OPS:Optimized Shuffle Management System for Apache Spark

Yuchen Cheng,Chunghsuan Wu,Yanqiang Liu,Rui Ren,Hong Xu,Bin Yang,Zhengwei Qi

摘要

近年来，分布式并行计算框架如 Hadoop-MapReduce 和 Spark 被广泛应用于大数据处理。由于这些框架都是采用批量同步并行 BSP(bulk synchronous parallel) 模型进行任务处理计算。中间过程的 Shuffle 阶段是必不可少的，中间数据的保存与拉取操作引起了密集的网络请求与磁盘读取 I/O 开销，这是分布式并行计算目前存在的重大性能瓶颈。本文提到的基于大数据计算引擎 Spark 的 OPS 分布式计算 Shuffle 管理系统，通过使用早期合并和早期 Shuffle 策略，大幅降低了 Shuffle 过程中的 I/O 开销，有效地调度了网络与磁盘资源，有力地缩减了任务处理的执行时间。除此之外,OPS 还提供了一个污点-重做策略来保证任务的正确执行，防止 Shuffle 过程中因节点崩溃而导致任务失败。OPS 优化 shuffle 过程中的 I/O 开销的程度接近 50%，平均缩减了 30% 的端对端任务完成时间。

关键词：分布式并行计算; BSP; Shuffle 性能优化; I/O 资源开销

1 引言

选题背景：主流的分布式并行计算框架如 Hadoop-MapReduce 和 Spark 都是采用批量同步并行 BSP(bulk synchronous parallel) 模型进行数据的处理与计算的，该计算模型的特点是数据处理的中间过程中会存在一个 Shuffle 阶段，会引起大量的磁盘读写 I/O 与网络读写 I/O 开销，严重影响待处理任务的执行时间。大量的网络读取请求与磁盘读取请求集中于 Shuffle 中间阶段，由于计算机硬件本身性能的瓶颈 (磁盘随机 I/O 读写)，运行时间在这个阶段会受到严重影响。通过改变这种计算模型或者优化中间过程的密集请求数量的相关操作是提高大数据并行处理计算效率的重要研究方向。

选题依据：随着需要处理的数据内容的不断输入，大数据计算框架 Spark 通过 Map 阶段处理后的中间数据不断增多以至于需要落盘存储。等待每一轮的数据都处理完成后，Reduce 阶段发送网络请求拉取 Map 节点处理后的中间数据再进行后续处理 (如合并统计 K-V 键值对数)。为了减少同步过程中执行时间的损耗，SCache 通过使用内存复制的方式将中间的 Shuffle 阶段与数据的传输与计算解耦合[1]，还有 Riffle 将数据合并阶段与内存合并重叠，将中间数据文件合并为一个更大的文件，将小文件的随机 I/O 低效读取转化为顺序 I/O 高效读取[2]。OPS 在融合这两种思想的前提下，优化中间过程中密集的 I/O 请求操作，提前将数据进行合并与分发至目标节点，实现大幅度降低任务执行的时间，而没有彻底改变 BSP 同步并行计算模型。

选题意义：在 Spark 的内部实现中，map 子任务的输出根据一个分区函数被分成多个块。一个块对应一个 reduce 子任务，这些块按顺序合成一个文件并存储在本地磁盘上，然后将这些不同分区块的偏移量被存储在一个索引文件中。当还原子任务启动并进入洗牌阶段时，还原子任务将根据数据依赖关系向各个节点发出洗牌请求。Spark 收到 shuffle 请求后，会根据索引文件中的偏移量读取并传输中间数据文件的一小部分。这种机制存在的问题是：当一个任务因为 CPU 核数等资源缺乏导致需要多轮次执行才能完成 Map 端数据处理时，每一轮处理完的中间数据不能及时地传输，只能存储在计

算节点的内存或磁盘里。这就是导致进入 Shuffle 阶段时密集磁盘 I/O 读取与网络 I/O 的根本原因，伴随着 TCP-Incast 等硬件性能瓶颈，任务的执行时间效率大打折扣。而本文分别提出的早期合并、早期 Shuffle 调度、本地抓取的三大策略很好地解决了上述问题，有效地调度了 I/O 资源，减少了 Shuffle 请求的集中程度，很好地加快了任务并行计算完成时间，具有较高的独创性与实现难度。因而选此作为复现论文题目有较大的意义，为我接下来进一步对 Spark 的理解与科研创新工作打好基础与铺垫。

