浅析大数据框架 Spark 和 MapReduce 性能差异: ——以典型机器学习负载为例

祁晟	郑乃千	吴秉阳	仲殷旻
2201111657	2201111684	2201111670	2201111686

1 引言

世界已经进入万物互联的新时代。在各行各业的信息化进程中, 大量线上数据涌现。存储收集大量数据已是常态, 从大量庞杂数据中发掘提炼有价值信息更是大数据环境不可或缺的重要一环。很多工作都致力于提供更高效、更简易的大数据分析框架。Spark (Zaharia et al. (2010)) 和 MapReduce (Dean and Ghemawat (2008)) 就是大规模数据分析框架的重要代表。对于大数据框架普通用户而言，他们通常缺乏对框架底层实现细节的了解，而又需要在两个框架中择优使用。因此本实验报告希望以典型机器学习模型为例，详细剖析 Spark 和 MapReduce 这两个流行大数据框架的性能差异，并阐明其中性能差异来源，希望最终能给大数据框架用户提供有参考价值的框架选择建议和框架实现优化建议。本实验报告将按如下形式组织:

1. 在第 2 章，我们将介绍我们选择任务负载时候的考量，以及任务负载的具体细节。
2. 在第 3 章, 我们将介绍我们选取的任务负载在 Spark 和 MapReduce 两个平台上的具体实现及优化。
3. 在第 4 章，我们将介绍 Spark 和 MapReduce 的性能评估结果，以及他们各自的详细性能分解结果。
4. 在第 5 章，我们将总结 Spark 和 MapReduce 的性能差异，并尝试向大数据框架普通用户给出建议。

相关代码实现开源在 Github 平台 。

2 经典机器学习负载：逻辑回归模型

本次项目我们选取了经典机器学习负载，逻辑回归模型，作为我们测试 Spark 和 MapReduce 的负载。逻辑回归模型是一类经典常用的机器学习分类模型，其致力于找到一个最好的超平面 w，从而最好的将两类向量嵌入分开。该算法使用梯度下降来更新这一超平面 w。w 最初是随机生成的，然后经过多轮迭代更新 w 实现较好分类效果。每轮迭代时，算法会算出每个数据点的 logstics 函数相对于 w 的梯度，并且使用该梯度更新 w。

我们选取这一算法作为我们测试的负载，是基于以下原因:

1. 该算法有足够复杂的代码逻辑，包含了 map 和 reduce 过程，并且包含多轮迭代这样复杂的逻辑结构。
2. 该算法在大数据场景有较高的并行度，可以充分展示 Spark 和 MapReduce 框架自身的性能潜力。

3. 该算法本身便是大家广泛使用的经典机器学习算法，有较高代表性。

基于以上考量，我们最终选取了逻辑回归模型作为任务负载。

3 逻辑回归模型在 Spark 和 MapReduce 框架上的具体实现

3.1 Spark 版本逻辑回归模型实现

Spark 版本逻辑回归模型用 Python 语言实现，选取了 PySpark 来支持 Spark 程序运行，整体实现非常便捷。我们没有直接使用 Spark 上层封装的 MLlib (Meng et al. (2016))，希望能够更好展现普通用户实现的程序的性能，避开专业优化。具体实现流程如下：

1. 初始化时，从 HDFS (Shvachko et al. (2010)) 内读取逻辑回归模型的数据集，并且随机初始化权重。
2. 每轮迭代开始时，对所有数据点调用 **map(gradient)** 方法，计算单点逻辑回归梯度。
3. 对所有单点逻辑回归梯度，执行 **collect(sum)** 方法，计算数据集整体逻辑回归梯度。
4. 每轮迭代重复上述 2 - 3 步，直至达到要求迭代轮数。

3.2 MapReduce 版本逻辑回归模型实现

MapReduce 版本逻辑回归模型用 Java 语言实现。逻辑回归模型被拆解成了多轮 MapReduce 任务。具体实现时，主要实现了 $mapreduce.Mapper$ 和 $mapreduce.Reducer$ 两个基类的子类，完成逻辑回归相应步骤。具体流程如下：

1. 在 **Mapper.setup()** 方法初始化时，从 HDFS 内读入逻辑回归权重。
2. 在 **Mapper.map()** 方法读入数据后，计算对应数据点的逻辑回归梯度。利用 *in-mapper-combine* 技巧，实现同一 Mapper 内数据点梯度预聚合。
3. 在 **Mapper.map()** 方法注销 Mapper 时，将聚合后的各个权重梯度分别发送给对应 Reducer 。
4. 在 **Reducer.reduce()** 方法收到所有 Mapper 发送的对应权重以后，进一步聚合梯度，并将结果存储至 HDFS 指定位置。

