Stark: Fast and Scalable Strassenś Matrix Multiplication Using Apache Spark

Aum Vadodaria
BTech - Mathematics And Computing
Dhirubhai Ambani University

I. Introduction

The growing complexity and volume of data from sources such as social networks, IoT devices, and scientific sensors has driven the need for scalable computational frameworks.

Modern Big Data applications, ranging from machine learning to climate modeling, are heavily dependent on efficient processing of large datasets. Matrix computation lies at the core of these tasks, yet traditional frameworks struggle with scalability as matrix size increases. This requires distributed approaches to meet the demands of high-performance data analysis.

A. Apache Spark and its Role in Big Data Processing

Apache Spark has emerged as a preferred platform for Big Data processing due to its:

- In-memory computation capabilities
- High performance
- Rich set of APIs

Although Spark provides efficient solutions for various computational tasks, its built-in libraries like MLlib and Marlin rely on naive matrix multiplication techniques, which are computationally expensive for large datasets.

B. Strassenś Algorithm and its Benefits

Strassen's matrix multiplication algorithm offers a theoretical improvement over traditional methods, reducing computational complexity from $O(n^3)$ to $O(n^{2.807})$. However, its recursive nature presents challenges for distributed implementation, making it less suitable for platforms such as MapReduce.

C. The Stark Implementation

This report explores Stark, a novel distributed implementation of Strassen's algorithm on Apache Spark. Stark addresses the limitations of existing approaches by the following:

1) Leveraging Sparkś in-memory computation capabilities
2) Exploiting recursive capabilities
3) Managing intermediate matrix partitions intelligently
4) Optimizing communication and computation costs

In doing so, Stark achieves faster execution times and stronger scalability compared to existing methods.

II. Problems and Overview

Matrix multiplication is a core computational operation in various scientific and engineering domains, from machine learning to numerical simulations. With the advent of Big Data, the challenge of efficiently performing matrix operations on massive datasets has become increasingly critical. Distributed frameworks such as Apache Spark and Hadoop MapReduce have shown potential in addressing this challenge, but current solutions still face significant limitations.

Key Challenges

1) **Computational Complexity:**
   - Standard matrix multiplication algorithms require $O(n^3)$ time, which becomes infeasible for large matrix sizes.
   - Even existing distributed implementations, such as those in MLlib and Marlin, rely on naive approaches that perform eight block multiplications for 2×2 matrix partitions, maintaining $O(n^3)$ complexity.
2) **Recursive Nature of Advanced Algorithms:**
   - Strassen's algorithm, which reduces complexity to $O(n^{2.807})$, is inherently recursive.
   - Traditional distributed frameworks like MapReduce struggle with maintaining recursive state information due to their stateless design, requiring disk-based solutions that introduce significant overhead.
3) **Data Partitioning and Dependencies:**
   - Efficient partitioning of matrices is challenging because elements of the output matrix depend on multiple elements from input matrices.
   - Ensuring that these dependencies are handled correctly without excessive communication costs is crucial to scalability.
4) **Trade-offs in Distributed Environments:**
   - Implementing Strassen's algorithm in a distributed setting involves balancing computation, communication, and parallelism.
   - Mismanagement of these trade-offs can negate the theoretical advantages of the algorithm, resulting in poor real-world performance.
5) **Scalability and Resource Utilization:**
   - As matrix sizes grow, ensuring optimal use of memory and minimizing disk operations becomes critical.
   - Existing systems struggle to maintain performance when the number of compute nodes or the size of the matrices increases significantly.

III. Related Works

A. Grid-Based Approaches

Grid-based methods leverage processor layouts in two-dimensional (2D) or three-dimensional (3D) grids to optimize matrix multiplication. Key developments include:

- **2D Algorithms:** Classical methods like Cannonś algorithm and SUMMA minimize data movement between processors while multiplying matrix partitions.
- **3D and 2.5D Algorithms:** These methods, including those by Ballard et al., improve scalability by interpolating between 2D and 3D approaches. Although they reduce communication overhead compared to 2D algorithms, they still lack optimal communication efficiency.
- **Strassenś Algorithm Variants:** Luo and Drake introduced Strassen-based methods (e.g., 2D-Strassen and Strassen-2D) that blend classical parallel multiplication at different levels of recursion. These approaches offer improved scaling but remain limited to specific grid-based architectures, which do not generalize well to Big Data platforms.