2 相关工作

2.1 SCache：Efficient shuffle management

为了解决在大规模数据并行分析中，具有数据依赖性的任务之间对分区数据进行跨网络读取与聚合带来较大开销的问题，Scache-Shuffle 管理系统应运而生[1]。其通过在实际任务执行之前提取和分析 Shuffle 相关性，采用启发式预调度结合 Shuffle 块大小预测来预取混洗数据并平衡每个节点上的负载，实现负载均衡。同时，SCache 充分利用系统内存来加速 Shuffle 过程，使用内存复制的方式来当作外部 shuffle 服务和协同调度器，进而提高 Spark 每一轮的任务处理速度。

需要注意的是：该方法还是完全基于内存这种容量稀缺资源来进行数据的存储与计算的，当数据量不断增大，计算轮数较多时，其还是按照原始 Spark 将中间数据落盘存储并等待网络拉取，未能解决数据计算与数据传输耦合的问题, 同时容错性能差，完全基于内存的计算一旦一个子任务失败，在当前阶段就必须重新计算所有子任务并合并数据。

2.2 Riffle: optimized shuffle service

Riffle 也是大数据分析框架的优化 Shuffle 服务系统，其主要是为了解决由于磁盘 I/O 操作随着数据量的增加而超线性增加所致的性能瓶颈问题。Riffle 通过将碎片化的中间 Shuffle 文件合并为较大的块文件，从而将小的随机磁盘 I/O 请求转换为大的顺序磁盘 I/O 请求，显著提高 I/O 效率；而且通过混合已经合并的与未合并的块文件以最小化合并操作的开销，进一步提高与性能和容错性。随着版本的不断迭代更新，Spark 同样也采用了该种机制，推出了 SortShuffleManager：将每一份小的中间数据文件合并，并提供索引文件，下游 Reduce 端根据索引文件拉取节点相对应需要处理的数据。减少零散的中间文件，有效地提高性能。

需要注意的是：该方法虽减少了 Shuffle 中间过程大量的 I/O 请求数量，但其并未解决 Shuffle 中间过程网络 I/O 集中、Map 阶段磁盘 I/O 密集的问题，并未很好地调度 I/O 资源、负载分配均衡，将数据的计算与传输解耦合，因而还有较多待改进的部分。

OPS 采取并优化了 Scache 中早期 Shuffle 与任务排序的设计，同时受 Riffle 设计的启发，提出了早期合并的策略，显著地优化了网络与磁盘 I/O 资源的调度利用，即使数据量在大于内存的工作负载下也能高效工作。

