BIG DATA PROCESSING

(IE494)

TERM PROJECT REPORT

Stark: Fast and Scalable Strassen's Matrix Multiplication Using Apache Spark

---

Aditya Patel
202203027

Dharmesh Kota
202203038

Contents

1 Introduction

The rapid growth in the volume and complexity of datasets generated from diverse sources has underscored the need for robust and scalable computational frameworks. These datasets come from various sources, such as Social media, IoT devices and Scientific sensors.

Big Data applications in fields such as **machine learning**, **climate modeling**, and **analytics** increasingly rely on efficient processing of large-scale datasets. At the heart of many such applications lies **matrix computation**, a fundamental operation in numerical and data-driven analysis. Traditional computation frameworks struggle with the demands of large matrix sizes, necessitating **distributed approaches** for scalable processing.

1.1 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

While 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.

1.2 Strassen's 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 like MapReduce.

1.3 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:

1. Leveraging Spark's in-memory computation capabilities
2. Exploiting recursive capabilities
3. Managing intermediate matrix partitions intelligently

4. Optimizing communication and computation costs

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

2 Problems and Overview

Matrix multiplication is a core computational operation across 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 like **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 2x2 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 for 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.

3 Related Works

The challenge of efficiently performing distributed matrix multiplication has been extensively studied across various domains, leading to the development of numerous approaches tailored to specific computational frameworks and algorithms. These approaches can broadly be categorized into three main groups: **Grid-Based Methods**, **BFS/DFS-Based Methods**, and **Hadoop/Spark-Based Methods**. Below, we discuss key contributions within these categories and their relevance to the present work.

3.1 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 Cannon's 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.
- **Strassen's 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.

3.2 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 Strassen's algorithm using BFS or DFS.
  - **BFS**: Requires more memory but reduces communication overhead by parallelizing intermediate sub-problems.
  - **DFS**: Consumes less memory but increases communication costs due to sequential processing of sub-problems.
- 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.

3.3 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.
  - For instance, Marlin reduces communication costs using Spark's join transformation but performs 8 block multiplications, limiting its efficiency compared to 7 multiplications in Stark.
- **Strassen's Algorithm on Spark**: Although Deng and Ramanan discussed implementing Strassen's algorithm with MapReduce, no concrete implementation or evaluation has been provided.

4 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 (M_1 through M_7).

2. **Recursive Multiplication**: Each recursion level computes the intermediate products M_1 to M_7 using block operations such as addition and subtraction. These computations are parallelized to maximize performance.

3. **Combine Phase**: The results of M_1 through M_7 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.

By leveraging the recursive nature of Strassen's method and integrating distributed computing techniques, this algorithm achieves a balance between computational efficiency and scalability, making it suitable for large matrix multiplication tasks.

4.1 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:

1. **Row Index:** This field stores the current row index of the sub-matrix, measured in multiples of b. As the matrix is divided during the algorithm, these indices are updated to track the new positions of the sub-matrices.

2. **Column Index:** Similar to the row index, this field keeps track of the current column index of the sub-matrix in multiples of b.

3. **Matrix Name:** This field is used as a key for grouping the blocks during computation. It consists of:

(a) **Matrix Tag:** A label identifying the matrix, such as A, B, or their sub-matrices (e.g., A_{11}, A_{12}, A_{21}, A_{22}).

(b) **M-Index:** An index denoting one of the seven intermediate matrices (M_1 to M_7) computed during Strassen's algorithm.

4. **Matrix Data:** A 2D array storing the matrix values of the block. This field holds the actual numerical entries of the sub-matrix.

This block structure is central to the distributed matrix multiplication algorithm, as it allows for parallel processing, data replication, and metadata tracking necessary for efficient computation.

4.2 Divide and Replication Phase

In this phase, the input matrices A and B are divided into four sub-matrices of equal size, denoted as A_{11}, A_{12}, A_{21}, and A_{22}, as well as B_{11}, B_{12}, B_{21}, and B_{22}. These sub-matrices are further replicated and annotated with metadata to facilitate the computation of the seven intermediate matrices M_1 to M_7 using Strassen's algorithm.

The **divide step** partitions the original matrix blocks into smaller sub-blocks as follows:

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

Each resulting sub-matrix is assigned coordinates relative to the original matrix block for tracking during recursive computations.

