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

CHANDAN MISRA, Indian Institute of Technology Kharagpur
SOURANGSHU BHATTACHARYA, Indian Institute of Technology Kharagpur
SOUMYA K. GHOSH, Indian Institute of Technology Kharagpur

This paper presents a new fast, highly scalable distributed matrix multiplication algorithm on Apache Spark, called Stark, based on Strassen's matrix multiplication algorithm. Stark preserves Strassen's 7 multiplication scheme in distributed environment and thus achieves faster execution. It is based on two new ideas; it creates a recursion tree of computation where each level of such tree corresponds to division and combination of distributed matrix blocks in the form of *Resilient Distributed Datasets* (RDDs); It processes each divide and combine step in parallel and memorize the sub-matrices by intelligently tagging matrix blocks in it. To the best of our knowledge, Stark is the first Strassen's implementation in Spark platform. We show experimentally that Stark has a strong scalability with increasing matrix size enabling us to multiply two 16384×16384 matrices with 28% and 36% less execution time than *Marlin* and *MLLib* respectively, state-of-the-art matrix multiplication approaches based on Spark.

CCS Concepts: • **Computing methodologies** → **MapReduce algorithms**;

Additional Key Words and Phrases: Linear Algebra, Matrix Multiplication, Strassen's Algorithm, Spark

ACM Reference format:
Chandan Misra, Sourangshu Bhattacharya, and Soumya K. Ghosh. 2010. Stark: Fast and Scalable Strassen's Matrix Multiplication using Apache Spark. *ACM Trans. Web* 9, 4, Article 39 (March 2010), 29 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Strassen's algorithm for matrix multiplication (MM) was proposed by Strassen's in 1968 [36]. The algorithm uses $O(n^{2.807})$ arithmetic operations, a much faster than traditional naive approach which requires $O(n^3)$ arithmetic operations. The algorithm is based on recursive block MMs and a trick to multiply 2×2 matrices in 7 multiplications. Each initial $2^n \times 2^n$ matrices are broken down into four $2^{n-1} \times 2^{n-1}$ sub-matrices and each of the sub-matrix

This work is supported by the National Science Foundation, under grant CNS-0435060, grant CCR-0325197 and grant EN-CS-0329609.
Author's addresses: C. Misra, Advanced Technology Development Center, Indian Institute of Technology Kharagpur; S. Bhattacharya and S. K. Ghosh, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur.

multiplication are performed using 7 sub-matrix multiplications and each of the sub-matrices are recursively computed in the same way.

Recent growth in massive dataset from different sources like social media, weather sensors, mobile devices etc. has made the path for research in domain like machine learning, climate science and social media analytics. These require large scale data processing with minimal effort and a system which scales as data grows with any failure. As these applications need matrix computation on massive dataset, there is a immediate requirement of large scale matrix computation which can not be done on a single machine. Large scale distributed data processing framework like Hadoop MapReduce [2] and Spark [3] have emerged as next generation parallel data-flow programming platform for data intensive complex analytics and building parallel applications in fields like machine learning, climate science and social media analytics. These reliable shared storage and analysis systems put the systems like RDBMS, grid computing and volunteer computing behind by its powerful batch processing, scalability and fault tolerance capabilities. Spark has gained its popularity for its in-memory data processing ability to run programs faster (up-to 100 times for in-memory analytics) than Hadoop MapReduce. Its general purpose engine supports a wide range of applications including batch, interactive, iterative algorithm and streaming and it offers simple APIs and rich built-in libraries like MLLib [27], GraphX [40] for data science tasks and data processing applications. Therefore, we can get substantial gain by implementing computing intensive algorithms which consume large data set as input. In the present work, we concentrate on Spark and describe a distributed implementation of Strassen's MM algorithm using it.

A substantial research work has been done in implementing distributed matrix multiplication in Hadoop MapReduce and Spark. One of the early work was *HAMA*, which implemented MM on MapReduce. However, it suffers from the shortcomings of Hadoop, communicating HDFS for each map or reduce task. To overcome this drawback, spark based MM implementation has come into existence. The first approach is the MM subroutine of machine learning library of *Spark*, which is called *MLLib*. The second one is MM system, which intelligently selects one of their three MM algorithms according to the size of the input matrices. However, both these schemes applied classical matrix multiplication approach which requires 8 multiplications. In the present work, we have tried to overcome this shortcoming by applying Strassen's MM algorithm on spark and preserving the 7 multiplication scheme in a distributed environment.

There are several benefits to implement Strassen's algorithm on Spark. First of all, Strassen's algorithm is inherently faster with a time complexity of $O(n^{2.807})$ compared to naive implementation having a time complexity of $O(n^3)$ when multiplying two $n \times n$ matrices. Implementing it on a fast processing engine results fast execution of the algorithm. Secondly, there have been several implementations of MM in Hadoop MapReduce apart from the naive techniques. These conventional approaches [18, 31], rely solely on MapReduce programming paradigm and block matrix data structure. However, Strassen's algorithm can not be implemented efficiently in Hadoop MapReduce for its inherent recursive nature. Therefore, Spark is a natural choice in this scenario making recursive calls to methods having Spark jobs defined in it. Thirdly, being implemented in Spark, our technique becomes a part of the overall Hadoop ecosystem, thereby supported by HDFS, Cassandra, HBase, Hive etc. Last but not the least, our technique can be used as a plug-in for certain large data analytics workflow, where the input matrices are generated by some other Spark or MapReduce jobs and the product matrix from our technique can be consumed by some other jobs in the workflow.

There are several challenges to implement the parallel version of Strassen's MM algorithm. They are — 1) The matrix is not easily partitionable i.e. each element in the product matrix depends on multiple elements in the input matrices. Therefore, each partition can not be processed independently which is a one of the requirements for MapReduce programming model. 2) Strassen's algorithm is iterative and thus not suitable for Hadoop framework. 3) It is very difficult to implement divide and conquer algorithms in MapReduce like framework where components of each partition depends on other partitions. In each Strassen's recursive call the input matrices are divided into 7 sub-matrices and each such sub-matrix depends on the elements of components of other partitions. Moreover, it is necessary to keep track of the sub-matrices and matrix blocks in the intermediate map group and reduce phases, so that it can be further divided or merged and can thus get the final position in the product matrix. The contributions of this paper is to overcome the above challenges with a distributed tail recursion which is created by intelligently labeling the sub-matrices and matrix blocks for each recursive call. The tags are chosen in a way such that the division of the matrices can be done in parallel in a top-down fashion and also product sub-matrices can be arranged from the divisions in parallel as well in a bottom-up approach.

1.0.1 Organization of the article. After presenting a detailed related work in section 2, we introduce the Strassen's multiplication algorithm on a single node in section 3.1. We introduce our algorithm *Stark* in section 3 and provides a detailed description of the algorithm along with the data structure used. In section 4, we evaluate the cost analysis of our algorithm along with two other competing approaches — *MLLib* and *Marlin* in order to show that *Stark* has a better performance over others. This will also guide us to explain our experimental results provides in section 5. Section 6 summarizes the results and discusses the future research direction.

2 RELATED WORK

An extensive researches can be found in the literatures for parallelizing MM in [7], [6], [9], [1], [38], [17], [8], [15], [26], [19], [31], [5], [33], [12] and [34]. Similarly Strassen's MM has also been studied for parallelization in [21], [14], [16], [24], [26], [37], [13], [29], [35], [23] and [4].

As pointed out in [17], [12] and [23], there are three types of parallel MM approaches and it also applies to Strassen's MM algorithm. These three types of parallel MM algorithm schemes are 1) Grid based approach, 2) BFS/DFS based approach and 3) Hadoop and Spark based approach.