3 本文方法

系统设计的数据处理分成三个阶段：

(1) 早期合并—步骤 ①②

未进行分区的数据直接传输至 ShuffleHandler，进行早期合并，新一轮 Map 继续执行。

(2) 早期 Shuffle—步骤 ③④⑤

早期合并后的数据直接传输至 Shuffler 进行早期 Shuffle，而不存储在本地磁盘。

(3) 本地获取—步骤 ⑥⑦

DataManager 提前接收传输的数据，并将其存储在 Reduce 目标节点的本地磁盘上，以确保容错与后续本地磁盘的直接抓取数据。

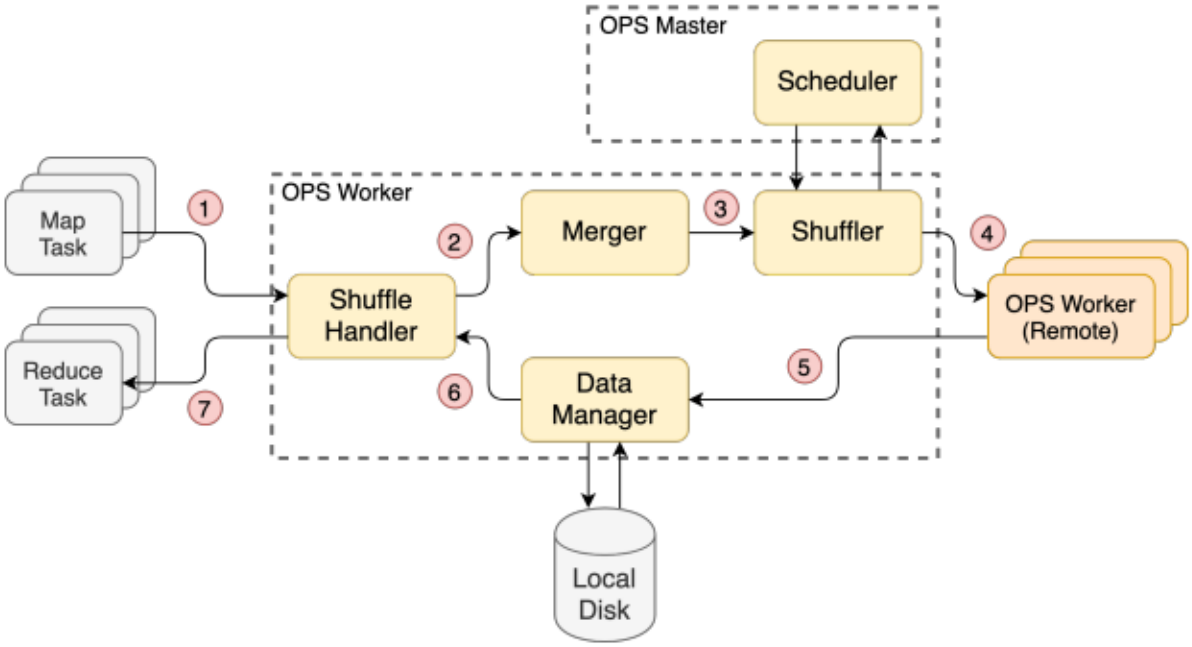

图 1: 系统方案设计示意图

3.1 中间数据的早期合并策略

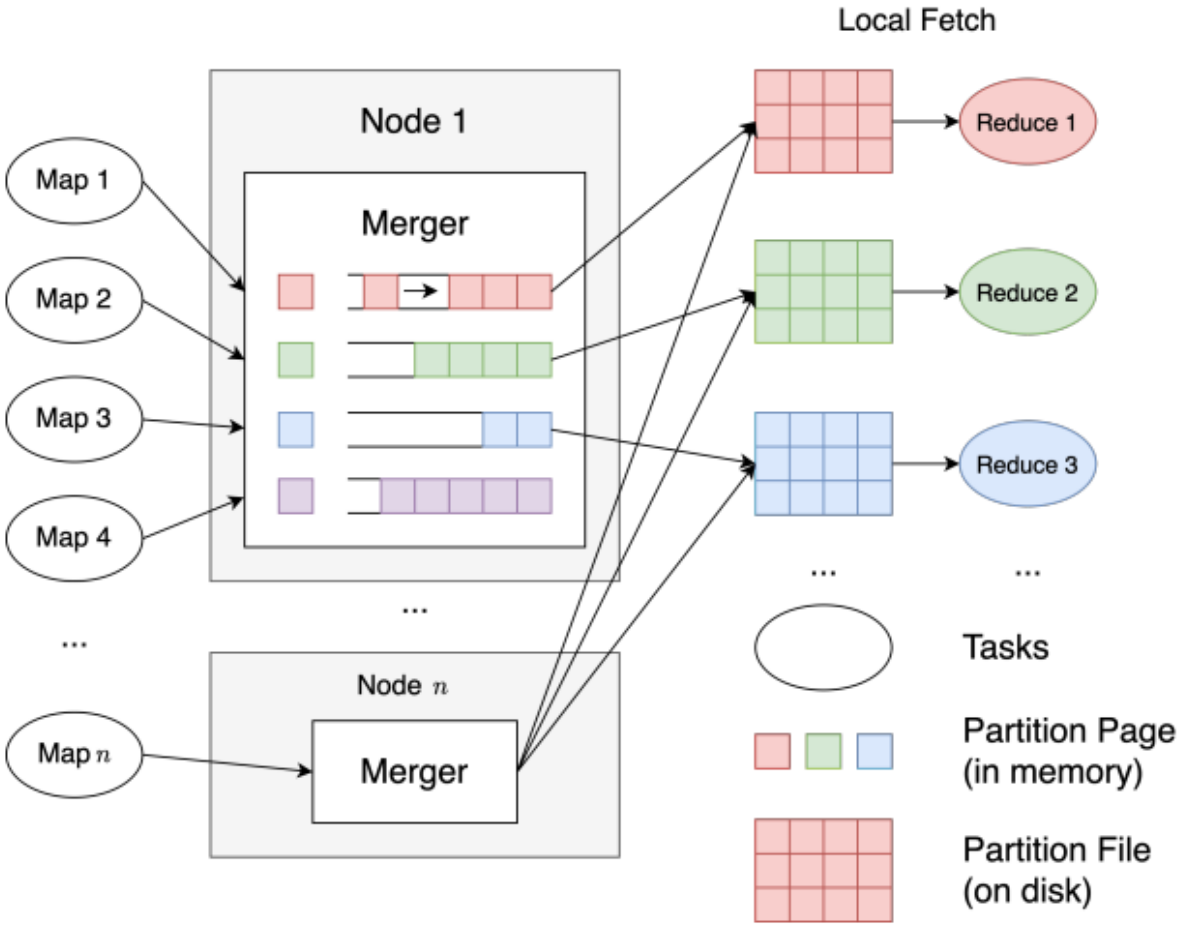

图 2: OPS 早期合并中间数据图

OPS 通过采用共享内存分页块的方式，提前将每个节点的每一轮 Map 后的中间数据进行存储并合并，而不是直接写入磁盘，消除了 Shuffle 前昂贵的磁盘 I/O 操作。该方式将 Shuffle 请求的数量改变为取决于物理节点的数量，而不再是 Map 子任务的数量, 极大地减少了 I/O 请求的开销。

OPS 的内存管理模块是在 JVM 的堆中分配内存空间，取一个叫 partition page(分区页) 的数组对象来存储同一分区的数据，分区页满后再添加至分区队列中，进行下一阶段的提前 Shuffle。如下算法 1 显示了 OPS 早期合并的过程。

Procedure 1 Early-merge algorithm

Input: intermediate data set *records* , array recording the current partition page *currentPages* , array recording the pointer of current partition pages *pageCursor*

```
function EarlyMerge(records)
while Records.hasNext do
    records← records.next
    partitionId←Partitioner.getPartition(records)
    InsertRecord(records,partitionId)
end
end function

function InsertRecord(records,partitionId)
     AcquireNewPageIfNecessary(records.size,partitionId)
     currentPages[partitionId].append(record)
     pageCursors[partitionId] += record.size