The **replication step** ensures that sufficient copies of these sub-matrices are made to compute all intermediate matrices. Specifically:

- Four copies of A_{11} and A_{22} are created, with M-indices M_1, M_3, M_5, and M_6 for A_{11}, and M_1, M_2, M_4, and M_7 for A_{22}.
- Two copies of A_{12} and A_{21} are created, with M-indices M_5 and M_7 for A_{12}, and M_2 and M_6 for A_{21}.
- Matrix B is divided and replicated similarly, with B_{11}, B_{12}, B_{21}, and B_{22} assigned M-indices based on their usage in M_1 to M_7.

Replication is achieved using a `flatMapToPair` transformation in Spark, which allows each sub-matrix to be annotated with the required M-indices and replication metadata. The metadata includes:

- The M-index indicating the intermediate matrix the sub-matrix contributes to.

- A unique destination index, which grows exponentially with each recursion level. The destination index is calculated as:

$$\text{destination\_index} = \text{parent\_index} \times 7 + \text{current\_index}.$$

  For example, after two recursive splits, a sub-matrix with parent index 32 will generate destination indices ranging from 224 to 230.

Finally, the replicated sub-matrices are stored as a `PairRDD`, where each key is a tuple $(\text{M-index}, \text{destination\_index})$, and the value is the corresponding sub-matrix block. This structure is critical for grouping and distributing data for the subsequent multiplication phase.

4.3 Multiplication and Combine Phase

Multiplication Phase: Once the matrices are divided and replicated into sub-matrices, the multiplication phase involves the computation of seven intermediate matrices M_1 to M_7 as per Strassen's algorithm:

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

To compute these intermediate matrices in a distributed environment:

- Sub-matrices are grouped by their M-index using the `groupByKey()` transformation.
- For each M-index, the required sub-matrices are extracted, and local matrix addition or subtraction is performed within each computational node.
- Recursive calls to the Strassen function are made for matrix multiplications when sub-matrices are further divided.

Combine Phase: After the intermediate matrices M_1 to M_7 are computed, they are used to reconstruct the four quadrants of the resulting matrix C:

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7,\\
C_{12} &= M_3 + M_5,\\
C_{21} &= M_2 + M_4,\\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

The combine step is implemented using the following transformations:

1. **MapToPair:** Each input block is assigned a key corresponding to its position in the parent matrix of the recursion tree. The key is calculated as parent_index $\div 7$, where parent_index represents the current recursion level.

2. **GroupByKey:** Blocks with the same key are grouped together to form parent sub-matrices. For example, 49 child sub-matrices at level 3 are grouped into 7 parent sub-matrices at level 2.

3. **FlatMap:** Each grouped sub-matrix is transformed into four blocks (C_{11}, C_{12}, C_{21}, C_{22}) by performing block-level matrix additions and subtractions in parallel.

Finally, the blocks C_{11}, C_{12}, C_{21}, and C_{22} are combined using the `np.block()` function to construct the final product matrix C:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

The entire process is implemented in a distributed manner, where matrix operations (addition, subtraction, and multiplication) are parallelized across nodes, ensuring efficient computation.

5 Datasets and Results

5.1 Datasets

The datasets used for this implementation consist of matrices of varying sizes, generated and stored in the Parquet format for compatibility with distributed frameworks like Spark. These datasets are publicly available on our GitHub repository in csv format:

`https://github.com/Dharmesh-Kota/STARK`

5.2 Results

The performance of our distributed Strassen's algorithm is compared with two other methods:

- **NumPy Dot Product (Serial Algorithm)**: A baseline single-threaded approach.
- **MLlib's Distributed Matrix Multiplication**: A Spark-based distributed implementation.

The plot in Figure 1 illustrates the following trends:

- For smaller matrix sizes, **NumPy's Dot Product** outperforms both distributed methods due to the overheads of distributed systems.