2.1 Grid Based Approach

The grid based approaches are particularly well suited for the processor layout in a two or three dimensional grid. In this layout, the communication occurs either in the same row or in the same column. Based on this processor layout, these approaches are again classified as 2D, 2.5D and 3D. 2D and 3D cases treat processors as two and three dimensional grid respectively. The most common known 2D algorithms are [7] and [38]. In 3D approaches like [1] and [6], the communication cost is minimized using extra memory than 2D algorithms. It also reduces the bandwidth cost compared to 2D algorithms. 2.5D multiplication approach in [32], [33] and [25], has been developed to interpolate between 2D and 3D approaches. It has a better bandwidth than 2D.

Strassen's MM has also gone through similar evolution and got new 2D and 2.5D approaches. Luo and Drake [24] presented a scalable and parallel Strassen's MM algorithm. They have provided two approaches to multiply two matrices in parallel. The first approach, is to use classical parallel matrix multiply for the parallelization and Strassen's multiply method locally — called the 2D-Strassen. In the second approach, they reversed the order i.e. they parallelize Strassen's at the higher levels and use standard parallel MM at lower levels — called the Strassen-2D. They analyzed the communication costs for these two approaches. Grayson et. al. [16] improved on the second approach and concluded that the second one is the best approach under their new circumstances. Then comes the 2.5D version of the 2D-Strassen and Strassen-2D algorithms. In [32], they got better communication efficiency than its 2D counterparts, but still lacks communication optimality. Grid based algorithms are very efficient in grid and torus based topologies but may not perform well in other more general topologies [12].

2.2 BFS/DFS Based Approach

The failure of the above mentioned approaches to achieve communication optimality, BFS/DFS approach is developed [4] for Strassen's algorithm. BFS/DFS approach treats processor layout as hierarchy rather than two or three dimensional grid and based on sequential recursive algorithm. Among Strassen based other parallel algorithms, (Communication Optimal Parallel Strassen's) *CAPS* [23] provides the minimized communication costs and runs faster in practice. Ballard et al. presented the communication costs for Strassen in [5] and [4] and also provides the communication lower bound for square as well as for rectangular matrices in [12]. *CAPS* matches the lower bound and provides communication optimality.

CAPS traverses the Strassen recursion tree in parallel in two ways. In the *unlimited memory* (UM) scheme, it takes k BFS steps and then perform local matrix multiplication. The *Limited Memory* approach takes l DFS steps and then k BFS steps. The memory footprint can be minimized by minimizing l. They also showed that, second approach can be tuned to get more complicated interleaving approach but does not attain optimality more than a constant factor. Though, our implementation follows a similar kind of recursion tree as *CAPS*, it is worth evaluating the algorithm in a scalable framework where data is distributed.

2.3 Hadoop and Spark Based Approach

There are several implementation of distributed MM on using Hadoop MapReduce and Spark. John Norstad in [28] presented four strategies to implement data parallel MM using block matrix data structure. However, all of them are based on classical parallel approach which requires eight multiplications.

There are other distributed framework that provides massive matrix computation like *HAMA* [31] and *MadLINQ* [30]. MM in *HAMA* is carried out using two approaches — *iterative* and *Block*. In iterative approach, each map task receives a row index of right matrix as a key and the column vector of the row as a value. Then it multiplies all columns of i^{th} row of left matrix with the received column vector. Finally, a reduce task collects the i^{th} product into the result matrix. *Block* approach reduces required data movement over the network by building a collection table and placing candidate block matrix in each row. However, iterative approach is not suitable in Hadoop for massive communication cost. Though *Block* approach incurs low communication cost, does not provide faster execution as it uses classical parallel MM approach. *MadLINQ*, built on top of Microsoft's *LINQ* framework and *Dryad* [20], is an example of cloud based linear algebra platform. However, it suffers from same kind of drawback as *HAMA*.

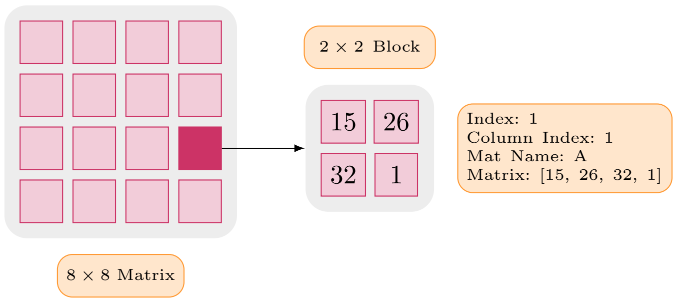


Fig. 1. Block Data Structure

Rong Gu et al. in [17] developed an efficient distributed computation library, called Marlin, on top of Apache Spark. They proposed three different MM algorithm based on the size of input matrices. They have shown that, Marlin is faster than R and distributed algorithms based on MapReduce. When the matrix sizes are square, they have used a hybrid of the naive block MM scheme. Though they have minimized the shuffle in join step, underlying they incur 8 multiplications compared to 7 multiplications on Stark, which makes Stark faster than Marlin. MLLib block MM does the same thing, but a little bit different way. The algorithm first lists all the partition for each block that are needed in the same place and then shuffles, which reduce the communication cost.

3 DISTRIBUTED STRASSEN'S ON SPARK

In this section, we discuss the implementation of *Stark* on Spark framework. Before doing so, we present the serial algorithm of the same which runs on a single computer, for completeness and easy understanding of the paper.

3.1 Naive Strassen's Preliminaries

Strassen's MM can multiply two 2×2 matrix using 7 multiplications and 18 additions, providing much faster execution compared to 8 multiplications and 4 additions of the classical algorithm. Algorithm 1 shows the serial single node version of the algorithm.

3.2 Block Data Structure

The main data structure used is a matrix, which is represented as an RDD of *blocks*. Blocks store information necessary for (1) representing the matrix i.e. storing all the entries and (2) book keeping information needed for running the algorithm.

Conceptually, each matrix of dimension n, is partitioned into b partitions, giving $\frac{n}{b}$ *block rows* and $\frac{n}{b}$ *block columns*. Here, *Block Rows* and *Block Columns* are defined by the number of rows and columns where each row and column corresponds to one block. Each matrix of size n is divided into four equal square sub-matrices of dimension $\frac{n}{2}$, until it reaches *block* dimension of $\frac{n}{b}$. These sub-matrices are stored in data structure called *blocks*, which is central to our algorithm. Note that, these *blocks* are of fixed size, and can be stored and multiplied on a single node. Each block contains four fields (depicted in Fig. 1):

(1) row-index: Stores current row index of the sub-matrix, in multiples of n. Note that as the larger matrix is split during execution of algorithm, these indices can change to keep track of the current position of sub-matrix.
(2) column-index: Similar to above, stores the current column index of the sub-matrix.

ALGORITHM 1: Strassen's Matrix Multiplication

```
Procedure Strassen's(A, B, threshold)
    A = input matrix of size $m \times p$;
    B = input matrix of size $p \times n$;
    C = input matrix of size $m \times n$;
    if $n$=thresold then
        Multiply A and B using naive approach;
    else
        Compute $A_{11}, B_{11}, ..., A_{22}, B_{22}$ by computing $m = \frac{n}{2}$;
        $M_1$ = STRASSEN'S($(A_{11} + A_{22})$,$(B_{11} + B_{22})$);
        $M_2$ = STRASSEN'S($(A_{21} + A_{22})$,$B_{11}$);
        $M_3$ = STRASSEN'S($A_{11}$,$(B_{12} - B_{22})$);
        $M_4$ = STRASSEN'S($A_{22}$,$(B_{21} - B_{11})$);
        $M_5$ = STRASSEN'S($(A_{11} + A_{12})$,$B_{22}$);
        $M_6$ = STRASSEN'S($(A_{21} - A_{11})$,$(B_{11} + B_{12})$);
        $M_7$ = STRASSEN'S($(A_{12} - A_{22})$,$(B_{21} + B_{22})$);
        $C_{11} = (M_1 + M_4 - M_5 + M_7)$;
        $C_{12} = (M_3 + M_5)$;
        $C_{21} = (M_2 + M_4)$;
        $C_{22} = (M_1 - M_2 - M_3 + M_6)$;
    end
    return C;
```

(3) mat-name: Stores a tag which is used as a key for grouping the blocks at each stage of the algorithm, so that blocks which need to be operated on are in the same group. It consists of a comma separated string which denotes two components:
   (a) The matrix tag: stores the matrix label, for example, A or B or one of the eight sub-matrices A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21} and B_{22} or M.
   (b) M-Index: Each sub-matrix is broken down into 7 sub-matrices. Therefore, this index helps to signify one of these 7 sub-matrices.
(4) matrix: 2D array storing the matrix.

