STAT243 Lecture 7.2 MapReduce and Dask

1 Overview

1.1 背景与发展 (Background)

- **传统高性能计算 (HPC)**
  传统 HPC 关注基于消息传递的并行计算技术，如 **MPI (Message Passing Interface)**，以及如何通过优化算法提高效率。
- **现代趋势**
  在过去二十年中，计算重点转向了**大规模分布式数据集**的处理：
  数据分布在多个机器上，但用户可像操作单个数据集一样进行分析。
- **常见工具**
  - **Spark**：在 2010 年代广受欢迎
  - **Dask**：Python 原生的分布式计算框架 (本单元主要讲解)

1.2 MapReduce 范式 (The MapReduce Paradigm)

- **核心思想**
  将数据分布在多个节点上，每个节点独立处理本地数据，然后再汇总结果。
  适用于超大规模数据集的分布式计算。
- **MapReduce 的四个主要步骤**
  1. **读取数据**：从文件或记录中读入原始数据
  2. **Map 阶段**：将输入转化为键值对 `{key, value}`
  3. **Reduce 阶段**：对每个键的所有值执行聚合操作，输出 `{key, result}`
  4. **输出结果**：写出最终的 `{key, result}` 集合
- **类比范式**
  Pandas 与 R 的 **dplyr** 中的 **split-apply-combine** 策略与 MapReduce 思想一致。

1.3 MapReduce 的灵活性与限制

- Map 阶段可以执行过滤、转换或扩展操作 (一个输入生成多个输出)。
- Reduce 阶段可执行复杂聚合与统计分析。
- 限制：
  - Map 阶段的每个记录必须独立处理
  - Reduce 阶段的每个键的聚合也必须独立
  - 这确保了可并行执行

> ⚠ **Remark: "Shuffling" data**
>
> 若操作需要跨节点移动大量数据，会显著降低性能
>
> - 例如计算 **中位数 (median)** 时，需移动数据以按组重排。
> - 而计算 **均值或总和 (mean/sum)** 时，只需传输每个节点的部分和，通信量小得多。
>
> 因此，MapReduce 的效率取决于任务是否需要大量数据移动。

1.4 Hadoop

- 基于 Java 的 MapReduce 框架
- 核心组件：
  - **HDFS (Hadoop Distributed File System)**：分布式文件系统，数据自动冗余存储
  - **任务调度系统**：监控节点执行与故障恢复
- 优点：容错性强，可在大规模集群上运行
- 缺点：
  - 配置复杂，维护成本高
  - 主要用于磁盘为主的计算 (IO 密集型)

1.5 Spark

- 可视为“内存化的 Hadoop”
- **将多节点内存视为统一内存池**，支持高速数据操作
- 特点：
  - 数据能放入内存时速度远超 Hadoop
  - 若超出内存则自动使用 HDFS
- 缺点：
  - 集群配置复杂，调优困难
  - 内存溢出错误常见且难以调试

---

2 使用 Dask 进行大数据处理 (Using Dask for Big Data Processing)

2.1 基本特性

- Dask 继承 MapReduce 思想，可：
  - 在单机多核或多节点上**并行化数据处理**
  - **自动将数据分块** (chunks/shards/partitions) 并行处理
- 优势：
  1. **并行化计算与读取**：每个 worker 处理部分数据
  2. **扩展可处理数据规模**：可利用多节点的内存或磁盘

2.2 注意事项

- 如果数据仅在一个磁盘上，多个进程同时读写会受限于磁盘 IO
- 超算系统通常具备并行文件系统，可实现真正的并行 IO
- Hadoop/Spark 通过“多节点多磁盘” (通常每个机器/节点一个磁盘) 实现分布式 IO
- 在单节点上使用 Dask 时，建议使用 **threads 调度器**，避免频繁数据拷贝

2.3 Dask DataFrame (类 Pandas 对象)

- **结构**：每个 Dask DataFrame 由多个小型 Pandas DataFrame 构成
- **默认调度器**：threads
- **功能**：支持大部分 Pandas 操作，但不完全兼容

2.3.1 示例：读取航班延误数据 (约 11 GB)