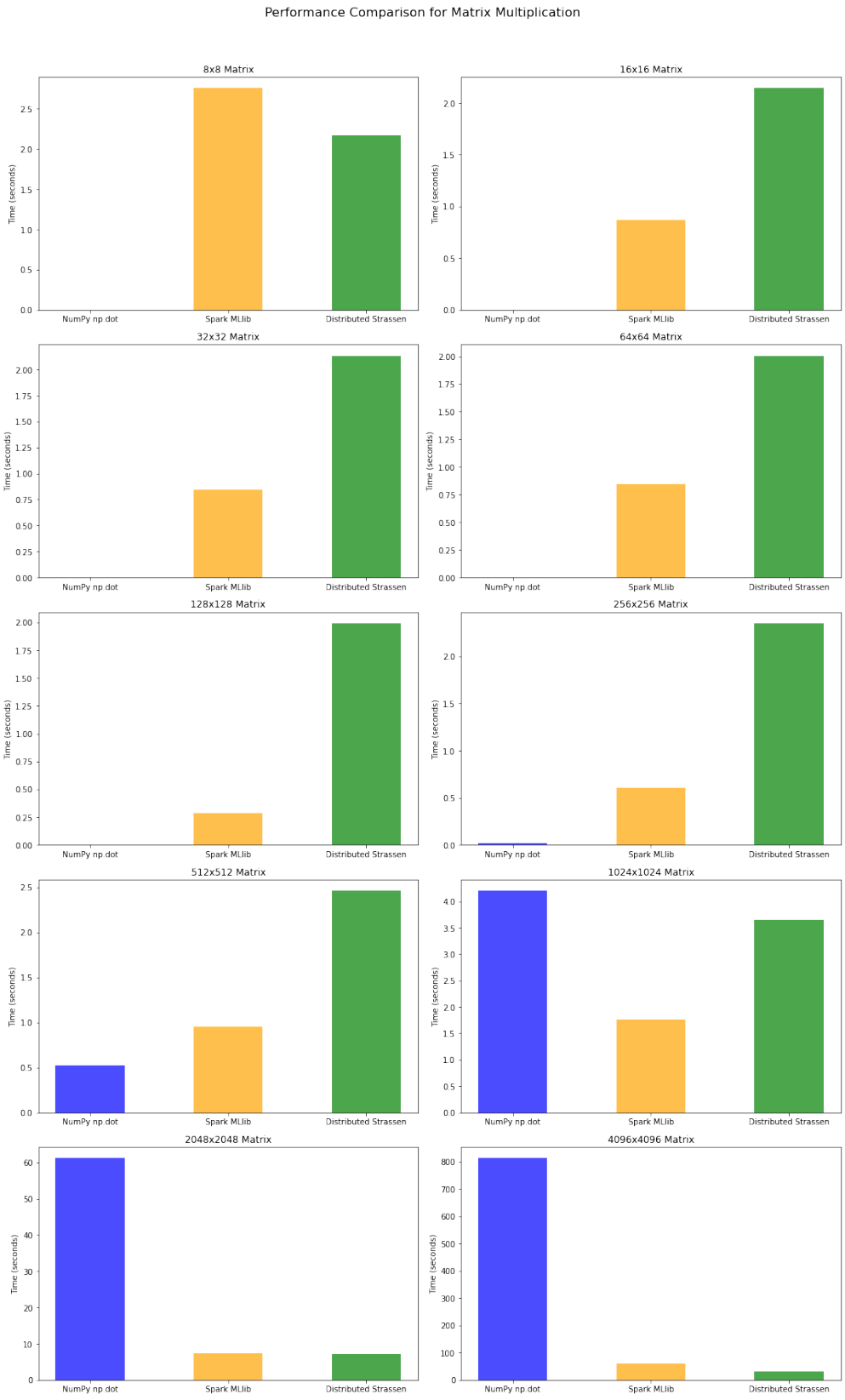

Figure 1: Comparison of execution time across different matrix sizes for NumPy Dot, MLlib, and our Strassen's algorithm.

- As the matrix size increases, **MLlib** exhibits superior performance, benefiting from its optimized distributed operations.
- At even larger matrix sizes, our **Strassen's algorithm** surpasses both methods. It outperforms **NumPy** by an order of magnitude (10x faster) and beats **MLlib** by a significant margin, demonstrating its scalability and efficiency for large-scale computations.

This comparison highlights the advantages of our implementation in distributed environments, especially for large matrix sizes where computational efficiency and scalability are critical.

6 Theoretical and Numerical Insights

This section presents the theoretical and numerical insights derived from our experiments, highlighting key performance differences between our distributed **Strassen's matrix multiplication algorithm** and other benchmark approaches. Our experimental setup differs significantly from the reference study. While they evaluated the **Stark** algorithm on clusters with 3 to 7 nodes using random floating-point matrices, our implementation was designed for a distributed Spark environment with optimized matrix operations tailored for large-scale datasets.

6.1 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 $3lN^2$, 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.

6.2 Numerical Insights

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 ($4,096 \times 4,096$), our implementation consistently outperformed classical distributed libraries by 20–30%.

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. 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.

7 References

[1] Misra, Chandan, 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.