3.3 Implementation Details

The core multiplication algorithm (described in Algorithm 2) takes two matrices (say A and B) represented as *RDD* of blocks, as input as shown in Fig. 2. The computation performed by the algorithm can be divided into 3 phases:

- Recursively splitting each input matrix into 4 equal sub-matrices and replicate the sub-matrices so as to facilitate the computation of intermediate matrices (M_1 to M_7).
- Multiply blocks serially to form blocks M_1 to M_7.
- Combine the sub-matrices to form matrices C_1 to C_4 of size 2^n from 2^{n-1}.

Each step mentioned above run in parallel inside the cluster. Each step is described in the following sections.

3.3.1 Divide and Replication Phase. In the divide step, the matrices are divided into 4 sub-matrices of equal size and the blocks constitutes each sub-matrix contains same $M - Index$ according to the location of the sub-matrix in its parent. The procedure is shown

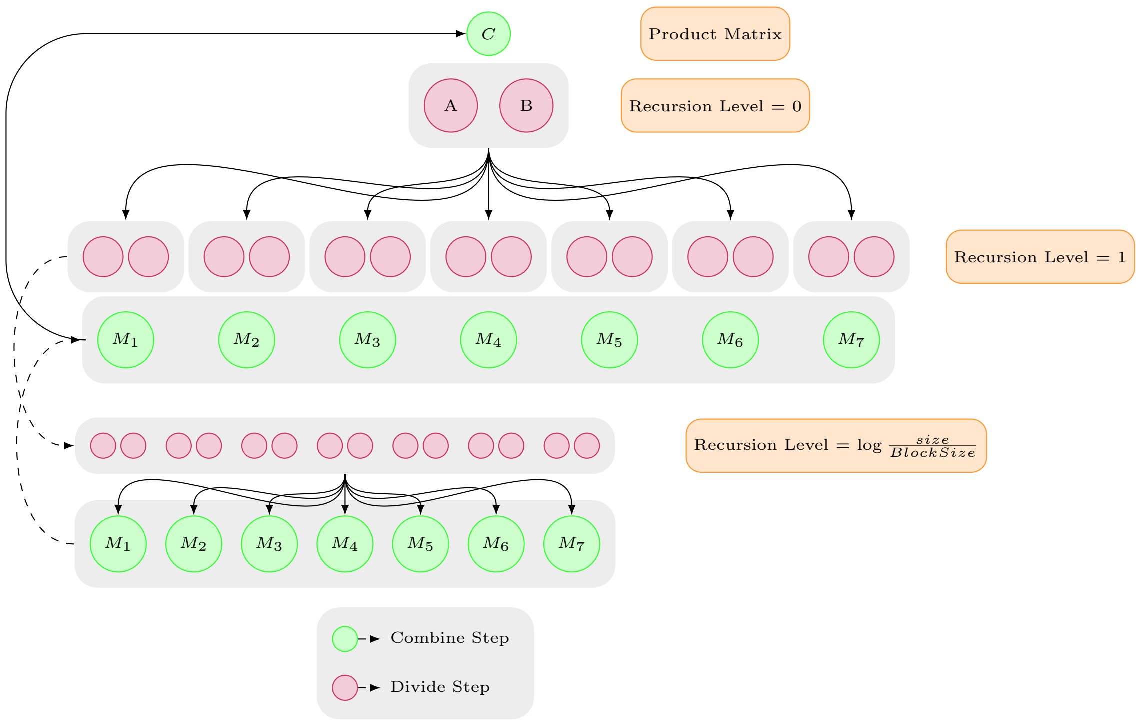


Fig. 2. The implementation flow of Stark

in Fig. 3. To create seven sub-matrices M_1 to M_7, we create 12 sub-matrices of size $2^{\frac{n}{2}}$ as shown in Algorithm 1. We create 4 copies of A_{11} and A_{22} and 2 copies of A_{12} and A_{21} using *flatMapToPair* transformation. Matrix B is divided similarly. *flatMapToPair* takes a single *key-value* pair and generates a list of *key-value* pairs having the same single key. In this case, it takes a single block of any of the two input matrices and returns a list of blocks according to the group it is about to be consumed i.e. M_1 to M_7 and the M-index of the recursive tree. The *mat-name* property of the block preserves predecessor sub-matrix name i.e. A_{11} to B_{22}.

Next, the blocks of similar key (M_1 to M_7) are grouped and thus contains the sub-matrices that form M1 to M7. For adding and subtracting blocks of sub-matrices we use *flapMap* transformation. It takes a *PairRDD* and returns a *RDD*. Here, it takes a list of blocks of 4 sub-matrices and returns a list of blocks of 2 sub-matrices. These 2 sub-matrices are generated using adding or subtracting the corresponding blocks of four sub-matrices as shown in Fig. 4.

Then we divide the intermediate blocks of the sub-matrices again by recursively calling the Strassen's method. Each time we go down the leaf of the execution tree, we divide the sub-matrices into smaller sub-matrices of size $\frac{n}{2}$. This process continues until it reaches the size of a block.

3.3.2 Multiplication of Block index Matrices. When the division reaches the size of user defined block size, the blocks are multiplied using serial MM algorithm. This is done using one *mapToPair*, followed by one *groupByKey* and one map function as shown in Algorithm 4. The *mapToPair* function takes each block and returns a *key-value* pair to group two

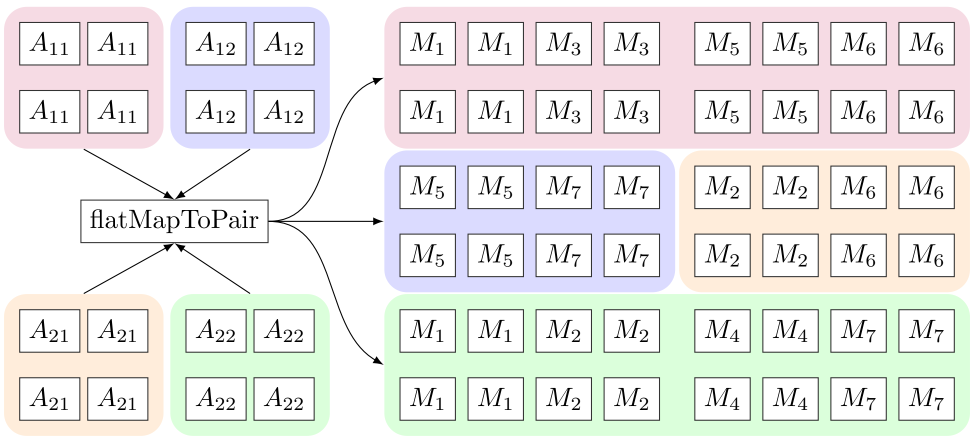

A_{11}
A_{12}
A_{21}
A_{22}
flatMapToPair
M_1 M_1 M_3 M_3 M_5 M_5 M_6 M_6
M_5 M_5 M_7 M_7 M_2 M_2 M_6 M_6
M_1 M_1 M_2 M_2 M_4 M_4 M_7 M_7