end function

function AcquireNewPageIfNecessary(AcquireNewPageIfNecessary,partitionId)
if currentPages[partitionId]+size> currentPages[partitionId].size then
    addPageToList(currentPages[partitionId],partitionId)
    pageCursors[partitionId]←0
    currentPages[partitionId]←MemoryManager.allocateNewPage()
end


end function
```

OPS 的合并器通过共享内存获得数据集记录并进行早期合并，使用默认的哈希分区函数获得分区 ID，再将数据插入到对应 ID 的内存分区页。当检查内存分区页数据已满时，放入队列里等待运输。插入数据的过程中，如果 OPS 的内存还足够，Merger 向内存管理模块请求一个新的分区页；如果 OPS 的内存已不足，allocateNewPage 函数将阻塞过程，直到一个分区页被释放，并且内存空间足够。

OPS 不断重复插入过程，并将同一物理节点上的 Map 子任务的所有输出合并为分区页。磁盘 I/O 不参与整个早期合并过程，合并后的分区数据等待提前 Shuffle，将以前随机的较小的磁盘 I/O 请求批量化为连续的较大请求。

3.2 中间数据的早期 Shuffle 策略

OPS 使用早期 Shuffle 策略将中间数据提前转移到指定节点的磁盘存储中，接下来的 reduce 子任务可以直接从本地磁盘中提取中间数据，而不会触发密集洗牌请求。当存储内存的使用达到一定的阈值时，OPS 会触发早期 Shuffle，根据分区队列将内存中的数据转移到分区节点。早期 Shuffle 策略最大限度地利用了有限的内存，提前有效地组合和传输数据，最终，中间数据直接存储在目标节点的磁盘上，等待 Reduce 子任务直接从本地磁盘上读取它。

3.3 中间数据的早期调度策略

OPS 的早期 Shuffle 策略的执行取决于一个关键信息：Reduce 子任务的调度结果；Spark 的调度器在 Map 阶段结束之前不会调度 Reduce 子任务，OPS 实现了自己的子任务调度器，根据内存中早期合并的分区结果预测最终的分区数据分布，并根据预测结果进行早期调度。为了准确预测，OPS 采用了线性模型描述分区数据与输入数据之间的关系：

$$PartitionSize_i = a_i \times InputSize + b_i \tag{1}$$

其中公式 (1) 参数: 索引 i 的分区数据的大小 $PartitionSize_i$，输入数据的大小 $InputSize$，a_i 和 b_i 是分区 i 的参数。