Python

```python
import dask
dask.config.set(scheduler='threads', num_workers=4)
import dask.dataframe as ddf

path = '/scratch/users/paciorek/243/AirlineData/csvs/'
air = ddf.read_csv(path + '*.csv.bz2',
      compression='bz2', encoding='latin1',
      dtype={'Distance': 'float64', 'CRSElapsedTime': 'float64',
             'TailNum': 'object', 'CancellationCode': 'object',
             'DepDelay': 'float64', 'ActualElapsedTime': 'float64',
             'ArrDelay': 'float64', 'ArrTime': 'float64', 'DepTime': 'float64'})
air.npartitions
```

- Dask 会并行读取多个压缩文件
- 但实际读取在 `.compute()` 调用前不会发生 **(lazy evaluation)**

2.3.2 示例：基本计算

Python

```python
max_delay = air.DepDelay.max().compute()
mean_delay = air.DepDelay.mean().compute()
```

- 计算中位数 (median) 会很慢甚至不可行，因为它涉及大量数据移动 (shuffle)

2.3.3 示例：分组聚合 (Split-Apply-Combine)

Python

```python
sub = air[(air.UniqueCarrier == 'UA') & (air.Origin == 'SFO')]
byDest = sub.groupby('Dest').DepDelay.mean()
results = byDest.compute()
```

⚠ Remark

多次调用 `.compute()` 会重复读取数据，应尽量一次计算完成

⚠ Remark: 结果内存占用

- `.compute()` 返回的结果是普通 Python 对象
- 若结果数据过大，可能导致内存溢出

2.4 Dask Bag (类 Python 列表)

- **结构**：类似 Python 的 list，但无固定顺序
- **默认调度器**：processes
- 适合执行类似并行 map 的操作

2.4.1 示例：Wikipedia 日志数据

Python

```python
import dask.multiprocessing
dask.config.set(scheduler='processes', num_workers=4)
```

```python
import dask.bag as db

path = '/scratch/users/paciorek/wikistats/dated_2017_small/dated/'
wiki = db.read_text(path + 'part-0*gz')
n = wiki.count().compute()
```

- 计算记录数约需 136 秒 (完整数据集)

2.4.2 示例：过滤操作

Python

```python
import re

def find(line, regex='Armenia'):
    vals = line.split(' ')
    if len(vals) < 6:
        return False
    tmp = re.search(regex, vals[3])
    return tmp is not None

armenia = wiki.filter(find)
result = armenia.count().compute()
```

Remark

应避免多次 `.compute()` 调用，应将所有操作构建完毕后统一执行

2.4.3 示例：转换为 DataFrame

Python

```python
def make_tuple(line):
    return tuple(line.split(' '))

dtypes = {'date': 'object', 'time': 'object', 'language': 'object',
          'webpage': 'object', 'hits': 'float64', 'size': 'float64'}

df = armenia.map(make_tuple).to_dataframe(dtypes)
result = df.compute()
```

2.5 Dask Array (类 NumPy 数组)

- **结构**：数组被分块 (by rows and columns) 存储为多个 NumPy 子数组
- **默认调度器**：threads
- 适用于大规模矩阵计算，但非所有 NumPy 操作都支持

2.5.1 示例：单节点数组操作

Python

```python
import dask
dask.config.set(scheduler='threads', num_workers=4)
import dask.array as da

x = da.random.normal(0, 1, size=(40000, 40000), chunks=(10000, 10000))
```

```
mycalc = da.mean(x, axis=1)
rs = mycalc.compute()
```

- 可在单机上高效处理 13 GB 数组
- Dask 通过 lazy evaluation 和 chuck，仅在需要结果时才加载数据

> ⚠ **Remark**
>
> 对于 mean 这种 row-based operation, 我们可能会考虑只按行进行分块, 即 `x = da.random.normal(0, 1, size=(40000,40000), chunks=(2500, 40000))`, 但实际的运行速度并没有明显差别, 因为均值计算可以分段进行, 并且只有少量的 summary statistics 需要在 workers 之间 move

2.5.2 示例：更大数组 (80 GB)

Python

```
x = da.random.normal(0, 1, size=(100000,100000), chunks=(10000,10000))
mycalc = da.mean(x, axis=1)
rs = mycalc.compute()
```

> ⚠ **Remark**
>
> 尽管数组理论上占用 80 GB，实际内存占用仅数 GB，因为 Dask 动态生成块而非一次性加载全部

2.6 分布式数组 (Distributed Arrays)

- 使用 Dask Distributed 可将数组分布到多台机器上计算
- 但需避免从单一进程分发数据，否则会触发大量网络复制