

**课 程 实 验 报 告**

**课程名称： 大数据分析**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期：**

**计算机科学与技术学院**

**目录**

[实验一 wordCount算法及其实现 1](#_Toc57053298)

[**1.1实验目的** 1](#_Toc57053299)

[**1.2 实验内容** 1](#_Toc57053300)

[**1.3 实验过程** 1](#_Toc57053301)

[1.3.1 编程思路 1](#_Toc57053302)

[1.3.2 遇到的问题及解决方式 1](#_Toc57053303)

[1.3.3 实验测试与结果分析 1](#_Toc57053304)

[**1.4 实验总结** 2](#_Toc57053305)

# 实验一 wordCount算法及其实现

## **1.1实验目的**

1、理解map-reduce算法思想与流程；

2、应用map-reduce思想解决wordCount问题；

3、（可选）掌握并应用combine与shuffle过程。

## **1.2 实验内容**

提供9个预处理过的源文件（source01-09）模拟9个分布式节点，每个源文件中包含一百万个由英文、数字和字符（不包括逗号）构成的单词，单词由逗号与换行符分割。

要求应用map-reduce思想，模拟9个map节点与3个reduce节点实现wordCount功能，输出对应的map文件和最终的reduce结果文件。由于源文件较大，要求使用多线程来模拟分布式节点。

学有余力的同学可以在map-reduce的基础上添加combine与shuffle过程，并可以计算线程运行时间来考察这些过程对算法整体的影响。

提示：实现shuffle过程时应保证每个reduce节点的工作量尽量相当，来减少整体运行时间。

## **1.3 实验过程**

### 1.3.1 编程思路

本实验是需要使用MapReduce的算法思想来完成WordCount的内容。MapReduce有这么几个算法步骤：

第一步是分割数据，也即map步骤，需要将大量的输入数据分割成一些小的块，这些小块可以并行处理。在WordCount中是需要将source01~09的文件中的单词拆分成单独的键值对。在Python中，我们可以使用with open来打开文件，并使用readlines方法逐行读取文件内容。然后可以使用字符串的split方法将每一行拆分成一个单词列表，并将这些单词存入一个数组中。最后，我们需要将这些单词按照键值对的格式写入对应的Map文件中，以便于Reduce过程的处理。再逐个按照键值对的格式写入对应的map文件。

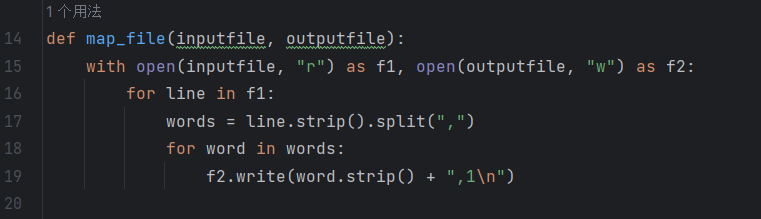
在map.py程序中，实现上述功能的代码如下：

图1.1读取文件

第二步是合并，也即combine步骤，Combine步骤通常用于一些可以在本地进行合并的场景，比如WordCount算法中对每个单词出现次数进行计数，就可以在Map阶段使用Combine步骤对每个Mapper的输出进行本地合并，减少Reduce阶段需要处理的数据量。在本实验中，我们对map处理后的文件，将里面含有相同键的单词进行统计次数，并将结果写入combine01~09的文件中。

在combine.py程序中，实现上述功能关键代码如下：

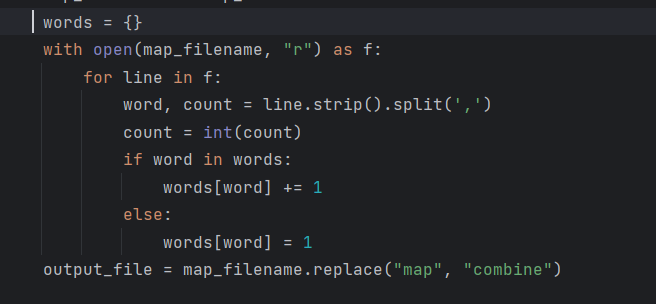


图1.2将本地先进行合并

第三步是分组，也即shuffle步骤，shuffle的作用是将combine阶段的中间结果按照键进行重新分配和排序，以便于Reduce阶段对数据的处理。在WordCount中是需要通过某种分配方式，让combine01~03的文件中单词尽量均匀的分配给后续处理的节点。在这里我们使用hash()方法来计算其哈希值，并且余3去判断应该分配到哪个节点。

