Parallel Matrix Multiplication Benchmarking

Yang, Jia Hao

University of Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain.

Abstract

This study investigates matrix multiplication using MapReduce in Python with the MrJob library, aiming to assess its efficiency compared to classical algorithms. The implemented methodology involves a three-step process, including mapping, multiplication, and summation within the MapReduce paradigm. Surprisingly, experimental results reveal suboptimal performance, prompting consideration of alternative approaches such as GPU-accelerated methods. The conclusion suggests that MapReduce may not be the most suitable choice for matrix multiplication, calling for further research into parallel processing techniques, GPU acceleration, algorithmic optimizations, and hybrid approaches. Future work could explore scalability, benchmarking, integration with machine learning, and the integration of matrix multiplication tasks within broader big data processing pipelines. This study provides insights into the limitations of MapReduce for matrix multiplication and proposes avenues for future research to enhance computational efficiency in diverse computational contexts.

Introduction

During the course of this project, our assigned task has centered around the execution of matrix multiplication—an increasingly pivotal subject in contemporary contexts due to its pronounced relevance in various technological domains, including but not limited to neural network training and modeling, network theory, solution of systems of linear equations, and graphics processing within the realm of video game development. To elucidate this concept further, we undertake a comparative analysis of a widely adopted programming technique conceived in the 1980s by Doug Cutting, known as MapReduce. Originally integrated into Hadoop to support the distributed processing requirements of the Nutch search engine project, MapReduce operates primarily through two fundamental operations: a Map operation and a Reduce operation. Generally, the data structures within a MapReduce paradigm are represented as tuple-value pairs. Succinctly, the Map operation is defined by the transformation $Map(k_1, v_1) \rightarrow List(k_2, v_2)$, while the Reduce operation is expressed as $Reduce(k_2, List(v_2)) \rightarrow List(v_3)$. Our implementation will be realized in Python, employing the MrJob library, which facilitates matrix multiplication in a manner simulating the distributed environment characteristic of Hadoop.

Methodology

The methodology employed for the implementation of matrix multiplication using MrJob in Python is characterized by its simplicity and comprises three principal methods. This implementation is encapsulated within a class that inherits from the MrJob class. The first method pertains to the mapping phase, wherein key-value pairs are extracted. Subsequently, the second method corresponds to the reduction phase, involving the multiplication of values originating from matrix A and matrix B, guided by a key established during the mapping operation. The final step encompasses an additional reduction, primarily focused on the summation of the outcomes derived from the preceding multiplication reduction. In the subsequent sections, a more elaborate elucidation of these methods will be presented.

Mapping

The mapping operation necessitates that the matrix file explicitly denotes whether it corresponds to matrix A or matrix B within the filename. This ensures the correct emission of key-value pairs contingent upon the matrix type. Specifically, when the matrix belongs to matrix A, the method emits a key-value pair in the form of (row, i) and $(0, col, value)$. Conversely, for matrix B, the emission comprises a key-value pair in the configuration of (j, col) and $(1, row, value)$. Notably, the utilization of the 'yield' keyword in Python is emphasized, as it generates a memory-efficient iterator, consuming significantly less memory compared to conventional data structures such as lists in Python.

Reduction Multiply

In the ensuing method, the multiplication of matrix values is addressed through the examination of key-value pairs. The determination of the matrix type is facilitated by inspecting the second element of the value tuple, as elucidated earlier in the discussion, where the tuple takes the form $(0, col, value)$ for matrix A and $(1, row, value)$ for matrix B. Consequently, the method proceeds to identify pairs with matching values in this second element, upon which the multiplication operation is executed.

Reduction Sum

The responsibility of this method lies in the aggregation of values and the subsequent generation of a text file representing the resulting matrix. The summation of values is performed, and the outcome is then written to a designated text file, effectively capturing the representation of the resultant matrix.

Experiments

The conducted experiments yielded unexpected results. Initial expectations might incline towards the MapReduce programming technique outperforming classical programming algorithms; however, the observed execution times were notably suboptimal. This raises the possibility that the MapReduce technique may not be inherently suitable for matrix multiplication, especially when compared to more sophisticated and efficient approaches, such as GPU-accelerated matrix multiplications. Subsequently, graphical representations of the obtained results for matrices of varying sizes will be presented in the following sections.