4 测试评估

4.1 实验环境

为了充分展现 Spark 和 MapReduce 两个大数据框架的性能潜力，我们选取物理光纤相连的两台同等规格物理服务器来进行实验，具体规格如下表 1 所示。

硬件组件	型号/性能
中央处理器	20 核 Intel Xeon Silver 4210R (2.4 GHz - 3.2 GHz)
主存储器	64 GB DDR 内存
网卡	1 Gbps 高速网卡

表 1: 实验服务器配置。

在软件环境方面，我们也力求使用最新版软件，尽可能的表现 Spark 和 MapReduce 两个大数据框架的最新情况。完整软件环境如下所示：

- Linux Kernel 5.4.0
- Ubuntu 18.04
- OpenJDK 11.0.6
- Spark 3.3.1
- Hadoop 3.3.4

4.2 数据集生成

为了充分测试 Spark 和 MapReduce 版本的逻辑回归模型在各种数据规模情况下的性能表现，我们构造了四种数据集规模：0.5 GB，1 GB，2 GB，和 4 GB。数据集内，每个数据点都由一个 10 维向量嵌入和一个对应值组成，与真实的高维大数据场景一致。

4.3 实验测量方法

在每轮测试前，我们都会进行预热，尽可能避免多次测试的抖动问题。预热时的相关操作、数据在预热后不会保存或者缓存在主存中，保证不会对正式测试逻辑产生影响。在正式测试过程中，我们设定逻辑回归模型会迭代训练 10 轮。根据我们的经验，之后的每轮迭代性能基本稳定，所以不做更久测试。我们测试的结果数据也结合了 Spark HistoryServer 和 MapReduce HistoryServer 记录的信息。他们对各过程详细的时间记录有助于我们进一步分析 Spark 和 MapReduce 之间的性能差异。

4.4 Spark 和 MapReduce 性能对比

我们在四种不同规模的数据集上，分别测试了 Spark 版本和 MapReduce 版本逻辑回归模型的端到端延迟。图 1 具体展示了 Spark 和 MapReduce 在不同规模数据集上的延迟表现。可以看到，随着数据规模增大，Spark 和 MapReduce 端到端延迟也近似线性增长，说明测试数据规模已经足够大到能够测试可扩展性。同时，在任何数据规模情况下，Spark 版本实现相比于 MapReduce 版本实现均有明显优势，可以实现 16.7 至 18.6 倍的性能提升。此外，我们观察到这种性能提升基本不随数据规模大小发生变化，充分说明了 Spark 在有充足内存的情况下，性能总是能显著强于 MapReduce。

4.5 Spark 和 MapReduce 迭代性能分解

为了进一步探析 Spark 和 MapReduce 性能差异的原因，我们将端到端延迟拆解为每轮迭代延迟，进一步分析两大框架的性能原因。图 4 展现了这两种大数据框架在不同迭代轮数是的每轮迭代耗时。如图 2 所示，Spark 除了第一轮偏高，其他每轮耗时较低且基本稳定，充分证明了 Spark 将中间结果存入内存带来的性能增益足够显著。而 MapReduce 性能则如图 3 所示，每轮迭代耗时基本相同。这也和 MapReduce 每轮都需要从 HDFS 内读入数据，并最终将更新后的逻辑回归模型的行为相吻合。因此我们基本认为 Spark 相比于 MapReduce 的主要性能增益来源于中间数据存储于内存中。

尽管如此，我们仍然对第一轮迭代时间做了进一步对比。图 5 展示了不同数据规模时，Spark 和 MapReduce 的耗时情况。可以发现，即使第一轮迭代 Spark 和 MapReduce 都需要从 HDFS 内读入数据，Spark 仍然明显快于 MapReduce。这固然因为 Spark 不需要把更新过后的逻辑回归模型权重写回 HDFS，但是如此明显的性能差异让我们认为 MapReduce 本身框架的性能开销也大于 Spark。