在shuffle.py程序中，实现上述功能关键代码如下：

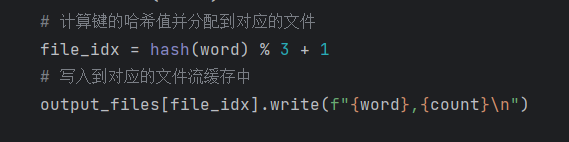


图1.3 shuffle步骤哈希分配结点

在上述程序处理完毕后，shuffle的文件大小基本平均，如下图：

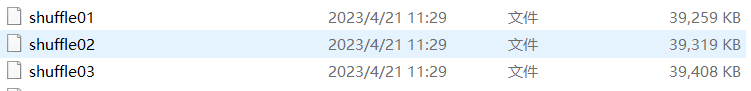


图1.4 分配后的文件大小

第四步是归约，也即reduce步骤，在reduce操作中，Reduce节点对每个键对应的值进行合并和计算，从而得到最终的输出结果。我们在这里需要将combine操作后的三个文件合并统计，采用多线程的方法调用python的threading库，在本实验中经过对比，采用ThreadPoolExecutor可以很大的减少程序需要花费的时间，帮助我们完成任务。

在reduce.py程序中，实现上述功能关键代码如下：

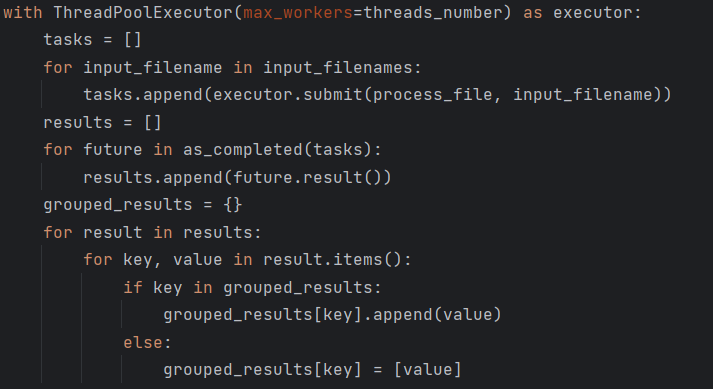


图1.5多线程过程

最后为方便比较算法消耗时间，在主体部分的两端加上time.time()来获取时间，相减得到运行时间。并且在main.py文件中，通过os库中的system方法让上述四个部分的程序按照顺序执行。

如下是main.py文件。

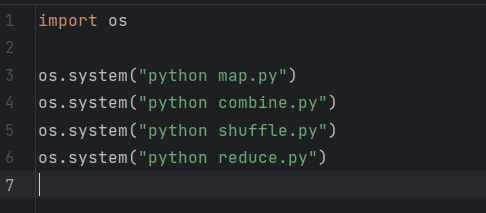


图1.6 main文件运行四个步骤

### 1.3.2 遇到的问题及解决方式

问题一，**如何将单词均匀分配给三个结点来合并统计，减少reduce处理的复杂度。**

一开始我是打算根据单词首字母直接将a-i、j-r、s-z这样直接分配给reduce的结点，但是这样处理没有考虑到部分首字母的单词比较多，最终还是可能会导致某个结点数据较大，延缓整个处理过程。所以我们需要一个更好的，更公平的方式来去分配这些工作。而python中的hash()函数可以很好的帮助我们。哈希函数将输入映射到一个固定大小的输出，该输出通常是一个整数，可以作为该python对象的唯一标识或者索引。对于本实验涉及到的单词，因为他们是不可变的对象，因此对于每一次程序运行时，他们的哈希值是固定不变的。通过对每个余三再加一的方式，分配好每个单词对应的处理结点。最后得到的三个文件大小十分相近，如下图所示：