OPS 的所有工作节点以固定的时间间隔不断向 OPS 的调度器上传分区信息，包括每个分区的大小和当时的输入数据。OPS 的调度器汇总不同工作节点上传的分区信息，并使用线性回归计算模型参数，直到早期调度开始。以下算法 2 描述了 OPS 使用一种源于 LPT[3]的启发式算法来获得一个近似的最优解，计算出最平衡的调度结果，进而使用最小堆排序的方式将最大的分区数据分配给工作量最小的节点进行处理。

Procedure 2 Partition data early-scheduling algorithm

Input: node array *N* ,partition data array *p*

```
function EarlySchedule( N , p )
    minHeap ←build the min-heap of N by N.size
    partitions ←sort p in the descending order
    i ←0
while i < len(partitions)-1 do
    minNode ←minHeap[0]
    minNode.place(partitions[i])
    minNode.size +=partitions[i].size /s[minNode.id ]
    Shift down minHeap[0] in min heap by size
    i++
end
end function
```

OPS 基于早期调度的结果，将早期合并的中间数据早期 Shuffle 传输至目标节点并落盘。相比于原始的 Spark，将数据的计算阶段与传输阶段成功地进行了解耦合，将整个 Shuffle 阶段提前到新一轮 Map 阶段的开始，使得 Map-Reduce 中间数据的传输与新一波的数据 Map 计算可以平行地进行，从而避免中间过程密集的网络与磁盘 I/O 的请求。

4 复现细节

4.1 实验环境搭建

Metric	Value
CPU	3.1 GHz Intel Xeon Platinum 8000 series (Skylake-SP or Cascade Lake)
vCPU	4
Memory	16 GB
Storage	AWS EBS SSD (gp2) 256 GB
Storage IOPS	750
Storage Bandwidth	250 Mbps
Network Bandwidth	4.8 Gbps
OS	Amazon Linux 2

图 3: 论文集群测试平台的单节点配置

论文原实验：Spark-sort 应用程序，1.6TB 的随机文本在任务总时间统计实验中被分成 1600(4 轮)、2400、3200、4000、4800、5600、6400(16 轮) 份，平均存储在 HDFS 的节点上。实验环境建立在亚马逊 AWS EC2 集群上，租用 100 个节点 CPU 总核数为 400，单轮并行 Map 子任务为 400 个，因而通过计算得知每个阶段的子任务执行轮数为 4-16 轮，每个子任务处理的数据量约为 256MB-1024MB。每轮 Map 完达到数据运输阈值 80% 早期合并运输至目标节点，下一轮 Map 子任务的执行与当前轮数据运输同时进行。

复现实验：实验室资源紧缺不足，根据实验验证目的做出调整：使用 java 包随机生成一个长度为输入值的随机文本字符串，大小在 32g，在任务总时间统计实验中被分成 32(2 轮)、40、64、88、120、128 份 (8 轮)，平均存储在 HDFS 的节点上。即调整 Map 阶段分区数由 32 变化为 128（4 节点，单节点 4 核，虚拟机内存分配 4g，任务执行时分配最大内存为 2g，体现原 Spark 数据量大时的落盘操作）并行处理，最多需要 8 轮执行完 Map 任务，每个子任务处理的数据量约为 256MB-1024MB，每轮 Map 完达到数据运输阈值 80% 提前合并并运输至目标节点。

目的验证：①I/O 资源合理调度，避免密集集中；②任务运行轮数一致情况下，任务运行时间相比原 Spark 消耗较少，减少 Shuffle 阶段时间开销。关键：运行轮数 >=2，体现早期合并与早期 Shuffle 的优势。主要工作：①服务器多节点 Spark 开发环境部署 jdk1.8 hadoop3.3.4 scala2.13.10 spark3.3.1 ②服务器多节点 OPS-Spark 开发环境部署 ③“半”开源 OPS 项目 Maven 构建与编译打包（最耗时）④编写测试程序 Spark-Sort.jar（Scala）, 数据文本生成程序（java），上传数据到 HDFS 通过 Spark 读取 ⑤CDH 大数据集群监控平台部署测试 IO 资源变化 ⑥算法 2 小 Demo 编写理解与 Shuffle 阈值 I/O 请求数计算

