Distributed Matrix Multiplication: A Comparative Study of MapReduce in Java and Python

https://github.com/klapa0/big_data_studies

Kacper Klasen FB390707

December 21, 2025

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data analysis. While dense matrix multiplication can be efficiently implemented on a single machine, very large matrices require distributed computation to scale effectively. In this study, we benchmarked distributed block matrix multiplication implemented in:

1. **Java MapReduce** using Hadoop in local mode.
2. **Python** using MrJob.

The goal of this experiment was to evaluate the performance and scaling behavior of block-based distributed multiplication with different matrix sizes.

2 Methodology

2.1 Block Matrix Multiplication

The distributed multiplication is implemented using the block decomposition approach:

1. Matrices A and B of size $N \times N$ are divided into $G \times G$ blocks, each of size $blockSize = N/G$.
2. Each block $A_{i,k}$ and $B_{k,j}$ is processed independently in a distributed fashion:
   - **Mapper**: Distributes blocks to reducers responsible for computing submatrices $C_{i,j}$.
   - **Reducer**: Performs local multiplication of matching blocks and sums the results to produce the corresponding block in C.

2.2 Java Implementation (Hadoop MapReduce)

- Mapper emits blocks with keys corresponding to the output block index (i, j).
- Reducer reconstructs local dense blocks from sparse input data and performs multiplication using a cache-friendly $i - k - j$ loop order.
- Hadoop is executed in local mode to simplify testing without HDFS.

2.3 Python Implementation (MrJob)

- MrJob executes locally.
- Sparse block matrices are represented as key-value pairs. The `mapper` and `reducer` functions in MRJob handle the data distribution and aggregation logic.

2.4 Benchmark Configuration

- Matrix sizes N: 100, 200, 300, ..., 1000.
- Grid dimension G: scales with matrix size ($gridDim = max(2, N/100)$).
- Repetitions: 5 measured runs + 2 warmup runs.
- Metrics: Execution time in milliseconds and total number of output elements.

3 Results

3.1 Java MapReduce Benchmark

Matrix Size	Grid	Block Size	Average Time [ms]
100x100	2x2	50	1109
200x200	2x2	100	1103
300x300	3x3	100	1100
400x400	4x4	100	1100
500x500	5x5	100	1077
600x600	6x6	100	2084
700x700	7x7	100	2281
800x800	8x8	100	3077
900x900	9x9	100	5078
1000x1000	10x10	100	7078

Table 1: Average execution times for Java MapReduce (local mode, excluding warmup runs).

3.2 Python MRJob

Matrix Size	Average Time [ms]
100x100	166.8
500x500	2023.6
1000x1000	12182.3

Table 2: Average execution times for Python MRJob.

4 Frequent Itemset Mining with Python MapReduce

4.1 Introduction

Frequent itemset mining is a fundamental task in data mining, aimed at discovering items that commonly appear together in transactional datasets. This operation is widely used for market basket analysis, recommendation systems, and association rule mining.

4.2 Methodology

For this experiment, we used a small synthetic dataset consisting of 5 transactions:

```
1, milk, bread, butter
2, milk, bread
3, bread, butter
4, eggs, milk
5, milk, bread, butter
```

The MapReduce approach in Python (using MRJob) was employed to count the occurrences of each item. A minimum support threshold of 3 was applied to filter frequent items. The process involved:

1. **Mapping:** Each item in a transaction is emitted with a count of 1.
2. **Reducing:** Counts are aggregated for each item, and only items meeting or exceeding the minimum support are retained.

4.3 Results

The frequent items identified with minimum support 3 are summarized in Table 3.

Item	Count
bread	4
milk	4
butter	3

Table 3: Frequent items discovered in the transaction dataset with minimum support of 3.

4.4 Analysis

The results indicate that *bread* and *milk* appear in 4 transactions, while *butter* appears in 3. *Eggs* does not meet the minimum support threshold and is excluded from the frequent itemset. This simple example demonstrates the effectiveness of the MapReduce paradigm for distributed counting: mappers handle local counts and reducers aggregate globally.

4.5 Conclusion

This experiment highlights how Python MRJob can be used to perform distributed frequent itemset mining. While the dataset is small, the methodology scales naturally to much larger transaction datasets across multiple nodes, making it a practical tool for real-world data mining applications.

5 Analysis

- **Overhead Dominates for Small Matrices**: Hadoop's local mode shows nearly constant runtime for $N \leq 500$ due to job setup overhead.
- **Grid Size Matters**: Larger matrices require more blocks to effectively distribute computation. Fixed small grids give misleadingly constant times.
- **Python vs Java**: Python MRJob is faster for small-scale tests due to lower framework overhead. Java Hadoop becomes competitive as matrix size grows.
- **Sparsity and Block Optimization**: Sparse blocks reduce computation, but for small matrices, the overhead dominates.

6 Conclusion

Distributed block matrix multiplication works correctly in both Java MapReduce and Python MRJob. For benchmarking, overhead dominates small workloads, so careful selection of grid dimension and block size is essential. Framework choice affects performance: Hadoop MapReduce has higher setup overhead, suitable for large datasets or clusters, while MRJob is lighter-weight, better for prototyping and small-to-medium workloads. Sparse representation is critical for very large matrices.

References

- ChatGPT, Assistance with LaTeX formatting and benchmark description. https://chat.openai.com/