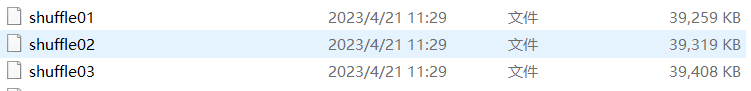


图1.7 shuffle操作后三个文件大小

问题二，**如何进行多线程方式来处理数据，而不是简单的for循环来进行处理。**

在第一次完成实验中，我采用的是for循环方式将每个source文件的单词先统计，最后直接将每个合并下来，这样是不符合实验要求，没有体现出MapPreduce的算法思想。着重在于，要实现模拟9个map节点与3个reduce节点实现wordCount功能，采用多线程方式来处理数据。这里我使用了python中的ThreadPoolExecutor库以及threading库，前者可以用来执行一组函数或方法，每个函数或方法都可以在一个独立的线程中执行。当有多个任务需要执行时，可以将这些任务提交给 ThreadPoolExecutor，它会自动将它们分配到可用的线程中执行，直到所有任务都完成为止，并且可以十分方便的控制其中的参数，例如线程池大小，线程池中线程最大数量，任务队列大小等。后者通过threading.Thread可以让我们实现线程，也即独立的指令序列，可以与程序中的其他线程并发运行。

相关程序如下图：

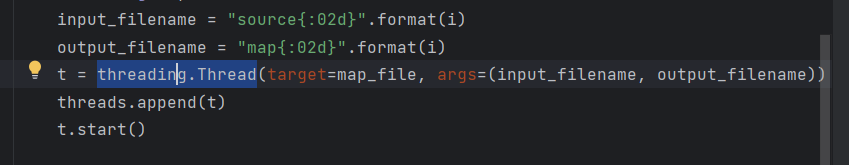
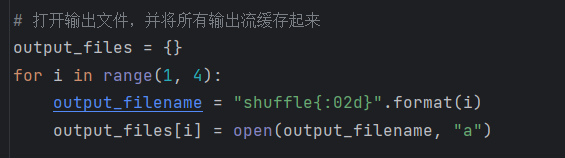


图1.8实现多线程操作

问题三，**在shuffle处理中，文件生成过慢，大概每个文件生成速度为6KB每秒，初版代码完成处理时间为10分钟，处理缓慢的原因是什么，该如何解决？**

原因是在shuffle处理中，需要同时写入大量数据，频繁地打开和关闭文件会降低程序效率。可以将所有的文件操作放到一个上下文管理器中，同时将每个文件的输出流缓存起来，统一处理写入操作。而如果我们采用使用缓冲区，可以减少写入操作的次数，从而提高程序的效率。



