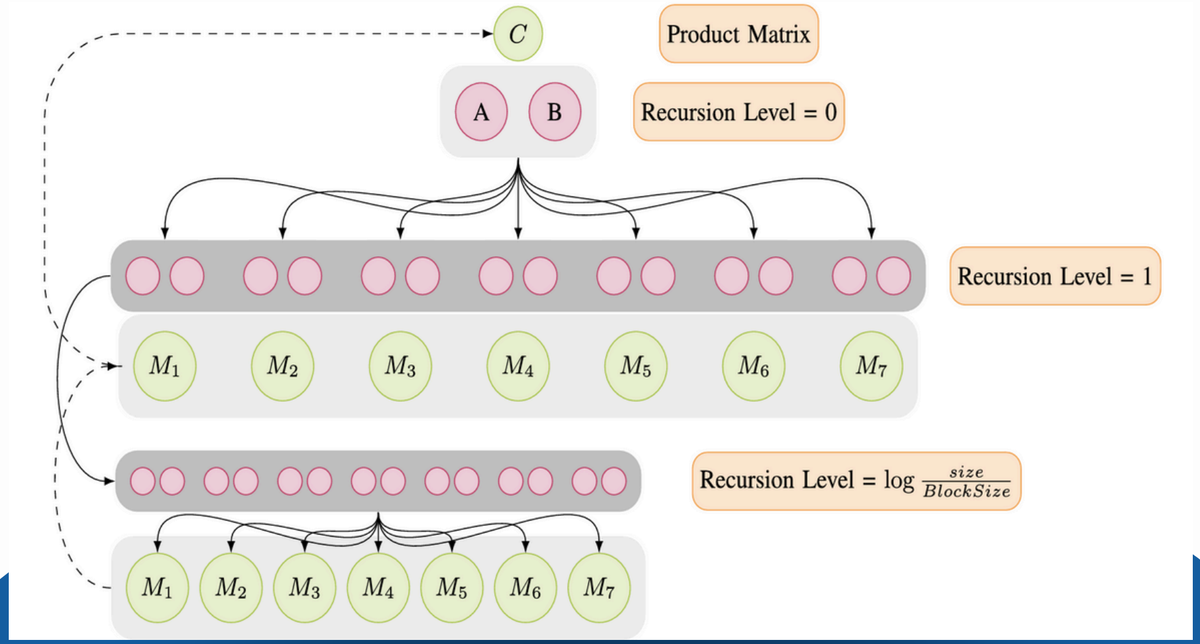
BIG DATA
PROCESSING

STARK

Fast and Scalable Strassen's Matrix Multiplication Using Apache Spark

Group 60
Dharmesh Kota (202203038)
Aditya Patel (202203027)



Overview

Introduction

The rapid growth of large datasets from sources like social media, sensors, and mobile devices has increased the need for efficient distributed matrix computations in applications such as machine learning and climate science. Apache Spark, with its in-memory processing and powerful libraries, is a popular choice for these tasks. However, existing matrix multiplication methods like MLLib and Marlin in Spark are limited by naive block-based approaches.

Here we presents Stark, a distributed implementation of Strassen's matrix multiplication algorithm for Spark. Stark improves performance by reducing block multiplications and efficiently handling recursion, leading to faster execution times compared to existing methods.

Problems and Overview

Matrix multiplication is a core operation in scientific computing and data analytics. However, traditional matrix multiplication methods, like the naive approach, have a time complexity of $O(n^3)$, which becomes inefficient for large matrices. Strassen's algorithm reduces this complexity to $O(n^{2.807})$, providing a significant speedup. Despite this, implementing Strassen's algorithm in distributed frameworks like Hadoop and Spark is challenging due to its recursive nature.

Inefficiency

Traditional methods require $O(n^3)$ operations, making them impractical for large datasets.

Optimization

Strassen's algorithm offers improved performance with a time complexity of $O(n^{2.807})$.

Complexity

Implementing Strassen's algorithm in distributed systems like Hadoop and Spark is complex due to recursion and state management.