4.2 与已有开源代码对比与创新点

复现过程与解决方案：

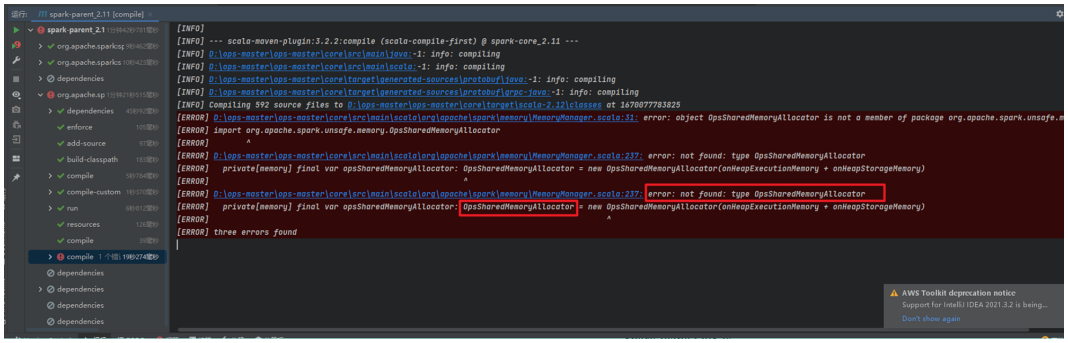

图 4: 复现过程主要问题

如图 4，开源项目代码编译报错，缺乏相关自定义类 OpsShareMemoryAllocator 即算法 1 内存调度分配模块。通过算法 1 理解与 Spark 内置内存分配模块的学习，编写百行实现（创新点）该模块最终成功编译，添加依赖打成 jar 包成功部署于服务器，如下图 5。

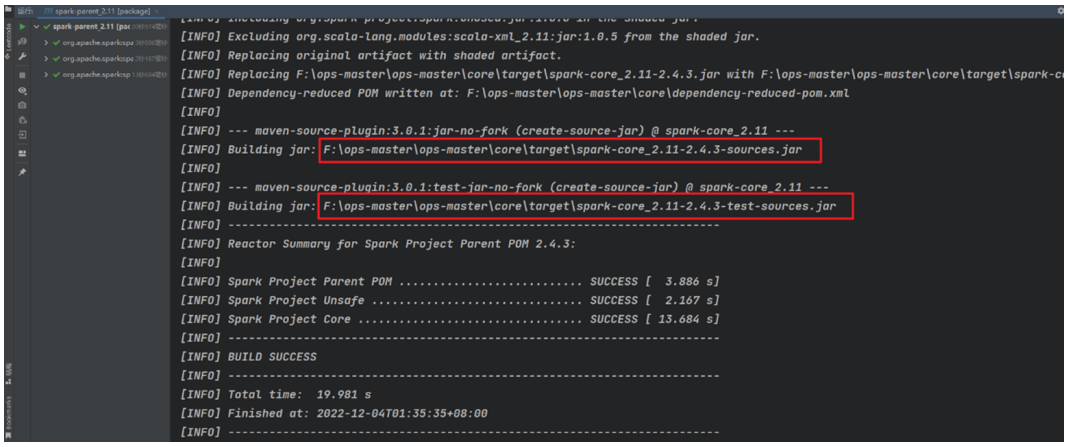

图 5: 复现编译过程

```
public class OpsShareMemoryAllocator implements MemoryAllocator{
      //分区数
      private final int numPartitions;
      //该数组记录了当前正在写入的分区页，数组的索引与分区页的ID相对应
      private MemoryBlock[] currentPages;
     //该数组记录了当前分区页的下一个数据应被写入的位置，数组的索引也与分区ID相对应。
      private long[] pageCursors;
      //Spark自带任务内存调度器
      private final TaskMemoryManager taskMemoryManager;
      //Spark自带任务块管理
      private final BlockManager blockManager;
      //Spark自带任务上下文环境
      private final TaskContext taskContext;
```

图 6: 类数据结构

```
private void acquireNewPageIfNecessary(int required, int partitionId) {
    //判断是否需要开辟新页  //required size
    if (currentPages[partitionId] == null ||
      pageCursors[partitionId] + required > currentPages[partitionId].getBaseOffset() + currentPages[
      partitionId].size() ) {
      if (currentPages[partitionId] != null) {
        addSharedPage(currentPages[partitionId].pageNumber);
      }
      currentPages[partitionId] = null;
      while (currentPages[partitionId] == null) {
        try {
          currentPages[partitionId] = allocateSharedPage(required);
          if (currentPages[partitionId] == null) {
            // sleep, wait for retry
            Thread.sleep(1000);
            continue;
          }
          currentPages[partitionId].partitionId = partitionId;
          pageCursors[partitionId] = currentPages[partitionId].getBaseOffset();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
  }
```

```
public void insertRecord(Object recordBase, long recordOffset, int length, int partitionId)
    throws IOException {
    //插入数据部分

    final int required = length;
    acquireNewPageIfNecessary(required, partitionId);

    assert(currentPages[partitionId] != null);
    final Object base = currentPages[partitionId].getBaseObject();

    currentPages[partitionId].pointers.add(new OpsPointer(pageCursors[partitionId], length,
    partitionId));
    //调用原Spark内存复制函数
    Platform.copyMemory(recordBase, recordOffset, base, pageCursors[partitionId], length);
    //pageCursor数组对应下标内容长度变化
    pageCursors[partitionId] += length;
  }
}
```