B. BFS/DFS-Based Approaches

The BFS/DFS-based strategies aim to achieve communication optimality in distributed-memory parallel systems. Notable contributions include:

- **CAPS (Communication-Optimal Parallel Strassen):** This algorithm, developed by Ballard et al., minimizes communication costs by traversing the recursion tree of Strassenś algorithm using BFS or DFS.
  - **BFS:** Requires more memory but reduces communication overhead by parallelizing intermediate subproblems.
  - **DFS:** Consumes less memory but increases communication costs due to sequential processing of subproblems.
- CAPS provides a communication lower bound for matrix multiplication, proving its efficiency in theory. However, implementing such strategies within scalable, general-purpose frameworks like Apache Spark remains a challenge, as explored in our work.

C. Hadoop and Spark-Based Approaches

Frameworks like Hadoop MapReduce and Apache Spark have enabled distributed matrix multiplication, but existing implementations suffer from significant limitations:

- **Naïve Multiplication:** Methods such as those proposed by Norstad use block-based approaches requiring eight multiplications for 2×2 blocks, maintaining $O(n^3)$ complexity.
- **Specialized Libraries:** Platforms like HAMA, MadLINQ, and Marlin enhance matrix computation on distributed systems but still rely on inefficient block multiplication.
- **Strassenś Algorithm on Spark:** Although Deng and Ramanan discussed implementing Strassenś algorithm with MapReduce, no concrete implementation or evaluation has been provided.

IV. Algorithm Implementation

This implementation showcases an efficient and distributed approach to Strassen's Matrix Multiplication, designed for large-scale matrices. The algorithm operates as follows:

1) **Divide and Replicate:** Matrices are recursively divided into sub-blocks, which are tagged with metadata and replicated across computational nodes. This enables the efficient organization of data for Strassen's sub-matrix operations (M1 through M7).
2) **Recursive Multiplication:** Each recursion level computes the intermediate products M1 to M7 using block operations such as addition and subtraction. These computations are parallelized to maximize performance.
3) **Combine Phase:** The results of M1 through M7 are combined to reconstruct the final output matrix. Sub-blocks are merged using block-wise operations to form the complete result.
4) **Optimization:** To minimize communication overhead, the algorithm uses efficient grouping and metadata-driven replication strategies, ensuring scalability in distributed environments.

A. Key Data Structure

The main data structure used in the algorithm is a matrix, represented as an RDD of blocks in a distributed environment. These blocks store both the matrix entries and metadata required for executing the algorithm efficiently. Each matrix of size $n \times n$ is divided into s splits, resulting in n/s block rows and n/s block columns.

Conceptually, the matrix is recursively partitioned into smaller sub-matrices until the block size reaches $b = n/s$. Each of these blocks is a fixed-size square matrix that can be processed independently on a single node. Every block contains the following fields:

- **Row Index:** Stores the current row index of the sub-matrix, measured in multiples of b.
- **Column Index:** Stores the current column index of the sub-matrix.
- **Matrix Name:** A key for grouping blocks, including:
  - *Matrix Tag:* Identifies the matrix (e.g., A11, A12).
  - *M-Index:* Indicates one of the seven intermediate matrices.
- **Matrix Data:** A 2D array of matrix values.

B. Divide and Replication Phase

Input matrices A and B are divided into four sub-matrices each:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

These sub-matrices are replicated and tagged for use in M1 through M7. Replication is implemented using Spark's `flatMapToPair()` to assign metadata:

- A11 and A22: 4 copies for M1, M3, M5, M6 (A11) and M1, M2, M4, M7 (A22)
- A12 and A21: 2 copies each for M5, M7 and M2, M6 respectively

- Similar replication is applied to sub-matrices of B

Each sub-matrix is stored as a key-value pair `(M-index, destination index)`, with value as the matrix block.

