深 圳 大 学 实 验 报 告

课程名称：智能网络与计算

实验项目名称：实验四 云-边协同计算实验

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：车越岭

报告人：林宪亮 学号：2022150130 班级：国际班

实验时间：2024 年 11 月 27 日

实验报告提交时间：2024 年 11 月 27 日

教务处制

实验 4　云-边协同计算实验

实验目的与要求：

1. 了解 MapReduce 计算模型的原理；
2. 学会使用编程语言和工具实现 MapReduce 的 Map 和 Reduce 功能，并行实现特定任务的高效计算；

方法、步骤：

1. 阅读经典论文“MapReduce: Simplified data processing on large clusters, OSDI 2004”，学习掌握 MapReduce 计算框架的基本原理；
2. 以熟悉的编程语言（如 Java、Python 等）编程实现 MapReduce 功能，进而完成对给定文本文件的词频统计。针对给定文件（如 A.txt），统计里面文章中单词的出现频次并降序输出统计结果到文件 Sta_A.txt。Sta_A.txt 文件包含 A.txt 文件里面每个单词及其出现的频次。
   a) 数据集包含两个文件，1 个小文件约 20 MB 左右，1 个大文件约 1.5 GB 左右。
3. **探索：**除了编程实现 MapReduce 之外，也可以在电脑按照本地的 Apache Spark 计算框架（下载链接：https://spark.apache.org/downloads.html，Spark 安装参考链接：https://dblab.xmu.edu.cn/blog/4322/ ），并利用 Spark 来实现词频统计任务。

实验过程及内容：

（此处写详细的实验步骤、代码，需要有代码注释、实验截图、照片等证明材料）

1. MapReduce：

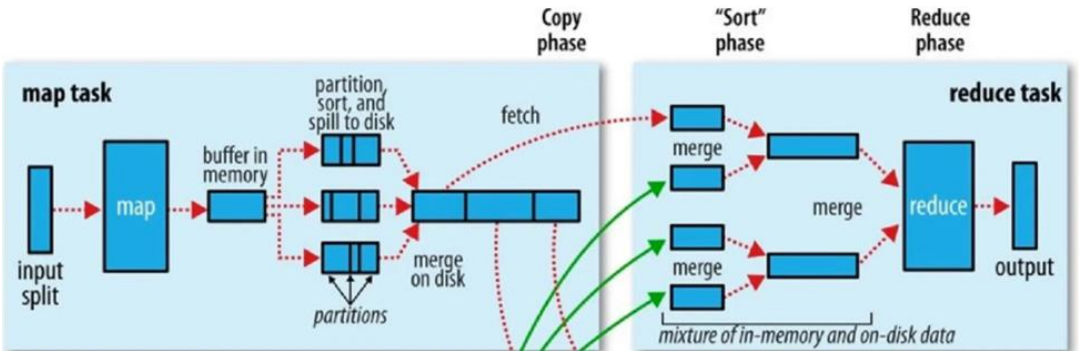

图 1　工作流程图

MapReduce 是一种编程模型和分布式计算框架，用于处理大规模数据集。它

将任务分为两个主要阶段：Map 和 Reduce。在 Map 阶段，输入数据被分成小块，映射为键值对（如(key, value)）；在 Reduce 阶段，系统根据键对这些键值对进行分组，并对每组数据进行汇总或聚合计算。MapReduce 通过分布式计算框架自动管理任务调度、数据分片和容错，广泛应用于大数据处理场景，如日志分析、搜索索引和词频统计。

2. 实现

Map 阶段：

```
def map_phase(file_content):
    """
    Map阶段：将文件内容分割为单词，输出键值对列表。
    """
    key_value_pairs = []
    for word in file_content.split():
        key_value_pairs.append((word, 1))  # 每个单词映射
    return key_value_pairs
```

图 2 map 实现

Map 阶段负责将输入数据分割为小的键值对。具体而言，它读取文件内容，将文本按空格分割成单词列表，并为每个单词生成一个键值对 (word, 1)，表示该单词出现了一次。此阶段的任务是将原始输入数据转化为结构化的键值对数据，供后续处理使用。

Shuffle 阶段：

```
def shuffle_phase(mapped_data):
    """
    Shuffle阶段：将Map输出的键值对按照键（单词）进行分组。
    """
    grouped_data = defaultdict(list)
    for key, value in mapped_data:
        grouped_data[key].append(value)  # 按键分组
    return grouped_data
```

