MapReduce Algorithms for Large-Scale Matrix Multiplication

Name: Caroline Gakii

Registration Number: CT404/204746/24

Lecturer: Dr Amos Chege

Institution: Meru University of Science & Technology

Date: 15th July 2025

Abstract

As data volume increases in scientific, industrial, and commercial use, scalable matrix computation becomes more and more essential. The existing matrix multiplication algorithms are not efficient at scale because of computation and memory limitations. This work discusses how MapReduce, a distributed computational model, can be utilized to implement and streamline large-scale matrix multiplication. Two methods are implemented: single step MapReduce and multi-step approach. The scalability and performance of the two are compared empirically using synthetic data. The experiments exhibit the trade-offs between communication overhead and computational complexity that inform practitioners with large matrices when using distributed computing.

1. Introduction

Matrix multiplication is a simple operation in data mining, scientific computing, and machine learning. However, when the data size is large, the old-fashioned way is out of the question because centralized memory and computation power have limitations. MapReduce programming model solves this issue by offering parallel computation across dispersed systems (Leskovec, et al., 2020). This paper investigates two matrix multiplication algorithms based on MapReduce with one MapReduce step and multiple MapReduce steps, respectively, and compares their performance when processing ever-larger synthetic datasets.

We refer to the early work in Mining of Massive Datasets (Leskovec, et al., 2020), Sections 2.3.9 and 2.3.10, where two main algorithmic solutions for matrix multiplication in MapReduce systems are given.

3. Methodology

3.1 Tools and Environment

The programming of matrix multiplication algorithms was conducted using the Python programming language (version 3.10) owing to its ease of use, broad ecosystem, and excellent support for scientific computing. The principal development environment was Jupyter Notebook, which provides an interactive and dynamic environment for code composition and execution, especially useful for visualization and step-by-step verification.

The work was based on some standard Python libraries. Python NumPy was used for efficient numerical calculation and matrix operation, and time module was used to measure execution time and see the performance. Experiment was performed on a local laptop with 8GB RAM, which provided sufficient computing power for synthetic data of this research.

The input data employed were synthetic square matrices of varying sizes—10×10, 50×50, 100×100, and 200×200—and were generated randomly. This was the setup for comparing performance at ascending scales with input data properties in check.

3.2 Implementation

Two MapReduce implementations of matrix multiplication were programmed following the concepts described in Sections 2.3.9 and 2.3.10 of Mining of Massive Datasets by Leskovec et al. (2020).

The single-step MapReduce implementation consists of a clean design (Leskovec et al., 2020, Section 2.3.10). One mapper generates intermediate key-value pairs of the form ((i, j), partial_product), where i and j are row and column indices of the product matrix, and partial_product is the product of the respective elements of matrices A and B. The reducer then takes all partial products for a specific key (i, j) and sums them up to compute the final value at a specific location in the resulting matrix C.

The two-phase multi-step MapReduce algorithm is slightly more complex and splits the procedure into two separate phases (Leskovec et al., 2020, Section 2.3.9). Mappers, during the first step, compute and emit partial products with keys related to the entries of the final matrix. In the second step, the reducers sum these values by adding together all partial products for each (i, j) key. This multi-step process mimics the way large distributed systems move and re-shuffle data around jobs and was included to measure scalability and architecture trade-offs.

3.3 Dataset Preparation

For the purpose of estimating the performance of both methods, synthetic datasets were created based on the numpy.random.randint() function. Four square matrices of varying sizes—10×10, 50×50, 100×100, and 200×200—were created using this method. Every matrix had randomly generated integers in a defined range, mimicking real-world situations without creating external dataset dependencies.

The experiments were executed sequentially within the identical hardware and software environment to ensure consistency in timing and output comparison. In controlled conditions, it was simple to equate differences in algorithmic running time and scalability to the algorithm itself independent of external factors.

4. Results

Matrix Size	Single-Step Time (sec)	Multi-Step Time (sec)
10×10	0.0758	0.0757
50×50	0.1416	0.1169
100×100	1.0081	0.9715
200×200	97.1220	11.9471

The multi-step algorithm outperformed the single-step algorithm at sizes of 100×100 and above. Both algorithms had exponentially increasing performance times with size but proportionally more so for the single-step algorithm. To add on that, both algorithms generated the same resultant matrices, confirming correctness.

5. Discussion

The experiment outcomes with both single-step and multi-step MapReduce algorithms provide valuable insights into their respective performance characteristics and scalability when both are applied to matrix multiplication.

For the smaller matrices such as 10×10 and 50×50, run times using both approaches were virtually identical to one another and with little fluctuation from system-level discrepancies of processing. These minor differences suggest that for small-scale computations, either approach would be adequate and the choice might boil down to code readability or individual implementation versus efficiency.

However, once the matrix size hit 100×100 and particularly 200×200, the multi-step MapReduce algorithm dramatically outperformed. With 200×200, the single-step solution took about 97 seconds, while the multi-step solution ran the operation in about 12 seconds. This differential performance can be attributed almost entirely to the fact that the multi-step approach truly

decouples the computation and aggregation stages, enabling better organization of the intermediate data and lesser redundancy in computation (Leskovec et al., 2020).

The multi-step solution performance improvement also illustrates the benefits to distributed systems from breaking tasks down into smaller steps. In a real MapReduce environment (e.g., Hadoop or Spark), such a pattern would allow for parallel computation and shuffling of data between nodes, offloading load from individual mappers and increasing throughput (Leskovec et al., 2020). Whereas this experiment was conducted on a single machine with a simulation of MapReduce, the multi-step version nonetheless illustrated structural advantages that gave faster execution for larger data.

Although being more scalable, the multi-stage method is inherently more wordy and complex. It has more lines of code, a few phases, which include passing data, as well as additional logic to total up the data. These are more challenging to implement and debug, especially for newcomers or in cases where there are limited resources.

In short, while the single-step algorithm may be suitable for smaller matrices or less complicated use cases, the multi-step algorithm is superior in terms of both performance and scalability for large-scale matrix multiplication operations. This agrees with actual design decisions in large data systems, where dividing computations into pipeline stages enhances efficiency and fault tolerance.

Reference

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of Massive Datasets* (3rd ed.). Cambridge University Press.