图 7: 算法 1 函数实现代码

图 6 显示了自定义类 OpsShareMemoryAllocator 的主要数据结构，再继承了父类 MemoryAllocator（源代码已有）的方法的基础上，加入记录当前分区页的数组 currentPages 与记录当前分区页的下一个数据应被写入的位置的数组 pageCursors。

整个数据合并的过程概括为：在向相应的分区页插入数据之前，Merger 首先执行 AcquireNew-PageIfNecessary 函数，检查当前分区页的空间是否足够。如果没有足够的空间，Merger 向内存管理模块请求一个新的分区页，并将旧的分区页添加到分区中，在队列中等待传输。如果 OPS 的内存用完了，allocateNewPage 函数将阻塞，直到一个分区页被释放，并且内存空间足够。最后，Merger 将数据插入到分区页中，并增加 pageCursor。至此，一个数据的插入过程结束。

4.3 界面分析与使用说明

数据的计算过程见视频；使用时先：build/mvn -DskipTests clean package 配置好服务器节点 Spark 运行环境后，再将相应的 Spark 开发程序 jar 包使用终端命令运行：./bin/spark-shell；spark-submit –class wordcount.app –master yarn –deploy-mode cluster –conf spark.executor.memory=2g –conf spark.executor.cores=4 –conf spark.dynamicAllocation.maxExecutors=32 ./ops-algos-1.0-SNAPSHOT-jar-with-dependencies.jar（指定 Executor 内存为 2g，分区数为 32，task 并行数为 4）。