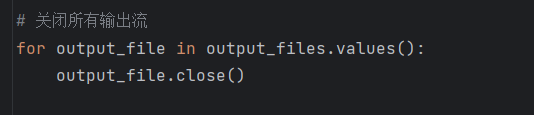


图1.9 将所有文件缓存统一操作

改进后shuffle处理步骤时间大幅度减少，从原来的十几分钟降至5秒内，如下图所示：



图1.10 改进后运行时间大幅度减少

### 1.3.3 实验测试与结果分析

执行main.py文件，得出各部分的文件处理结果，以及处理时间。

各部分处理时间，如下图所示：

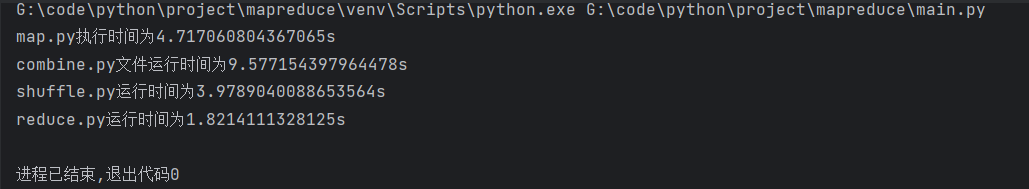


图1.11 各部分处理时间

map.py处理后，共9个文件，如下图所示：

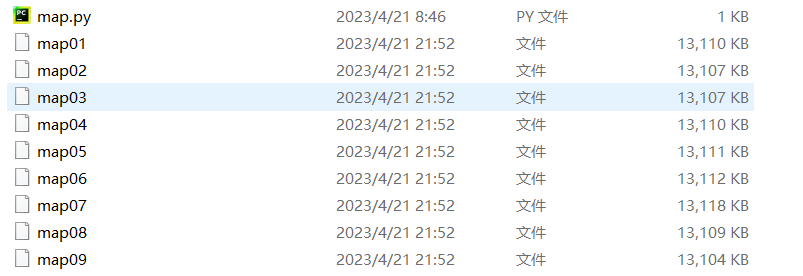


图1.12 map处理后文件

combine.py处理后，共九个文件，如下图所示：

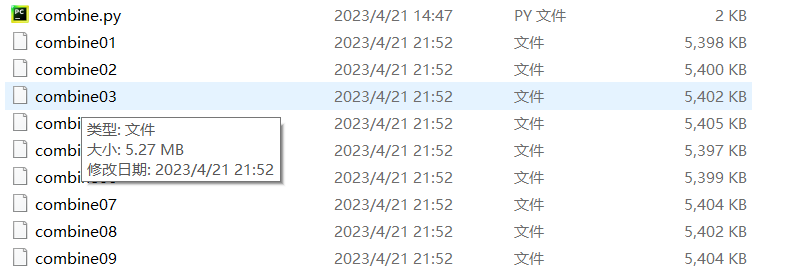
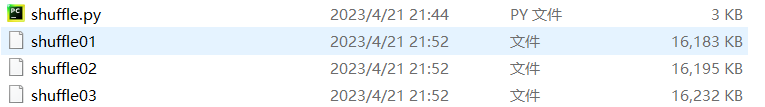