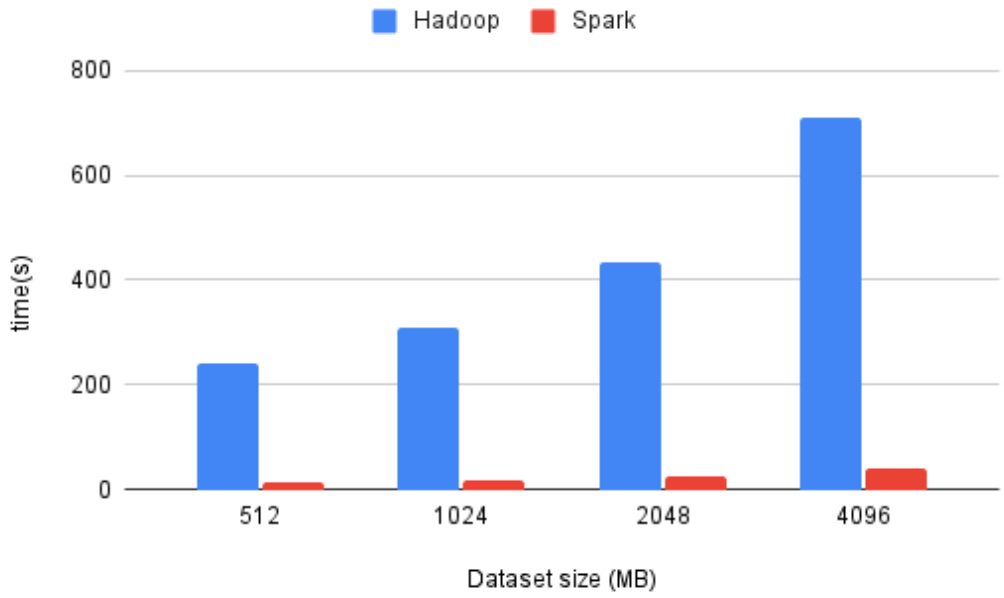

图 1: Spark 和 MapReduce 端到端延迟比较。

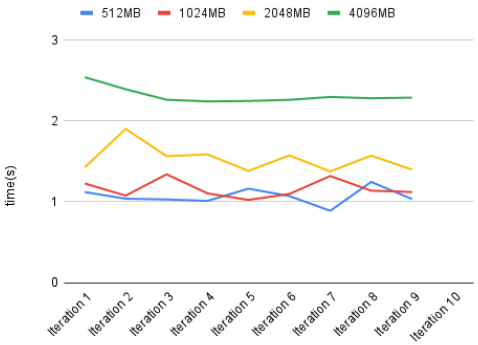

图 2: Spark 每轮迭代耗时

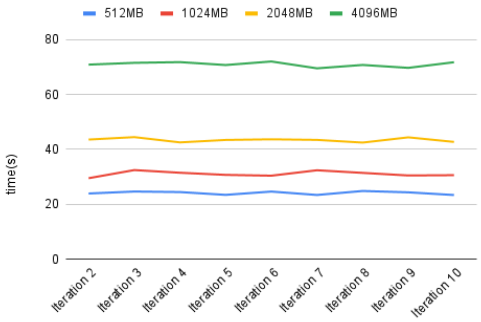

图 3: MapReduce 每轮迭代耗时

图 4: Spark 和 MapReduce 每轮迭代耗时时序图。

4.6 验证 MapReduce 性能瓶颈

为进一步剖析 MapReduce 的性能瓶颈，我们针对 MapReduce 每一轮的迭代耗时做了进一步分解。具体来说，我们将 MapReduce 分为四个阶段：map、shuffle、reduce、other。map 表示 Mapper 耗时；shuffle 包括从文件中读入数据并 shuffle 到各个 Mapper 上的时间和 Mapper 产生的中间结果 shuffle 到 Reducer 上的时间；reduce 表示 Reducer 的处理耗时；other 表示其他耗时。

图～6 展现了具体的耗时分解情况，因为 reduce 耗时几乎为 0，因此没有在图中呈现。可以看到耗时最大的部分是 shuffle 阶段。shuffle 阶段的主要任务是从 HDFS 中读取输入、写入输出以及 Mapper 将中间结果写入本地文件并传给相应 Reducer。因为我们实验环境网络延迟