5 实验结果分析

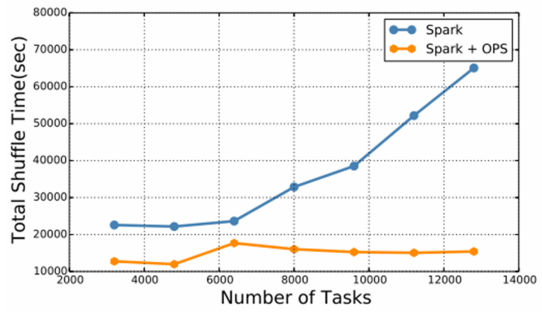

图 8: 论文-不同子任务数下的总 Shuffle 时间

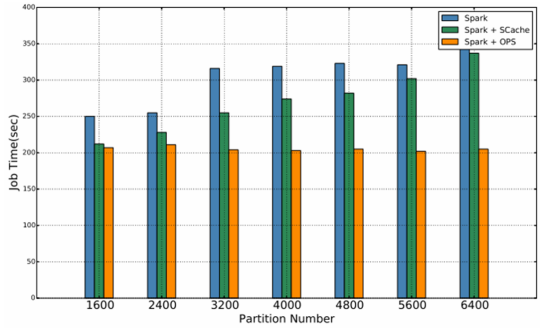

图 9: 论文-不同子任务数下的总任务执行时间

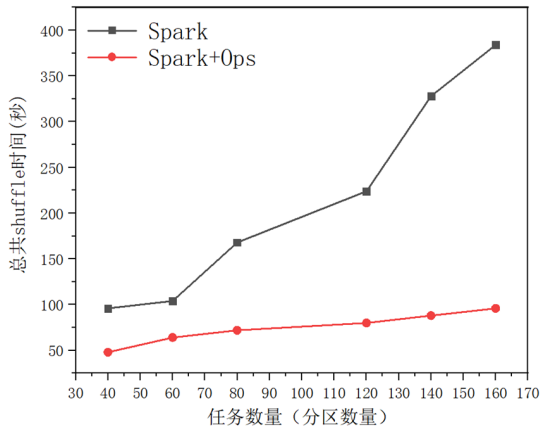

图 10: 复现-不同子任务数下的总 Shuffle 时间

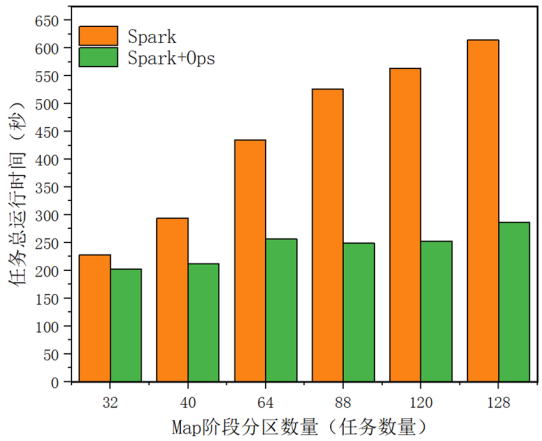

图 11: 复现-不同子任务数下的总任务执行时间

分析如下：如上图 8 与 10 的对比所示，OPS 很明显可以通过减少 Shuffle 运行时间的消耗从而提高数据分布式计算的效率，在 CPU 内核与内存等资源充足的条件下，Map-reduce 的时间损耗主要发生在 shuffle 阶段，且主要原因是密集的磁盘与网络 I/O 所花的时间。

如图 9 与 11 所示，当指定不同的 Map 子任务数（不同的数据分区数量）时，随着分区数量的增大与 CPU 核数并行利用的限制，数据块越来越小与零散化，原始 Spark 的磁盘 I/O 数骤增，I/O 读取也是主要为小文件的随机 I/O 低效地读取，因而所花的时间会越来越高。而 OPS 很好地通过早期合并转化为顺序 I/O 的高效读取，并且是只有在 Reduce 阶段才进行数据从磁盘的本地读取，很好地规避

了该部分的时间消耗。OPS 通过策略成功地减少了因 Spark 计算轮数增大所带来的剧烈时间开销，虽然 Shuffle 数据总量不变，但当任务分区数从 32 增加至 128，轮数从 2 轮增加到 8 轮，OPS 的运行时间并没发生太大的变化，原始 Spark 的 Shuffle 阶段的总执行时间逐渐增加，最终增加了近 3 倍。

6 总结与展望

随着数据量和计算规模的增加，shuffle 阶段的密集网络和磁盘 I/O 成为分布式并行计算中最重要的性能瓶颈。为了优化上述 shuffle 阶段的性能，基于 Spark 的 Shuffle 管理系统 OPS 应运而生。OPS 取代了原来 Spark 的中间数据传输方式，通过早期合并与早期 Shuffle 本地读取的策略将小文件的随机 I/O 转换为大文件块的顺序 I/O，并且将 Shuffle 阶段的数据传输与 Map 每一轮计算的阶段分开。

本次复现成功地编写补充代码通过编译与运行，做了大量的环境部署工作（多节点多依赖的框架）与测试环境，编写了具体的 Spark 开发应用程序与 Java 数据生成程序，理解算法的基础上编写了 Demo 加强理解，最终成功地验证了论文思想的可靠性与准确性。

此次较简陋的复现花费了较多的时间，但一份付出一分收获。选择该论文的目的就是为了更加了解 Spark 的运行机制与分布式计算的模型特点，针对 Shuffle 这个阶段做出改进与理解 Spark 的源码设计艺术是目前我们实验室工作的一大重点，进一步利用实验室已有的 RSP 技术与 logo 计算框架，广域互联网连接的多个数据中心的计算和存储资源，采用多集群 Spark 协同计算，实现大规模分布式大数据智能计算任务的目标。

由于本人理论知识与动手能力的欠缺，此次复现过程也存在较多不足：比如很好地利用论文提及的 HiBench 监测系统监测完美的集群 I/O 资源变化图，而是通过 CDH 监测平台 +Shell 终端系统资源监测的方式分析 I/O 资源的变化趋势，而不能得到具体的数据点绘图。

参考文献

[1] FU Z, SONG T, QI Z, et al. Efficient shuffle management with SCache for DAG computing frameworks [C]//Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2018: 305-316.

[2] ZHANG H, CHO B, SEYFE E, et al. Riffle: optimized shuffle service for large-scale data analytics[C]//Proceedings of the Thirteenth EuroSys Conference. 2018: 1-15.

[3] GRAHAM R L. Bounds on multiprocessing timing anomalies[J]. SIAM journal on Applied Mathematics, 1969, 17(2): 416-429.