Fig. 3. Divide and Replicating Sub-Matrices

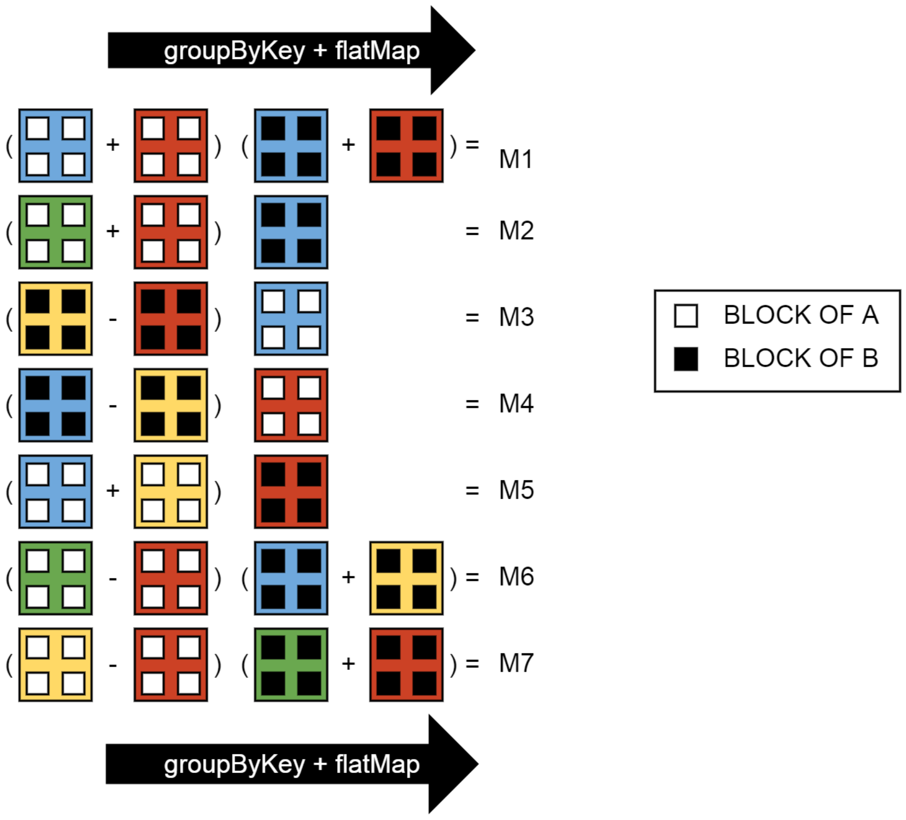


Fig. 4. Addition and Subtraction of sub-matrices

ALGORITHM 2: Distributed Strassen's

```
Procedure DistStrass($RDD < Block > A$, $RDD < Block > B$, $int\ n$)
    Result: RDD of blocks of the product matrix C
    $size$ = Size of matrix $A$ or $B$;
    $blockSize$ = Size of a single matrix block;
    $n = \frac{size}{blockSize}$;
    if $n=1$ then
        /* Boundery Condition: RDD A and B contain blocks with a pair of blocks
           (candidates for multiplication) having same matname property          */
        MulBlockMat($A$,$B$);
    else
        $n = \frac{n}{2}$;
        /* Divide Matrix A and B into 4 sub-matrices each ($A_{11}$ to $B_{22}$). Replicate
           and add or subtract the sub-matrices so that they can form 7 sub-matrices
           ($M_1$ to $M_7$)                                                       */
        D = DivNRep($A$,$B$);
        /* Recursively call DistStrass() to multiply two sub-matrices of block size
           $\frac{n}{2}$                                                          */
        R = DistStrass($A$,$B$,$n$);
        /* Combine seven submatrices (single RDD of blocks ($M_1$ to $M_7$)) of size $\frac{n}{2}$
           into single matrix (RDD of blocks ($C$))                               */
        C = Combine($R$);
    end
    return C;
```

ALGORITHM 3: Divide and Replication

```
Procedure DivNRep($RDD < Block > A$, $RDD < Block > B$)
    Result: $RDD < Block > C$
    /* Make union of two input RDDs. Each block of the resulting RDD having a tag
       with string similar to $A|B$, $M_{1|..|7}$, M-index. In the first recursive call the
       tag is $A|B, M, 0$                                                         */
    $RDD < Block > AunionB = A$.union($B$);
    /* Map each block to multiple (key, Block) pairs according to the block index.
       For example, $A_{11}$ is replicated four times. Each key contains string $M_{1|2...|7}$,
       M-index. Each block contains a tag with string $A_{11|12|21|22}$ or $B_{11|12|21|22}$.   */
    $PairRDD < string, Block > firstMap = AunionB$.flatMapToPair();
    /* Group the blocks according to the key. For each key this will group blocks
       with tags that eventually form $M_1$ to $M_7$.                              */
    $PairRDD < string, iterable < Block >> group = firstMap$.groupByKey();
    /* Add or subtract blocks with the tag start with similar character ($A|B$) to get
       the two blocks of RDDs for the next divide phase.                          */
    $RDD < Block > C = group$.mapToPair();
    return C;
```

potential blocks for multiplication. The *groupByKey* groups two blocks and map function returns the multiplication of the two blocks. The keys in the *mapToPair* function are chosen in such a way so that all the leaf index blocks are multiplied in parallel. We transform the leaf node block matrices to Breeze matrices to make the multiplication faster on a single node.

ALGORITHM 4: Block Matrix Multiplication

```
Procedure MulBlockMat($RDD < Block > A,\ RDD < Block > B$)
    /* Result contains RDD of blocks. Each block is the product of two matrix blocks
       residing in the same computer.                                              */
    Result: $RDD < Block > C$
    /* Make union of two input RDDs. Each block of the resulting RDD having a tag
       with string similar to $A|B, M_{1|2\ldots|7}, index$.                       */
    $RDD < Block > AunionB = A$.union($B$);
    /* Map each block to a (key,Block) pair. The key contains string $M_{1|2\ldots|7}, index$.
       Each Block contains a tag with string $A|B$.                                */
    $PairRDD < string, Block > firstMap = AunionB$.mapToPair();
    /* Group the blocks according to the key. For each key, this will group two
       blocks, one with block tag $A$ and another with $B$.                        */
    $PairRDD < string, iterable < Block >> group = firstMap$.groupByKey();
    /* Multiply two block matrix inside a single computer serially and return each
       Block to the resulting RDD.                                                 */
    $RDD < Block > C = group$.map();
    return C;
```

3.3.3 Combining the Sub-matrices. In this step, the product matrix blocks of seven sub-matrices M_1 to M_7 are rearranged to produce a larger matrix. Combination phase occurs when the recursive call to Strassen's procedure returns. In each such return the size of the matrices becomes $2^{\frac{n}{2}}$ to 2^n. Each such combine step is done in parallel. The combine step is shown in Algorithm 5.

ALGORITHM 5: Combine Phase

```
Procedure Combine($RDD < Block > BlockRDD$)
    Result: $RDD < Block > C$
    /* Map each block to (key,Block) pair. Both the key and block mat-name contains
       string $M_{1|2\ldots|7}, index$. indexes are divided by 7 to blocks can be grouped of
       the same M sub-matrix.                                                      */
    $PairRDD < String, Block > firstMap = BlockRDD$.map();
    /* Group the blocks that comes from the same M sub-matrix                      */
    $PairRDD < string, iterable < Block >> group = firstMap$.groupByKey();
    /* combine the 7 sub-matrices of size n/2 to a single sub-matrix of size n
       having the same key                                                         */
    $RDD < Block > C = group$.flatMap();
    return C;
```

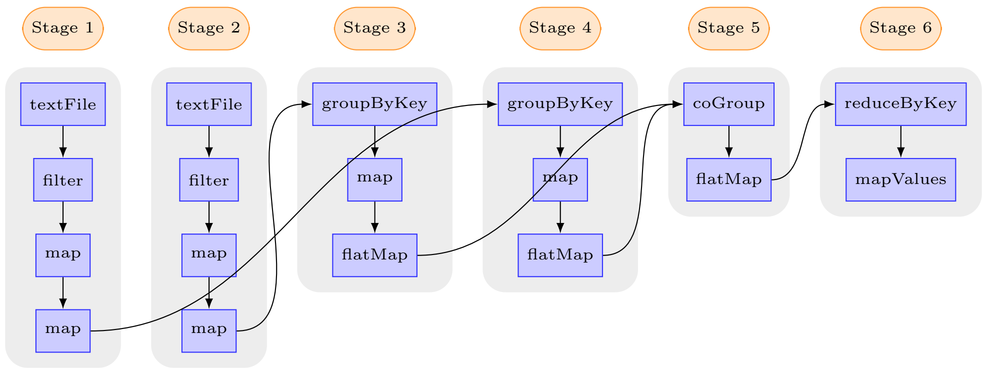


Fig. 5. Lineage graph of *MLLib*

4 PERFORMANCE MODELING

We will make an attempt to estimate the performance of Stark and state-of-the art *Marlin* approach on a hypothetical exascale machine. We follow a methodology from [17], which gives runtime estimation based on the following parameters —