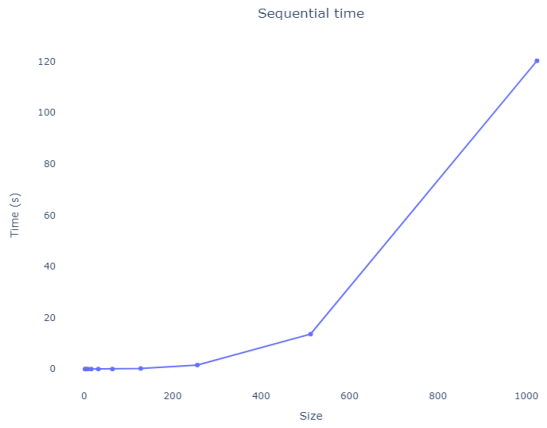

Figure 1: Sequential Matrix Multiplication execution time

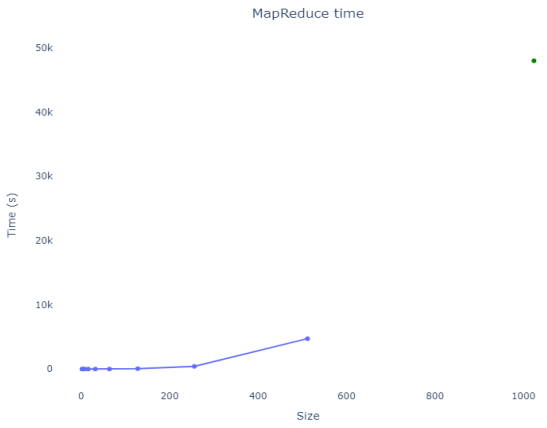

Figure 2: Map Reduce Matrix Multiplication execution time

It is imperative to note that the green data point on the graphs does not correspond to an actual measured value; rather, it represents a predicted estimation of the time required for the execution of

a 1024x1024 matrix. This predictive value was included due to the extended duration of execution, rendering it impractical to obtain a real-time measurement.

Finally, a consolidated view of both graphs is presented, enabling a comprehensive comparison of the observed trends and performances across different matrix sizes and a table with the execution times.

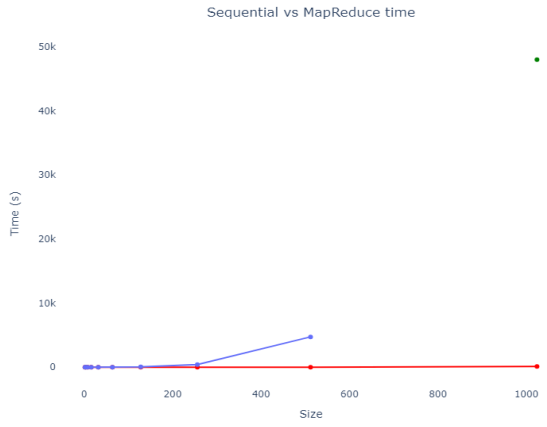

Figure 3: Sequential vs Map Reduce Matrix Multiplication execution time

sizes	sequential	map reduce
2	0.000000	0.331921
4	0.000000	0.782201
8	0.000000	1.159106
16	0.000503	1.409268
32	0.003000	2.568929
64	0.024374	7.617075
128	0.189823	48.587793
256	1.537851	416.139312
512	13.605884	4735.060078
1024	120.207050	-

Figure 4: Times table of both implementations

Conclusions

In conclusion, our investigation into matrix multiplication using the MapReduce programming technique, specifically implemented through Python and the MrJob library, has yielded unexpected and suboptimal results. Contrary to initial expectations, where MapReduce was anticipated to outperform classical programming algorithms, the observed execution times were notably poor. This prompts consideration of the viability of MapReduce for matrix multiplication tasks, especially when contrasted with more advanced approaches, such as GPU-accelerated matrix multiplications.

The outlined methodology involved a three-step process, including mapping, multiplication, and summation, each executed within the MapReduce paradigm. The mapping operation relied on explicit matrix type indications within filenames, emitting key-value pairs accordingly. The multiplication method facilitated the multiplication of matrix values based on key-value pairs, guided by the second element of the value tuple denoting matrix type. The final step involved the summation of values and the generation of a resulting matrix represented in a text file.

The experimental results, depicted through graphical representations for matrices of varying sizes, showcased the inefficiency of the MapReduce technique in comparison to classical algorithms. Furthermore, a predictive green data point was introduced for a 1024x1024 matrix due to impractical execution times.

Considering these findings, it is evident that MapReduce may not be the most suitable choice for matrix multiplication tasks, particularly when alternative advanced techniques, like GPU-accelerated methods, offer more efficient solutions. Further research and exploration of alternative parallel processing methodologies may be warranted to optimize matrix multiplication tasks in diverse computational contexts.

Future work

Future work in this domain could explore several avenues to enhance the understanding of matrix multiplication and optimize its computational efficiency. Some potential directions for future research include:

- **Parallel Processing Techniques:** Investigate and implement alternative parallel processing techniques beyond MapReduce, such as Apache Spark or distributed computing frameworks optimized for matrix operations. This could provide insights into the comparative performance of different parallelization approaches.
- **GPU Acceleration:** Explore the utilization of GPU-accelerated matrix multiplication techniques. Modern GPUs are well-suited for parallel computation and can significantly expedite matrix operations. Investigating frameworks like CUDA or OpenCL for GPU acceleration may offer performance improvements.
- **Algorithmic Optimization:** Delve into algorithmic improvements for matrix multiplication. Consider exploring more sophisticated algorithms, such as Strassen's algorithm or parallelized versions of traditional algorithms, to enhance computational efficiency.
- **Hybrid Approaches:** Investigate hybrid approaches that combine the strengths of multiple techniques, such as using MapReduce for specific preprocessing tasks and transitioning to GPU-accelerated methods for the core matrix multiplication steps.

References

[1] **Map Reduce Programming. ULPGC**

[2] **Map Reduce. Wikipedia.** https://es.wikipedia.org/wiki/MapReduce