2.4 Exp9-2

本次实验中，我们将对 Hadoop Streaming 进行实践，实现 PageRank 的计算。由于 PageRank 的计算需要进行多次迭代，十分符合 Hadoop Streaming 的特性，将计算部署在 Hadoop

上也理所应当。

2.4.1 实验步骤

实验中测试的 Input 的格式如下：

ID	PR	Link ID
1	0.25	2 3 4
2	0.25	3 4
3	0.25	4
4	0.25	2

可以发现，某些网页没有指向它的入链，这应当在之后的过程中引起重视。

Mapper Mapper 的输入格式和 Input 的格式相同，输出为"LinkID averPR ID" 的键值对。

```
import sys

for line in sys.stdin:
    line = line.strip()
    if len(line) > 0:
        words = line.split()
        fromID = int(words[0])
        pr = float(words[1])
        toID = words[2::]

        for to in toID:
            print '%s\t%s\t%s' % (to, pr / len(toID), fromID)
```

这里，每个网站的 PageRank 被均分给每个出链。我将原本的入链-出链对转化为出链-入链对，而将累加工作交个 Reducer 来完成。

Reducer Reducer 接受 Mapper 的输入，即"LinkID averPR ID" 的键值对，输出与 Input 格式相同的结果。

首先，我们进行相关的初始化，将阻尼因子 d 设为 0.85，并创建存放 Web Graph 和 PageRank 的两个字典。

```
d = 0.85
ID = {}
PR = {}
```

然后，我们读取输入，根据公式 $R(u) = \frac{1-d}{N} + d\sum_{v \in B_u} \frac{R(v)}{N_v}$，对输入进行累加。

```
for line in sys.stdin:
    line = line.strip()
    if len(line) > 0:
        words = line.split()
        toID = int(words[0])
```

```
            pr = float(words[1])
            fromID = int(words[2])

            if not toID in PR:
                PR[toID] = 0

            if not fromID in ID:
                ID[fromID] = []

            ID[fromID].append(toID)
            PR[toID] += d * pr
```

值得注意的是，正如之前提到的，某些网页没有指向它的入链，这意味着在读取输入时将不会重新计算它的 PageRank 值。尽管在最后这个值确实不会改变且为 $\frac{1-d}{N}$，但在第一次 MapReduce 过程中，初始输入可能与 $\frac{1-d}{N}$ 不同。因此我们需要在读取输入之后的过程中进行修正。

```
    for key, value in ID.items():
        if key not in PR:
            PR[key] = (1 - d) / len(ID)

        else:
            PR[key] += (1 - d) / len(ID)
```

最后，我们可以进行输出，保证格式与 Input 的格式相同。

```
    for key, value in ID.items():
        toIDs = ''
        for toID in value:
            toIDs += str(toID)
            toIDs += ' '

        print '%s\t%s\t%s' % (key, PR[key], toIDs)
```

Linux 脚本 我通过一个 Linux 脚本来完成 MapReduce 的迭代。

首先，command 中定义了 Streaming 的主体，它包含了 MapReduce 过程中的 Python 脚本文件。mv 等变量定义了 HDFS 的一些操作。

```
    command='hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.2.0.jar \
        -file Mymapper.py \
        -mapper "Mymapper.py" \
        -file Myreducer2.py \
        -reducer "Myreducer2.py"'
    mv='hadoop fs -mv '
    rm='hadoop fs -rm -r '
    cp2local='hadoop fs -copyToLocal '
    input='tempinput'
```

然后是 MapReduce 的迭代过程。过程的迭代次数为命令行输入，每次迭代将结果保存在 tempoutput 中，然后将下一次的输入设置为本次的输出。

为了便于查看结果，我将每次的结果都保存在一个文件夹中。

```
for ((i=1;i<$1+1;i++));
do
    echo "Processing $i"
    output="tempoutput_$i"
    eval "$command -input $input/* -output $output"
    input=$output
    eval "$rm $input/_SUCCESS"

    eval "$cp2local $output/* /home/hduser/result/$i"
done
```