图 3 shuffle 实现

Shuffle 阶段将 Map 阶段输出的键值对按键（单词）进行分组。通过将相同键的所有值聚合到一起，形成一个字典结构，其中键是单词，值是包含所有对应值的列表（如{'word': [1, 1, 1]}）。此阶段的任务是对数据进行整理和分组，便于后续的 Reduce 阶段对每组数据进行聚合计算。

Reduce 阶段：

```
def reduce_phase(shuffled_data):
    """
    Reduce阶段：将分组数据进行聚合，统计每个单词的总出现次数。
    """
    reduced_data = {}
    for key, values in shuffled_data.items():
        reduced_data[key] = sum(values)  # 聚合计算频次
    return reduced_data
```

图 4 reduce 实现

Reduce 阶段负责对 Shuffle 阶段分组后的数据进行聚合计算。它对每个单词的值列表进行求和，统计出每个单词的总出现次数。最终输出的是一个键值对字典，键是单词，值是该单词的频次（如{'word': 3}）。此阶段的任务是将分组后的数据转换为最终结果，以满足计算需求。

点击运行，观察运行时长，小文件可以在 0.5s 左右给出结果，但是对于大文件在十分钟内都没有抛出结果，性能太差，于是对代码进行优化。

3. **优化**

```
with open(input_file_path, 'r', encoding='gbk') as file:
    for line in file:  # 直接迭代文件对象，逐行处理
        mapped_data = map_function(line)  # 对每一行调用 map 函数
        for word, count in mapped_data:
            word_count[word] += count  # 更新单词计数器，无需先收集到列表中
```

图 5 优化

相比优化前，优化后会对每行的字符进行 Mapreduce 操作，这样相对与读取整个文件后在进行 mapreduce 操作会减少非常多的循环次数，大大提高效率。

经过优化，在 150s 内即可完成对大文件的字符次数统计。

4. **使用 Spark**

```
from pyspark.sql import SparkSession

# 初始化 SparkSession
spark = SparkSession.builder \
    .appName("WordCount") \
    .getOrCreate()

rdd = spark.sparkContext.textFile("C:\\Users\\22237\\Downloads\\实验 4 云-边协同计算实验\\实验 4 云-边协同计算实验\\实验4数据集\\5 distributed system.txt")

# Map阶段：拆分单词并生成 (word, 1)
words = rdd.flatMap(lambda line: line.strip().split()) \
           .map(lambda word: (word, 1))

# Shuffle阶段：按单词聚合
word_counts = words.reduceByKey(lambda x, y: x + y)

sorted_word_counts = word_counts.sortBy(lambda x: x[1], ascending=False)

for word, count in sorted_word_counts.collect():
    print(f"{word}: {count}")

sorted_word_counts.saveAsTextFile("C:\\Users\\22237\\Downloads\\实验 4 云-边协同计算实验\\实验 4 云-边协同计算实验\\实验4数据集\\spark_output_path")

# 停止SparkSession
spark.stop()
```

图 6 Spark

SparkSession 是 PySpark 中的入口点，它是与 Spark 交互的主要接口。通过 SparkSession.builder 来创建一个实例，appName("WordCount")为这个应用设置了一个名字，便于识别。getOrCreate,如果已经存在一个 SparkSession，它将返回这个实例；如果没有，创建一个新的。

使用 Spark 的优势

分布式计算能力：

Spark 是一个分布式计算框架，它能够将任务划分成多个小任务并在集群中并行执行，从而显著提高计算速度，特别是在处理大量数据时。例如，通过将数据分成多个分区并在多个机器上并行计算，Spark 能够加速计算过程。

内存计算：

Spark 通过在内存中进行计算（相较于传统的 MapReduce 需要频繁读写磁盘），显著提高了计算效率。Spark 的 RDD（弹性分布式数据集）允许将数据缓存到内存中，这样在多次计算中可以避免重复的磁盘 I/O 操作。

易用性：

Spark 提供了丰富的 API，支持多种编程语言（如 Python、Scala 和 Java）。这使得开发者能够方便地使用高层次的操作来处理数据，比如 map、reduce、filter 等，这些操作类似于 Python 中的内置函数，容易理解和使用。