- m = number of rows in matrix A
- k = number of columns in matrix A
- n = number of columns in matrix B
- r = number of rows splits in matrix A
- k = number of column splits in matrix A
- n = number of column splits in matrix B
- $|A|$ = Size of matrix A
- $|B|$ = Size of matrix B

Therefore,

- Total number of blocks in matrix $A = rs$
- Total number of blocks in matrix $B = st$ and
- Total number of blocks in both matrix A and $B = rs + st$

4.1 Cost Analysis of MLLib

In this section we derive the cost of the MM subroutine of *MLLib* package. As we have used block matrix data structure for *Stark*, the same has been used for experimentation among four choices - *RowMatrix*, *IndexedRowMatrix*, *CoordinateMatrix* and *BlockMatrix*. In the pre-processing step we have transformed the input file to *CoordinateMatrix*, and then converetd it into *BlockMatrix*, a data structure synonymous to *Stark's* block matrix structure. While converting to *BlockMatrix*, we provided two parameters - *rowsPerBlock* and *colsPerBlock*. As each block is square, the value of these two parameters are same.

Before multiplying, the method partitions the matrix blocks with unique *partitionID* using *GridPartitioner* approach. It partitions the input matrices in a grid structure and calculates four values - *numRowBlocks* (number of row blocks), *numColBlocks* (number of Column blocks), *rowsPerPart* (number of row blocks per partition) and *colsPerPart* (number of column blocks per partition). It then simulates the multiplication by calculating the destination *partitionIDs* for each block so that only the required blocks is to be shuffled.

Table 1. Stagewise cost analysis of MLLib

Stage-Step	Comp	Network	Parallelization Factor
Stage 3-flatMap	b^3	NA	$min[b^2, cores]$
Stage 4-flatMap	$2b^3$	NA	$min[2b^2, cores]$
Stage 5-co-Group	NA	$2b^2n^2$	$min[b^2, cores]$
Stage 5-flatMap	$(\frac{n}{b})^3$	NA	$min[b^2, cores]$
Stage 6-reduceByKey	NA	$\frac{n}{b}$	$min[b^2, cores]$

This cuts the communication cost. For simulation, all the *partitionIDs* needs to be collected at master node. Therefore, the *IO* cost for this simulation part is

$$\begin{aligned} IO_{simulation} &= (\frac{n}{b})^2 + (\frac{n}{b})^2 \\ &= 2 \times \frac{n^2}{b^2} \end{aligned} \tag{1}$$

We ignore the computation cost here, because the *groupByKey* and *map* steps are done on a single machine, the cost of which much less than the overall cost of the approach.

After simulating, two *flatMap* steps are executed for two input matrices. These are shown in *Stage 3* and *Stage 4* of the lineage of *MLLib* in Fig. 5. The *flatMap* is the actual block replication step where one block of A is taken at a time and replicated as many times as it needs to be multiplied by other blocks of B. This value is the number of *partitionIDs* of the blocks of B , the input block should multiply which is equal to b. Therefore, total number of input blocks are b^2 and computation cost of *Stage 3* and *Stage 4* are

$$Comp_{Stage3} = b^3 \tag{2}$$

The parallelization factor for these two steps is

$$PF_{Stage3} = b^2 \tag{3}$$

After that, the actual shuffling takes place using co-group in *Stage 5*. It groups the values associated to similar keys for both the A and B blocks, which is equal to

$$\begin{aligned} IO_{co-group} &= ((numColPartB)\,|A| + (numRowPartA)\,|B|) \\ &= 2b^2n^2 \end{aligned} \tag{4}$$

The *flatMap* step in *Stage 5* computes the block level multiplication, the cost of which is

$$Comp_{flatMap} = (\frac{n}{b})^3 \tag{5}$$

The parallelization factor of *Stage 5* proportional to the total number of partitions of the product matrix, which is

$$PF_{Stage5} = b^2 \tag{6}$$

The *reduceByKey* step of *Stage 6* aggregates the multiplication terms in a group and add them. The *IO* cost for this step is

$$IO_{reduceByKey} = \frac{n}{b} \tag{7}$$

and the parallelization factor is same as above

$$PF_{Stage6} = b^2 \tag{8}$$

Therefore, the total cost of *MLLib* is

$$\begin{aligned} Cost_{MLLib} &= (IO_{Stage3} + IO_{Stage4} + IO_{Stage5} + Comp_{Stage5} + IO_{Stage6}) \\ &= \frac{b^3 + b^3 + 2b^2n^2 + (\frac{n}{b})^3 + \frac{n}{b}}{min[b^2, cores]} \end{aligned} \tag{9}$$

4.2 Cost Analysis of Marlin

Marlin job execution consists of 6 stages as shown in Fig. 6. However, *Stage 1* and *Stage 3* are part of the pre-processing steps. Also, the *groupByKey* and *mapValues* of *Stage 2* and *Stage 4* are not part of the actual multiplication job execution. Therefore, only *flatMap* step of *Stage 2* and *Stage 4* and entire *Stage 5* and *Stage 6* are part of the actual execution. A summary of the cost analysis is tabulated in Table 2.

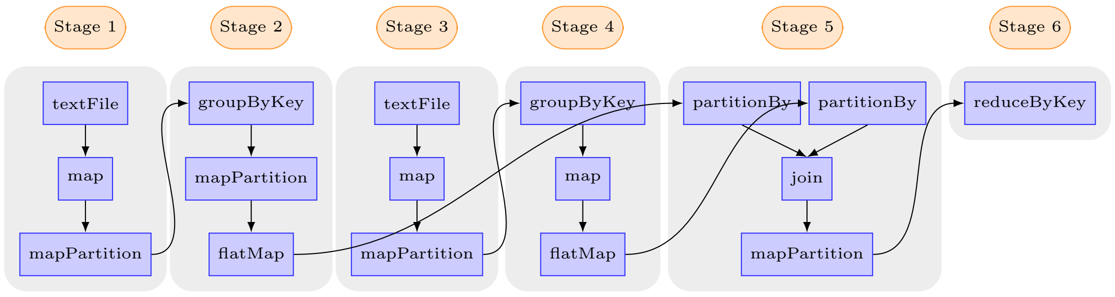


Fig. 6. Lineage graph of *Marlin*

Table 2. Stagewise cost analysis of Marlin

Stage-Step	Comp	Network	Parallelization Factor
Stage 2-flatMap	$2b^3$	$2bn^2$	$min[2b^2, cores]$
Stage 4-flatMap	$2b^3$	$2bn^2$	$min[2b^2, cores]$
Stage 5-Join	NA	bn^2	$min[b^3, cores]$
Stage 5-mapPartition	n^3	bn^2	$min[b^3, cores]$
Stage 6-reduceByKey	NA	bn^2	$min[b^2, cores]$

4.2.1 *Cost in Stage 2.*

The only one step in *Stage 2* that contributes to the cost is the *flatMap* step. Below we derive the cost of *flatMap* step.