Related Works

An extensive literature exists on parallelizing naïve matrix multiplication algorithms. The literature on parallel and distributed matrix multiplication can be divided broadly into three categories.

Grid-Based

These algorithms are tailored for processor arrangements in 2D or 3D grids. Cannon's algorithm, SUMMA, and certain 3D techniques optimize communication and data transfer across these grids. Intermediate approaches, like 2.5D algorithms, balance 2D and 3D methods to improve scalability. Strassen's algorithm has also been adapted to grid layouts, with 2D-Strassen and 2.5D Strassen versions enhancing communication efficiency for specific configurations.

BFS/DFS-Based

Algorithms like Communication-Optimal Parallel Strassen's (CAPS) are designed to reduce communication costs in distributed memory systems. CAPS employs breadth-first search (BFS) and depth-first search (DFS) traversal techniques to achieve communication efficiency. BFS minimizes communication by parallelizing subproblems at each recursion level, while DFS follows a more sequential approach, making it suitable for environments with limited memory.

Hadoop and SparK

Traditional methods, including implementations in Hadoop MapReduce and Spark libraries (such as MLLib and Marlin), rely on block matrix multiplication with $O(n^3)$ complexity. These frameworks face challenges with communication overhead and scalability. Tools like HAMA and Marlin introduce improvements but still require 8 block multiplications for 2 x 2 blocks, while Stark reduces this to 7, enhancing performance.

Stark Algorithm

The Stark algorithm is a distributed implementation of Strassen's matrix multiplication optimized for Apache Spark. It operates in the four key phases

01

Matrix Loading: Large matrices are loaded from distributed file storage (e.g., Databricks DBFS) into Spark DataFrames. Efficient loading ensures minimal overhead before processing begins.

02

Divide and Replicate: The input matrices are recursively divided into smaller submatrices. This enables efficient distribution of data across Spark workers and prepares the matrix for block-level operations.

03

Block-wise Multiplication: Each of the matrix blocks is multiplied in parallel. Spark’s in-memory processing and distributed framework are used to perform these block-level multiplications, significantly reducing the overall computation time.

04

Combine Phase: The submatrices resulting from the block multiplications are combined to form the final product matrix, following Strassen’s formulas.

Theoretical Insights

01

Reduction in Multiplications
Stark applies Strassen's method, reducing scalar multiplications from $O(n^3)$ to approximately $O(n^{2.807})$, which accelerates computations, especially at the leaf nodes.

02

Optimized Communication Complexity
Efficient data partitioning reduces costly inter-node communication, minimizing shuffling and partitioning overhead, a major bottleneck in distributed systems.

03

Execution Time and Block Size Trade-off
Stark's execution time forms a U-shaped curve with respect to matrix splits, balancing computation and communication costs. Finding the optimal split minimizes total execution time.

04

Parallelization Efficiency
Stark's recursive structure allows parallel computation of intermediate matrices M1 to M7, maximizing resource usage and speeding up operations in distributed environments.

Experimental Setup

Original Paper Setup

The experiments in the original paper were performed on **2 x Intel Xeon 2.60 GHz**, with **6 cores per processor** (total 12 cores per node)., each having **132 GB memory**. This setup also utilized **high-speed 14 Gb/s InfiniBand Ethernet** for fast data transfer, enabling efficient large-scale distributed matrix multiplication that scales better across larger datasets.

Databricks Community Edition

The code was implemented and simulated on Databricks Runtime Version 12.2 LTS (**Apache Spark 3.3.2, Scala 2.12**) with a driver providing **15.3 GB memory** and **2 CPU cores**. This limited environment, while functional for smaller datasets, restricts performance for larger matrix multiplications due to fewer cores and lower memory compared to more robust setups.