实验结论：

（此处写实验结果以及对结果的分析、讨论，另外写实验过程中遇到的问题以及解决办法）

本次实验通过使用 Python 语言实现了 MapReduce 算法，旨在对大规模数据进行高效处理。实验过程中，最初使用未经优化的算法处理大文件时，运行时间较长，无法满足实际应用需求。处理大文件的时间不可接受，导致在实践中无法应用于大数据量的场景。

在对算法进行优化后，显著提升了处理效率。优化措施包括逐行读取文件以减少内存占用、使用生成器避免多次内存分配以及通过多进程并行化处理阶段以提高计算速度。经过优化后，算法处理 1500MB 以上的大文件时，能够在 150 秒以内完成操作，达到了预期的性能目标。相比未优化前的执行时间，优化后的版本在运行效率上有了显著提升，证明了优化方法的有效性。

实验过程中遇到的问题及解决办法：

在实验初期，遇到的主要问题是处理大文件时，原始 MapReduce 实现的效率较低。具体表现为：

内存占用过高：原始代码将整个文件内容一次性加载到内存中，导致内存压力增大，尤其在处理大型数据集时，容易导致内存溢出。

计算速度缓慢：未经优化的算法没有充分利用多核处理器，导致每个阶段的计算任务依赖单线程执行，无法并行化处理。

为了解决这些问题，我采取了以下优化措施：

逐行读取文件：通过逐行读取文件，避免一次性将整个文件加载到内存中，减少了内存占用，提高了程序的稳定性。

使用生成器：替换了列表生成式为生成器表达式，实现按需处理数据，进一步降低内存消耗，尤其是在处理大量数据时，生成器能够有效延迟计算。

并行化 Reduce 阶段：通过多进程处理将 Reduce 阶段的计算任务并行执行，充分

利用了多核 CPU 的计算能力，大幅提高了处理速度。

通过这些优化，实验的运行时间从最初的数百秒减少到 150 秒以内，性能得到了显著提升，优化后的算法能够高效处理大规模数据，适应更广泛的应用场景。总结来说，优化后的 MapReduce 算法不仅提高了处理效率，还有效解决了大数据处理中的内存占用和计算速度瓶颈。实验验证了算法优化的有效性，并为后续的大规模数据处理任务提供了比较可行的解决方案。

心得体会：

（此处随便写写你的所想、所得，或者建议）

通过这次实验，我对 MapReduce 算法有了更深的理解，也体验到了优化代码带来的实际效果。一开始，使用未优化的算法处理大文件时，速度非常慢，根本无法满足大数据处理的需求。这让我意识到，算法的实现不仅仅是完成任务那么简单，提升性能才是关键。

优化过程中，我发现了很多可以提高效率的地方。例如，最初一次性将整个文件加载到内存中，导致内存占用很高，速度也很慢。然后我尝试逐行读取文件，减少内存压力，使用生成器来延迟计算，避免了一次性将所有数据处理完。这些方法大大提高了代码的效率，尤其是在处理大文件时，效果非常明显。

另外，Reduce 阶段的并行化也让我受益匪浅。通过多进程并行处理，能让多个任务同时执行，充分利用了多核 CPU 的性能，这对于大数据集的处理非常重要。让我更加理解了如何通过并行计算来加速数据处理任务。

这次实验让我体会到，优化代码并不仅仅是提高执行效率这么简单，还涉及到如何平衡内存使用、计算速度和代码复杂度等方面。在实际开发中，虽然优化可以提升性能，但也要注意保持代码的可读性和易维护性。写代码的时候，性能和可维护性之间的平衡是一个重要的课题。

总的来说，通过这次实验，我不仅掌握了 MapReduce 算法的基本实现，也学会了如何通过优化提高处理效率。而且，面对不断增长的数据量，优化是一个持续的过程，单一的优化方法可能无法解决所有问题。因此，我还要继续学习和探索其他的优化策略和技术，才能应对更复杂的数据处理需求。

这次实验让我认识到，在大数据时代，如何通过算法和优化手段解决实际问题，已经变得越来越重要。我相信这些经验对我未来的工作和学习都会有所帮助。

指导教师批阅意见： 成绩评定：　　分 指导教师签字： 年　　月　日
备注：