Cost in flatMap Step. In this step, each matrix block is taken as input and a list of matrix blocks is returned. Each block of matrix A returns the number of columns of B blocks and each block of B returns the number of rows of A blocks. Therefore, each block of total rs blocks generates t copies of A blocks and each block of total st blocks generates r copies of B blocks. Therefore, IO cost can be derived as

$$\begin{aligned} IO_{flatMap} &= (t\,|A| + r\,|B|) \\ &= (r \times s \times t) + (r \times s \times t) \\ &= 2bn^2 \end{aligned} \tag{10}$$

The parallelization factor can be derived as

$$\begin{aligned} PF_{flatMap} &= (rs + st) \\ &= 2b^2 \end{aligned} \tag{11}$$

We are ignoring the computing cost $Comp_{flatMap}$ cost in the *flatMap* step as it is smaller than the *IO* cost calculated above. Therefore, total cost in *Stage 2* is

$$Cost_{Stage2} = (2bn^2/min[2b^2, cores]) \tag{12}$$

4.2.2 Cost in Stage 4. The cost in *Stage 4* is same as *Stage 2* and therefore, total cost in *Stage 4* is

$$Cost_{Stage2} = (2bn^2/min[2b^2, cores]) \tag{13}$$

4.2.3 Cost in Stage 5. We have two steps in *Stage 5*. They are *join* and *mapPartition*. We compute the *IO* cost for *join* and *Comp* and *IO* cost for *mapPartition*.

Cost in Join Step. In this step, the output from both the *flatMap* steps are joined, so that related blocks stay together for the next *mapPartition* step of *Stage 5*. It shuffles only one matrix (either A or B) through the network. Assuming only shuffling matrix B, the cost spending on the network communication can be derived as

$$\begin{aligned} IO_{Join} &= (r \times |B|) \\ &= (r \times s \times t) \\ &= bn^2 \end{aligned} \tag{14}$$

At the same time, the programs reads blocks in matrix A locally, where the cost is much less than the networking cost and we can omit it in this step.

The parallelization factor is the number of multiplications done for each block times number of blocks in product matrix. Therefore,

$$\begin{aligned} PF_{Join} &= (r \times s \times t) \\ &= b^3 \end{aligned} \tag{15}$$

Cost in mapPartition Step. In this step the MM are conducted locally. The computing cost of this step is

$$\begin{aligned} Comp_{mapPartition} &= ((m/b) \times (n/b) \times (k/b)) \\ &= (n/b)^3 \end{aligned} \tag{16}$$

After the *mapPartition* step the results blocks needs to be written to disks for the next shuffle phase. The *IO* cost can be derived as

$$\begin{aligned} IO_{mapPartition} &= (s \times |C|) \\ &= (r \times s \times t) \\ &= bn^2 \end{aligned} \tag{17}$$

The parallelization factor is the number of multiplications done for each block times number of blocks in product matrix. Therefore,

$$\begin{aligned} PF_{mapPartition} &= (r \times s \times t) \\ &= b^3 \end{aligned} \tag{18}$$

Therefore, total cost in *Stage 5* is

$$\begin{aligned} Cost_{Stage5} &= (IO_{Join}/PF_{Join}) + ((Comp_{mapPartition} + IO_{mapPartition})/PF_{mapPartition}) \\ &= (bn^2/min[b^3, cores]) + ((n^3 + bn^2)/min[b^3, cores]) \end{aligned} \tag{19}$$

4.2.4 Cost in Stage 6. Cost in *Stage 6* is the *IO* cost in only step of this stage i.e. *reduceByKey* step.

Cost in reduceByKey Step. The *IO* cost in this step is

$$\begin{aligned} IO_{reduceByKey} &= (s \times |C|) \\ &= (r \times s \times t) \\ &= bn^2 \end{aligned} \tag{20}$$

In this step, the parallelization factor is the number of additions can be done in parallel which is equal to the number of blocks in the product matrix. Therefore,

$$\begin{aligned} PF_{reduceByKey} &= (r \times t) \\ &= b^2 \end{aligned} \tag{21}$$

Total cost in *Stage 6* is

$$\begin{aligned} Cost_{Stage6} &= (IO_{reduceByKey}/PF_{reduceByKey}) \\ &= (bn^2/min[b^2, cores]) \end{aligned} \tag{22}$$

4.3 Cost Analysis of Stark

Unlike *MLLib* and *Marlin*, *Stark* does not posses a constant number of stages. It depends on the number of recursive call to the algorithm. The algorithm can be divided into three main sections - *Divide*, *Multiply* and *Combine*. *Divide* section recursively divides the input matrices into seven sub-matrices and is done by three steps - *flatMap*, *groupByKey* and *mapPartition*. The number of occurrences of these three steps depend on the number of recursive calls. The number of recursive calls are again equal to the logarithm of the number of partitions of the matrix. For example, for a $2^p \times 2^p$ matrix and $2^q \times 2^q$ matrix blocks, these three steps occur $\log_2(2^p/2^q) = (p - q)$ times. These three steps covers $(p - q)$ entire stages and first two of total three steps of an additional stage.

Multiply section does the actual multiplication of leaf node matrix blocks. It consists of three steps - *map*, *groupByKey* and *flatMap*. The first step occupy the last step of the previous stage and the next two steps occupy the first two steps of the next stage.

Table 3. Stage-wise cost analysis of Stark

Stage-Step	Comp	Network	P.F.
Stage 1 to Stage $p-q$ - map Divide	$\sum_{i=0}^{p-q-1}(7/2)^i(2b^2)$	$\sum_{i=0}^{p-q-1}3.(7/2)^i(2n^2)$	$min[7^i, cores]$
Stage 1 to Stage $p-q$ - groupByKey Divide	NA	$\sum_{i=0}^{p-q-1}3.(7/2)^i(2n^2)$	$min[7^i, cores]$
Stage 2 to Stage $p-q+1$ - flatMap Divide	$\sum_{i=1}^{p-q}(7/2)^i(2b^2)$	NA	$min[7^i, cores]$
Stage $p-q+1$ - map Leaf	NA	$(7/4)^{p-q}(2n^2)$	$min[7^{p-q}, cores]$
Stage $p-q+2$ - groupByKey Leaf	NA	$(7/4)^{p-q}(2n^2)$	$min[7^{p-q}, cores]$
Stage $p-q+1$ - map Leaf	$(7/4)^{p-q}(n^3)$	NA	$min[7^{p-q}, cores]$
Stage $p-q+2$ to Stage $2p-2q+1$ - map Combine	$\sum_{i=p-q}^{1}(7/4)^i(b^2)$	$\sum_{i=p-q}^{1}(7/2)^i(n^2)$	$min[7^i, cores]$
Stage $p-q+3$ to Stage $2p-2q+2$ - groupByKey Combine	NA	$\sum_{i=p-q}^{1}(7/4)^i(n^2)$	$min[7^i, cores]$
Stage $p-q+3$ to Stage $2p-2q+2$ - flatMap Combine	$(12+8.(n)^2).4^{i-1}$	NA	$min[7^i, cores]$

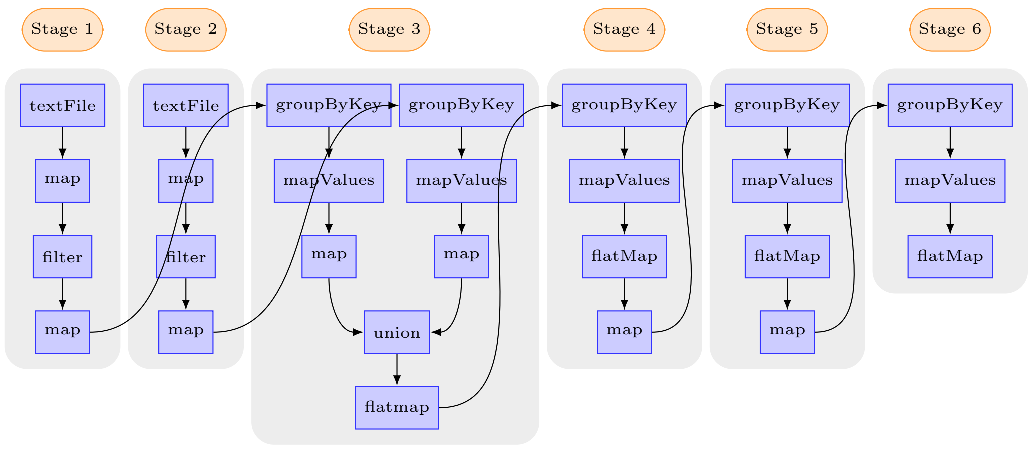


Fig. 7. Lineage graph of *Stark*

Combine section combines the sub-matrices into a single matrix after the recursive call finishes. Like the first section, this section also consists of three steps - *map*, *groupByKey* and *flatMap* and their occurrences are governed by the same logarithm function. Therefore, the total number of stages for a matrix of size of $2^p \times 2^p$ matrix and $2^q \times 2^q$ matrix blocks is

$$\begin{aligned} stages &= 2\log_2(2^p/2^q) + 2 \\ &= 2\log_2 2^{p-q} + 2 \\ &= 2(p-q) + 2 \end{aligned} \tag{23}$$

Fig. 7 shows the lineage of *Stark* when the value of $(p-q)$ is equal to 1. For other scenarios, the lineage can be drawn according to equation 23. The cost analysis is summarized in Table 3.