2.4.2 实验结果

现在我可以使用 MapReduce 来执行 PageRank 的运算。

```
18/11/22 13:52:48 INFO mapreduce.Job: Job job_1542815784210_0040 running in uber mode : false
18/11/22 13:52:48 INFO mapreduce.Job:  map 0% reduce 0%
18/11/22 13:53:00 INFO mapreduce.Job:  map 50% reduce 0%
18/11/22 13:53:01 INFO mapreduce.Job:  map 100% reduce 0%
18/11/22 13:53:10 INFO mapreduce.Job:  map 100% reduce 100%
18/11/22 13:53:11 INFO mapreduce.Job: Job job_1542815784210_0040 completed successfully
```

某次 MapReduce 的执行

application_1542815784210_0043	hduser	streamjob6132751773764153137.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:55:07 GMT	N/A	RUNNING	UNDEFINED
application_1542815784210_0042	hduser	streamjob2276244958559281821.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:54:24 GMT	Thu, 22 Nov 2018 05:54:55 GMT	FINISHED	SUCCEEDED
application_1542815784210_0041	hduser	streamjob4835241434176740849.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:53:31 GMT	Thu, 22 Nov 2018 05:54:06 GMT	FINISHED	SUCCEEDED
application_1542815784210_0040	hduser	streamjob5649492541607101374.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:52:39 GMT	Thu, 22 Nov 2018 05:53:10 GMT	FINISHED	SUCCEEDED
application_1542815784210_0039	hduser	streamjob1037764610254415320.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:51:57 GMT	Thu, 22 Nov 2018 05:52:27 GMT	FINISHED	SUCCEEDED
application_1542815784210_0038	hduser	streamjob8798457390378483875.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:51:12 GMT	Thu, 22 Nov 2018 05:51:42 GMT	FINISHED	SUCCEEDED
application_1542815784210_0037	hduser	streamjob1170398906347539953 2.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:50:25 GMT	Thu, 22 Nov 2018 05:50:56 GMT	FINISHED	SUCCEEDED
application_1542815784210_0036	hduser	streamjob840777674830623508.jar	MAPREDUCE	default	Thu, 22 Nov 2018 05:49:40 GMT	Thu, 22 Nov 2018 05:50:10 GMT	FINISHED	SUCCEEDED

Streaming

我进行了 20 次迭代，发现在第 7 次左右，结果开始趋于稳定。而迭代 20 次后，最终的结果与理论值十分接近。

```
hduser@fffffarmer-virtual-machine:~$ cd result/
hduser@fffffarmer-virtual-machine:~/result$ ls
1  10  11  12  13  14  15  16  17  18  19  2  20  3  4  5  6  7  8  9
hduser@fffffarmer-virtual-machine:~/result$ cat 5
1       0.0375  2 3 4
2       0.380087740885  3 4
3       0.205733649088  4
4       0.376678610026  2
hduser@fffffarmer-virtual-machine:~/result$ cat 7
1       0.0375  2 3 4
2       0.374980507861  3 4
3       0.204653272872  4
4       0.382866219267  2
hduser@fffffarmer-virtual-machine:~/result$ cat 20
1       0.0375  2 3 4
2       0.373246900331  3 4
3       0.206753592081  4
4       0.382499507589  2
```

迭代结果

3 实验感想

实验 8 和实验 9 的实现并不算困难，但在实践中我还是遇到了许多疑惑。其中最大的疑惑便在于 Hadoop 并行运算的处理细节。

3.1 Hadoop 特性

在之前我提到 Hadoop 并不适合进行计算密集型的工作。真的是这样吗? 如果是，为什么呢?

事实上，这样的说法是片面的。通过了解 Hadoop 的相关知识可以知道，Hadoop 本身是有对计算密集型工作的相关支持的，具体表现为 Hadoop 的数据本地化特性。

对于大规模的数据处理，一般的高性能计算方法是将作业分散到集群的各台机器上，而这些机器访问存储区域网络所组成的共享文件系统。如果节点需要访问的数据量变的庞大，很多计算节点将因为网络带宽的瓶颈而不得不闲下来等数据。

反映到 Hadoop 上，Hadoop 通过尽量在计算节点上存储数据来实现数据的本地快速访问，并通过显示网络拓扑结构来保留网络带宽。

因此 Hadoop 并非不适合进行计算密集型的工作。只是带宽存在局限，在之前的 Pi 值计算时并不能完全发挥 Hadoop 并行运算的优势。这提醒着我还需了解更多的 Hadoop 技术细节。