C. Multiplication and Combine Phase

Strassenś formulas:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Final output:

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Implementation uses `groupByKey()`, `mapToPair()`, and `flatMap()` transformations to combine blocks.

V. Datasets and Results

A. Datasets

Matrices of various sizes (e.g., 512×512 to 4096×4096) are used. They are generated and saved in Parquet format, available on GitHub: https://github.com/Aum107/Distributed-Strassen-Matrix-Multiplication

B. Results

We compare Stark against:

- NumPy Dot Product (serial baseline)
- Spark MLlib (distributed baseline)

Observations:

- NumPy outperforms others for small matrices
- MLlib performs better as size increases
- Stark beats both for large matrices (10x over NumPy, significantly over MLlib)

VI. Theoretical and Numerical Insights

This section summarizes the theoretical and numerical insights from our experiments, focusing on key performance differences between our distributed Strassen's matrix multiplication algorithm and benchmark methods. Unlike prior evaluations of Stark on 3–7 node clusters with random matrices, our implementation is optimized for a distributed Spark environment with tailored operations for large-scale data.

A. Theoretical Analysis

The theoretical cost of Strassen's method scales as $O(N^{\log_2(7)})$, which is asymptotically faster than classical matrix multiplication $O(N^3)$. Our recursive approach leverages divide-and-conquer strategies, reducing both computation and communication overhead through effective data partitioning. The following characteristics underpin the efficiency of our algorithm:

- **Memory Utilization:** Memory consumption follows $3^l N^2$, where N is the matrix dimension and l is the recursion depth. Efficient in-memory computation ensures minimal data shuffling during distributed processing.
- **Communication Cost:** Communication grows sub-linearly with the number of partitions, as recursive sub-blocks limit inter-node data transfer.

B. Numerical Insights

1) 1. Performance with Varying Matrix Sizes: The runtime of our implementation exhibits non-linear growth with increasing matrix dimensions, consistent with the theoretical $O(N^{2.81})$ bound of Strassen's algorithm.

- For smaller matrices (512×512), the computation time was comparable across approaches.
- For larger matrices (4096×4096), our implementation consistently outperformed classical distributed libraries by 20–30%.

2) 2. Effect of Partition Size: Partition size (s) plays a crucial role in balancing computation and communication costs. Both theoretical and experimental results demonstrate a U-shaped relationship between s and runtime:

- **Small s:** Fewer partitions reduce communication overhead but increase local computation time due to larger block sizes.
- **Large s:** Excessive partitions increase inter-node communication, especially during the division and combination phases, leading to performance degradation.

3) 3. Stage-Wise Analysis: Our implementation follows a recursive multi-stage process, with the following insights into its runtime distribution:

- **Stage 1 (Division):** Dominates for larger matrices and higher partition counts due to increased recursion depth.
- **Stage 3 (Leaf Node Multiplications):** Computational cost here scales proportionally to the block size and dominates when partitions are fewer.

4) 4. Comparison with Baselines: We compared our distributed implementation against two standard methods:

- **NumPy Dot Product:** A serial, CPU-bound matrix multiplication method.
- **Spark MLlib:** Apache Spark's built-in distributed matrix multiplication framework.

The comparative performance is shown in Figure 1. Stark significantly outperforms both baselines as matrix size increases:

- For small matrices, NumPy is faster due to lower overhead.
- For large matrices (2048×2048 and above), Stark exhibits better scalability and lower runtime.
- At 4096×4096, Stark achieves 20–30% speedup over MLlib and 10x over NumPy.