4.3.1 Cost Analysis in Divide Section. As already stated, *Divide* section consists of three steps - *flatMap*, *groupByKey* and *mapPartition*. Each of the steps occurs $(p-q)$ times.

Cost Analysis of flatMap Step. There are two types of cost associated with this step - computation cost and IO cost. The computation cost for a single *flatMap* can be derived as

$$\begin{aligned} Comp_{SingleflatMap} &= (|A| + |B|) \\ &= (rs + st) \\ &= 2b^2 \end{aligned} \tag{24}$$

As the recursion tree going down to the leaf nodes, the sub-matrix size starts to decrease whereas the number of nodes at each level increases 7 times. Therefore, the total cost for this step for all the stages can be derived as

$$Comp_{flatMap} = \sum_{i=0}^{p-q-1} (7/2)^i (2b^2) \tag{25}$$

The IO cost can be derived similarly as

$$IO_{flatMap} = \sum_{i=0}^{p-q-1} 3.(7/2)^i (2n^2) \tag{26}$$

As the program progresses the parallelization factor increases as $(7/4)^i(2b^2)$ for $(i = 0, 1, 2, ..., (p-q-1))$.

Cost Analysis of groupByKey Step. The network cost of this step is

$$IO_{groupByKey} = \sum_{i=0}^{p-q-1} 3.(7/2)^i (2n^2) \tag{27}$$

Cost Analysis of mapPartition Step. The compute cost of *mapPartition* step can be derived as

$$Comp_{mapPartition} = \sum_{i=1}^{p-q} (7/2)^i (2b^2) \tag{28}$$

As the program progresses the parallelization factor increases as $(7)^i$ for $(i = 1, 2, ..., (p-q))$.

4.3.2 Cost Analysis of Leaf Node Multiplication Section. There are three steps for this section - *mapPartition*, *groupByKey* and *map*. Each one occurs only once. The IO cost of *mapPartition* step can be derived as

$$IO_{mapPartition} = (7/4)^{p-q}(2n^2) \tag{29}$$

The *IO* cost for *groupByKey* step is

$$IO_{groupByKey} = (7/4)^{p-q}(2n^2) \tag{30}$$

and the *compute* cost for *map* step is

$$Comp_{map} = (7/4)^{p-q}(n^3) \tag{31}$$

and the parallelization factor is $(7)^{p-q}$.

4.3.3 Cost Analysis of Combine Section. The *Compute* cost for *mapToPair* step can be derived as

$$Comp_{mapPartition} = \sum_{i=p-q}^{1} (7/4)^i (b^2) \tag{32}$$

and *IO* cost is same and parallelization factor is $(7)^i$ for $(i = (p-q), (p-q-1), ..., 1$.
The *IO* cost for *groupByKey* step is

$$IO_{groupByKey} = \sum_{i=p-q}^{1} (7/4)^i (n^2) \tag{33}$$

and the parallelization factor is same as before.
The *Compute* cost for *flatMap* step is

$$Comp_{flatMap} = \sum_{i=p-q}^{1} (7)^{i-1}(7 + b^2/4^i) \tag{34}$$

Total cost for *Stage 1*

$$Cost_{Stage1} = \frac{6n^2}{min[1, cores]} \tag{35}$$

Total Cost for *Stage 2* to *Stage* $(p-q)$

$$\begin{aligned} Cost_{Stage2to(p-q)} = &\sum_{i=0}^{p-q-2} \frac{3.(7/2)^i.(2n^2)}{min[(7/4)^i \times 2b^2, cores]} + \sum_{i=0}^{p-q-1} \frac{3.(7/2)^i.(2n^2)}{min[7^{i+1}, cores]} \\ &+ \sum_{i=1}^{p-q} \frac{(7/2)^i(2b^2)}{min[7^{i+1}, cores]} + \frac{(7/4)^{p-q}(2n^2)}{min[7^{p-q}, cores]} \end{aligned} \tag{36}$$

Total Cost for *Stage (p-q+1)*

$$Cost_{Stage(p-q+1)} = \frac{(7/4)^{p-q}(2n^2)}{min[7^{p-q}, cores]} + \frac{(7/4)^{p-q}(n^3)}{min[7^{p-q}, cores]} + \frac{(7/4)^{p-q}(b^2)}{min[7^i, cores]} \tag{37}$$

Total cost for *Stage (p-q+1)* to *Stage (2(p-q)+1)*

$$
\begin{aligned}
Cost_{Stage(p-q+2)to(2(p-q)+1)} = &\sum_{i=p-q-1}^{1} \frac{(7/4)^i(n^2)}{min[7^i, cores]} + \sum_{i=p-q-1}^{1} \frac{7^{i-1}(7+b^2/4^i)}{min[7^i, cores]} \\
&+ \sum_{i=p-q-1}^{1} \frac{(7/4)^i(b^2)}{min[7^i, cores]}
\end{aligned}
\tag{38}
$$

Total cost for *Stage 2(p-q)+2*

$$
Cost_{Stage(2(p-q)+2)} = \frac{(7/4)n^2}{min[7, cores]} + \frac{(12+8n^2)\times 4^{i-1}}{min[7, cores]} \tag{39}
$$

5 EXPERIMENTS

In this section, we perform experiments to evaluate the execution efficiency of *Stark* comparing it with *Marlin* and *MLLib* and scalability of the algorithm compared to ideal scalability. First, we select the fastest execution time among different partition size for each approach and compare them. Second, we conduct a series of experiments to individually evaluate the effect of partition size and matrix size of each competing approach. At last, we evaluate the scalability of *Stark*.

5.1 Test Setup

All the experiments are carried out on a dedicated cluster of 3 nodes. Software and hardware specifications are summarized in Table 4. Here NA means not applicable.

Table 4. Summary of Test setup components specifications

Component Name	Component Size	Specification
Processor	2	Intel Xeon 2.60 GHz
Core	6 per processor	NA
Physical Memory	132 GB	NA
Ethernet	14 Gb/s	Infini Band
OS	NA	CentOS 5
File System	NA	Ext3
Apache Spark	NA	1.6.0
Apache Hadoop	NA	2.6.0
Java	NA	1.7.0 update 79

For block level multiplications *Stark* uses Breeze, a single node low-level linear algebra library. Breeze provides Java APIs to *Stark* through Spark, and it calls C/Fortran-implemented native library, such as BLAS, to execute the linear algebra computation through the Java Native Interface (JNI). We have tested the algorithms on matrices with increasing cardinality from (16×16) to (16384×16384). All of these test matrices have been generated randomly using Java Random class. The elements of the matrices are of double-precision 64-bit IEEE 754 floating point type.