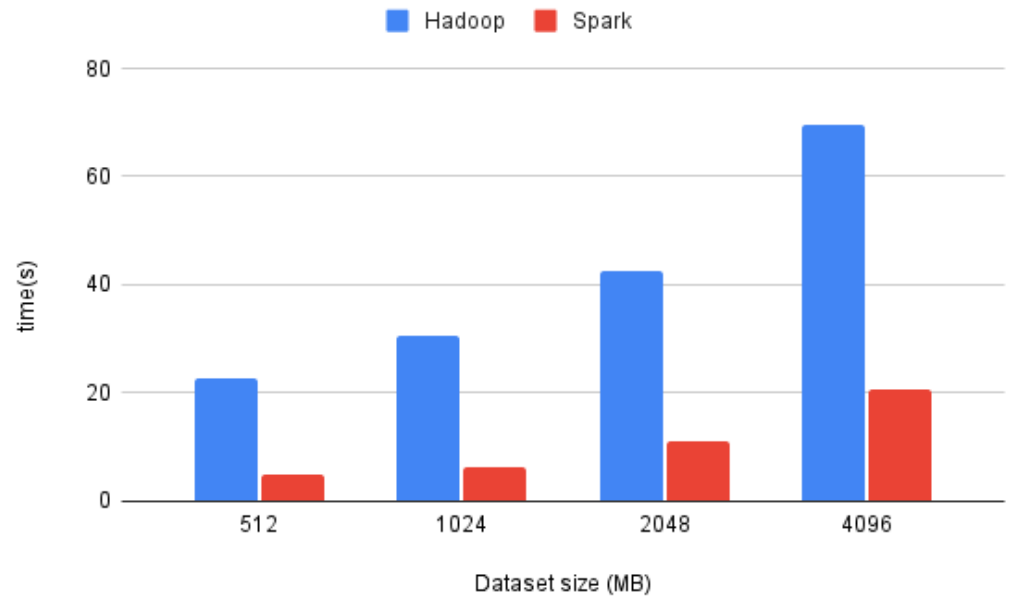

图 5: Spark 和 MapReduce 第一轮迭代时间对比。

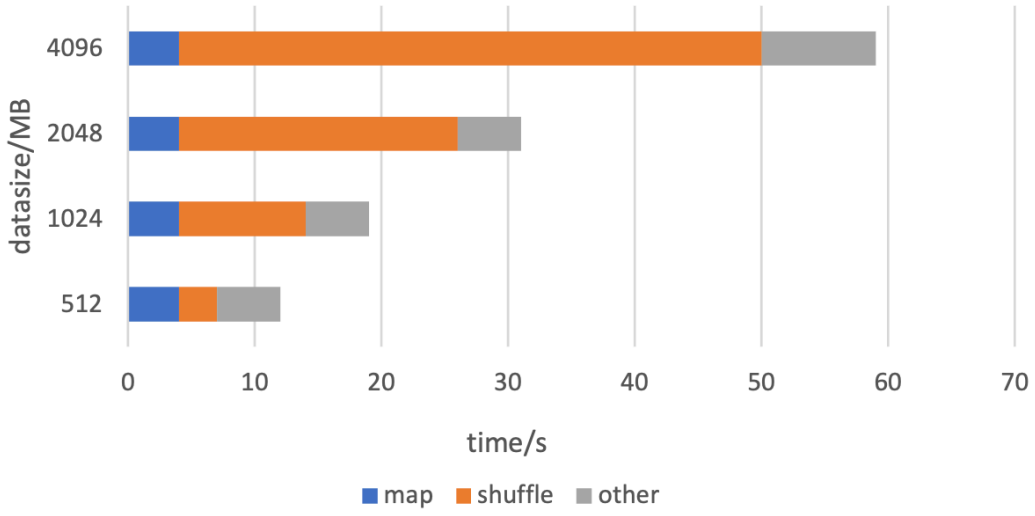

图 6: MapReduce 分阶段分解。

小于 1 毫秒且带宽大于 1Gbps，所以可以排除网络传输问题，这证明了主要问题还是在于中间结果没有缓存在内存中。另外，分解中 other 一部分的耗时也不可小觑，也印证了之前小节认为 MapReduce 本身开销较大的问题。

5 总结

在本次报告中，我们选取了逻辑回归模型这一代表性的机器学习任务为负载，在不同规模数据集上详细比较 Spark 和 MapReduce 两大流行大数据框架的性能差异。实验结果表明，当机器内存足够保存相关数据时，Spark 相比于 MapReduce 在性能上有明显优势。在使用过程中，我们也发现，虽然 Spark 和 MapReduce 在表达能力上没有明显差异，但是 Spark 有更易用的 API，也有更丰富的生态，一些常用代码可以直接调用成熟的函数库，比如 MLlib。因此，对于大数据框架普通用户，应该首选 Spark。此外，我们详细的性能分解也证明了 Spark 很大一部分的性能增益来自与将中间结果缓存在内存中。所以当编写 Spark 代码时，用户还应该尽可能将数据缓存在内存中，尽可能避免 HDFS 的读取和写入，也要尽可能避免重计算的发生，进而充分释放 Spark 性能潜力。

参考文献

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.