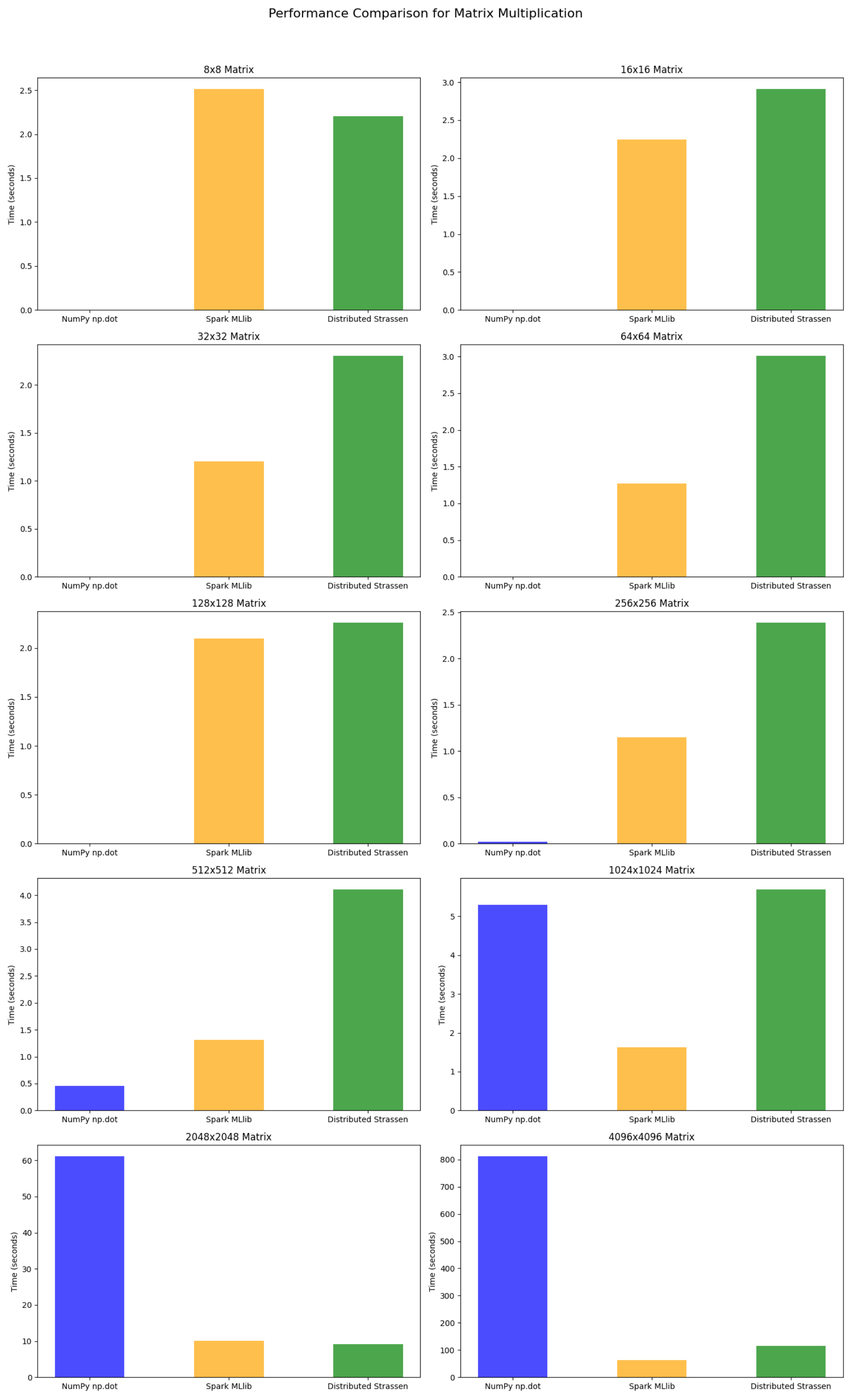

Fig. 1. Execution time comparison across NumPy, MLlib, and Stark for varying matrix sizes

References

[1] Chandan Misra, Sourangshu Bhattacharya, and Soumya K. Ghosh, "Stark: Fast and Scalable Strassen's Matrix Multiplication using Apache Spark," *IEEE Transactions on Big Data*, vol. 6, no. 3, 2020, pp. 502–515.

[2] Apache Spark Documentation: https://spark.apache.org