图1.13 combine处理后文件

shuffle.py处理后，共三个文件，如下图所示：

图1.14 shuffle处理后文件

最后经过reduce.py处理后，汇总到reduceResult文件中，确认程序运行成功，结果正确。以下是本次实验流程图：

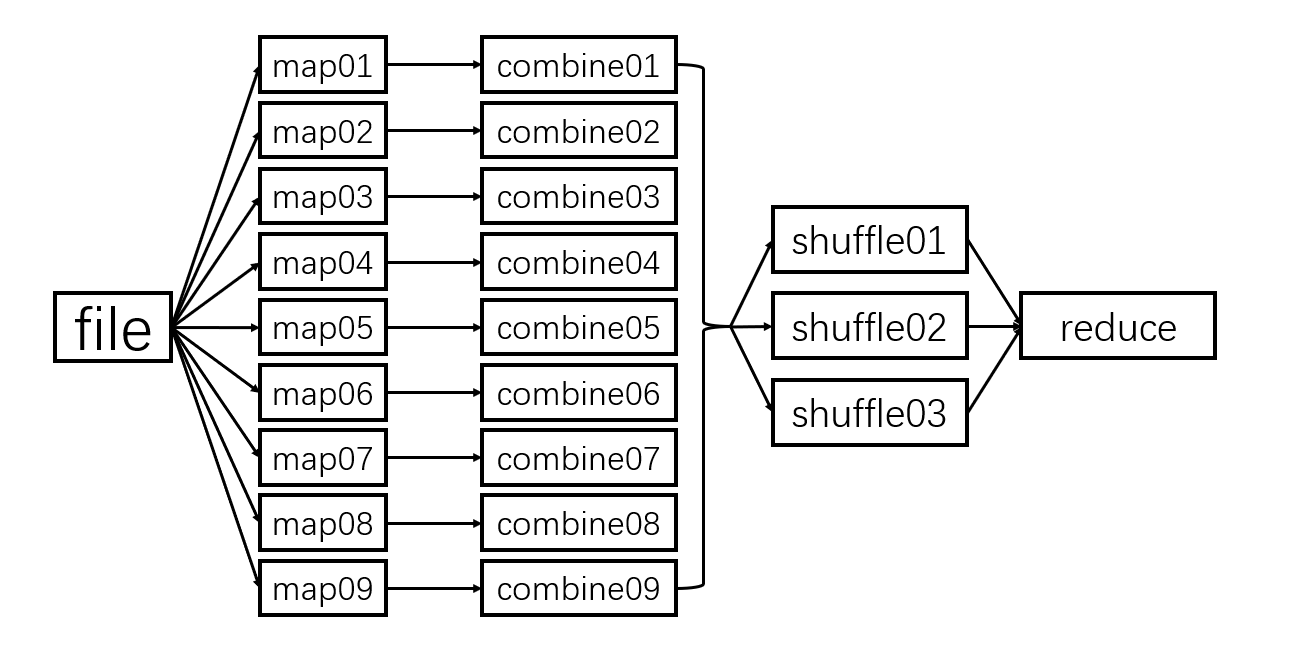


图1.15 实验流程图

在与其他同学的程序运行结果比较过程中，以及助教指出尚可改进之处，我对我的代码结果有了如下分析：

1. 整体大致实现了MapReduce思想来完成WordCount的功能，但是在combine和shuffle步骤上太过繁琐，可以将map、combine、shuffle三个步骤合成一个程序进行分布式处理，这样可以减少每一步的处理时间。
2. 多线程部分处理不够仔细，多线程同时写入文件可能会引起资源竞争和死锁问题。对于这个漏洞，我在上网查询后知道了可以使用线程锁来保护共享资源，避免多个线程同时访问和修改同一个文件。在写入文件时，每个线程需要获取锁来确保写入操作的原子性和可靠性。
3. 对于原文件处理不够完善，由于原文件内容较多、文件较大，可以使用python中的mmap模块，直接将文件映射到内存中，以便更加高效地对文件进行读取写入的相关操作。
4. 多线程分布任务不够完善，部分文件采用for循环来将任务文件分给线程处理，可能会导致程序运行缓慢，应该多查询相关库函数的用法，体会MapReduce算法中的分布式处理和合并这关键处理步骤。

## **1.4 实验总结**

本次实验通过WordCount这一个简单但又不简单的内容，让我对MapReduce有了更深的理解。对于后者算法来说，其核心是在于将大量的数据分割成几个部分，并且将这些部分分配个多个结点来处理，最后将处理的结果进行合并得到最后的结果。

我们首先要实现的就是分布式这一个工作，在本次实验中化简成多线程处理。我使用了Python中的`threading`模块，创建了9个线程，每个线程读取一个文件块，处理其中的单词，并将结果输出到一个中间文件中。

我们不仅要实现分配或者说多线程处理，更重要的是将任务尽量均匀分配给结点，让每个结点都尽可能高效率的进行任务处理。而在本次实验中，我采用了hash方法来实现均匀分配任务。

接着上一段来说，得到中间文件后，但是还没有进行最终的统计。因此我们需要进行reduce操作，即将相同单词出现的次数进行合并，并统计出每个单词的总出现次数。在本次实验中，我使用了三个reduce节点，每个reduce节点处理其中的一部分，将结果合并成最终的输出文件。在这一过程中我们使用了ThreadPoolExecutor的方式来将任务加入线程池，实现reduce归并的工作。

最后关于MapReduce的一点感想，作为一种分布式计算框架，被用于大规模数据集的并行处理和生成计算结果。极大的方便了对当今互联网产生的大量数据进行处理，是一种十分高效、可信的处理方法。