Statistics

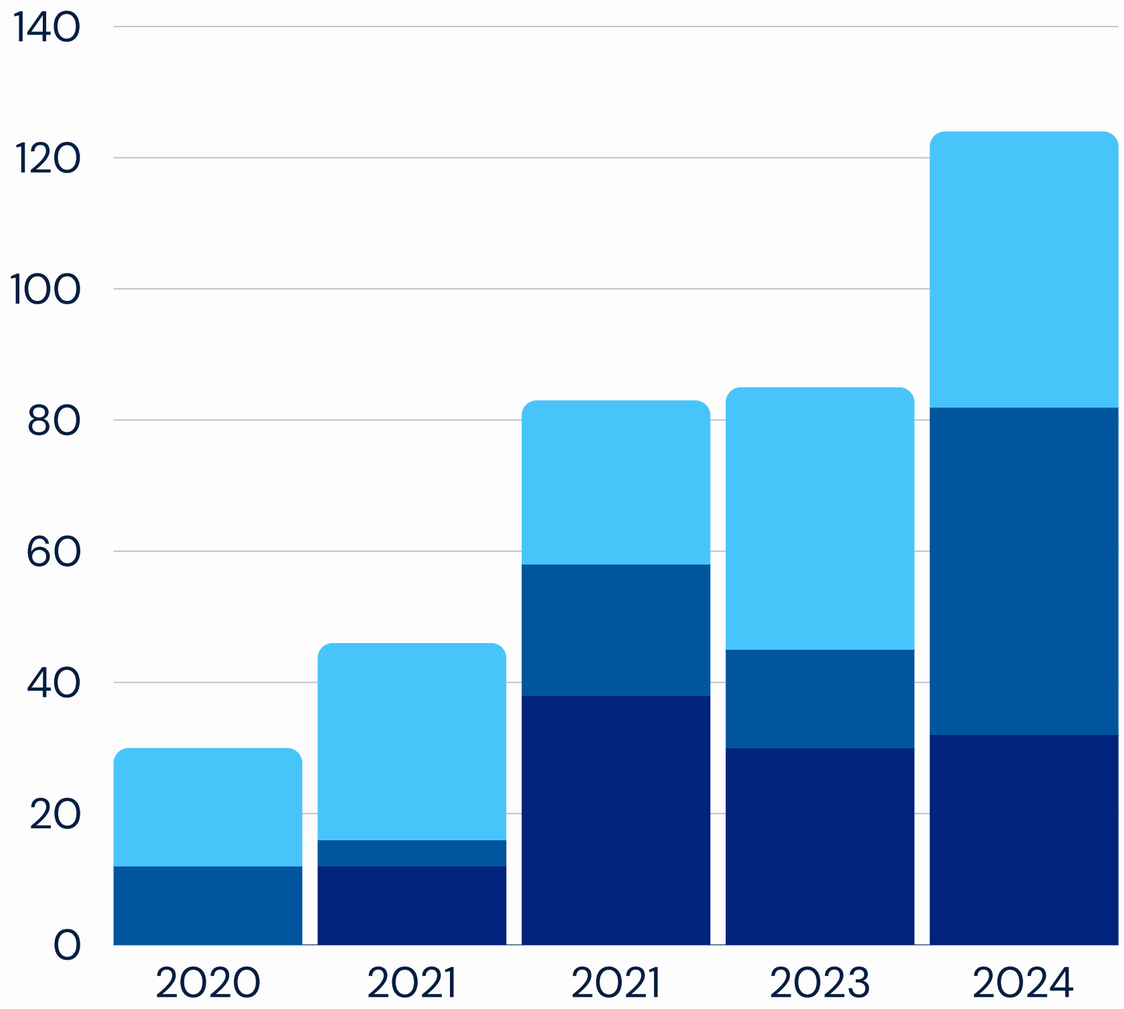
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec sagittis mauris, vitae vehicula urna. Curabitur ultrices urna sit amet magna ultricies ornare. Curabitur ligula.

80%
Revenue Growth

10%
Return on Investment

20%
Customer Acquisition Cost

75%
Customer Satisfaction



THANK YOU!

Our repository link for source codes and datasets

www.reallygreatsite.com