5.1.1 Resource Utilization Plan. While running the jobs in the cluster, we customize three parameters — the number of executors, the executor memory and the executor cores. We wanted a fair comparison among the competing approaches and therefore, we ensured jobs should not experience *thrashing* and none of the cases tasks should fail and jobs had to be restarted. For this reason, we restricted ourselves to choose the parameters value which provides good utilization of cluster resources and mitigating the chance of task failures. By experimentation we found that, keeping executor memory as 50 GB ensures successful execution of jobs without *"out of memory"* error or any task failures for all the competing approaches. This includes the small amount of overhead to determine the full request to YARN for each executor which is equal to 3.5 GB. Therefore, the executor memory is 46.5 GB. Though the physical memory of each node is 132 GB, we keep only 100 GB as YARN resource allocated memory for each node. Therefore, the total physical memory for job execution is 100 GB resulting 2 executors per node and a total 6 executors. We reserve, 1 core for operating system and hadoop daemons. Therefore, available total core is 11. This leaves 5 cores for each executor. We used these values of the run time resource parameters in all the experiments except the scalability test, where we have test the approach with varied number of executors. The resource utilization plan is summarized in Table 5.

Table 5. Resource Utilization Plan for Efficiency Experiments of Stark with *MLLib* and *Marlin*

Component Name	Component Size
Executors	5
Executor Cores	5
Executor Memory	40GB
YARN Memory	100GB

5.2 Efficiency

5.2.1 Practical Utility of Stark: Comparison with state-of-the-art distributed systems. In this section, we examine the performance of Stark with other Spark based distributed MM approaches i.e. *Marlin* and *MLLib*. We report the running time of the competing approaches with increasing matrix size. We take the best running time (fastest) among all the running time taken for different block sizes. The experimental results are shown in Fig. 8. It can be seen that performance of *Stark* is faster than the *Marlin* and *MLLib*. Also, the performance is increased as matrix size is increased from (4096×4096) to (16384×16384).

5.2.2 Variation with matrix size. In this experiment we compare the running time for increasing matrix size. For each matrix size we compare the running time for increasing partition size. Fig. 9 shows the comparison.

The results show that, as matrix size increases the execution time of Stark grows slowly than others providing faster execution at 16 partition size in 8192×8192 and all the partition sizes in 16384×16384. Moreover, as the partition size increases the difference in running time also increases.

5.2.3 Comparison with state of the art single node systems. The second experiment (Table 6) examines how the runtime improves if we use Stark on the cluster compared to the serial MM on a single node having similar configuration of a single node in the cluster. The intension of this experiment is to provide a demonstration of how well our approach

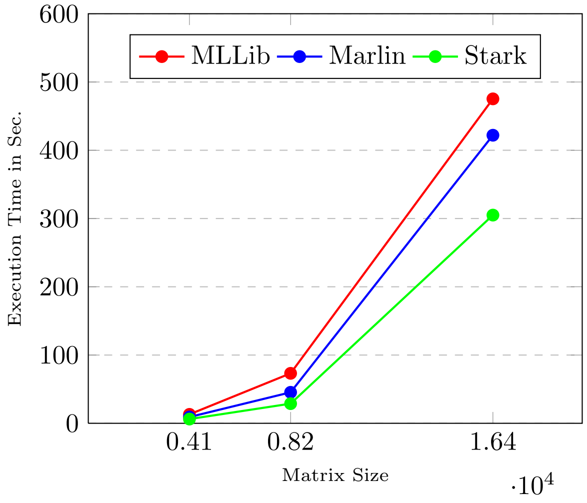
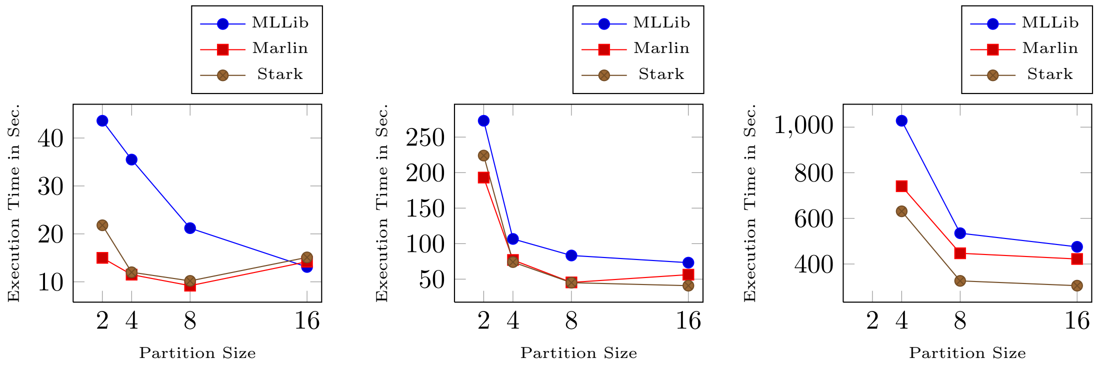


Fig. 8. Fastest running time of three systems among different block sizes



(a) 4096×4096 (b) 8192×8192 (c) 16384×16384

,

Fig. 9. Comparing running time of Stark, Marlin and MLLib for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing partition size

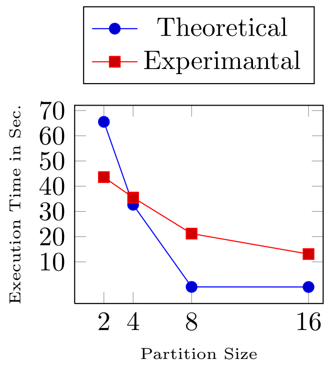
performs when robust linear algebraic libraries are run on a single node with equal power and specifications. To present such comparison, we use Colt [10] and JBlas [22], two linear algebra routines implementing MM algorithm having optimized code. Colt is a set of open source libraries for high performance technical computing in Java. On the other hand, JBlas is a fast linear algebra library for Java based on BLAS and LAPACK. We use the parallel version of Colt library, named ParallelColt [39] which uses threads automatically when computations are done on a machine with multiple CPUs. Additionally, we use two other variations of single node serial MM algorithm — the three loop naïve approach and the single node Strassen's MM algorithm. We use *NA* when the execution time is more than reasonable time i.e. more than 1 hour.

Table 6. Performance comparison of five systems with increasing matrix size (The unit of execution time is second)

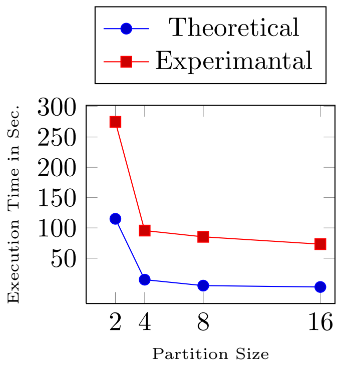
Matrix	SN	SS	Colt	JBlas	Stark
512×512	< 1	< 1	< 1	< 1	5
1024×1024	15	2	1	< 1	6
2048×2048	177	14	13	2	9
4096×4096	2112	100	135	16	6.2
8192×8192	NA	394	1163	119	28.8
16384×16384	NA	2453	NA	862	305

For each matrix of size $2^n \times 2^n$, we take the running time increasing the block size from 2 to $2^{\frac{n}{2}} \times 2^{\frac{n}{2}}$. We take the fastest running time for each matrix size and report it. The results shows that *Stark* outperforms others with a very substantial margin.

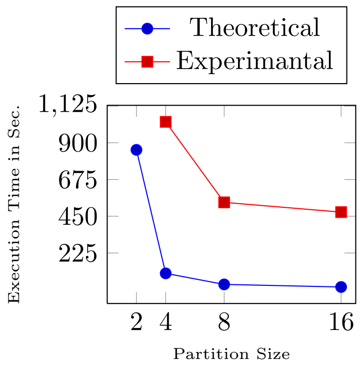
5.2.4 Variation with partition size. In this experiment, we show that there is an optimal partition size for each of the competing method. To demonstrate that we generate three scenarios for each method. We increase matrix size from (4096×4096) to (16384×16384) respectively and increase partition size from 4 to 16 in each scenario. Fig. 10, Fig. 11 and Fig. 12 show scenarios respectively.



(a) 4096×4096



(b) 8192×8192



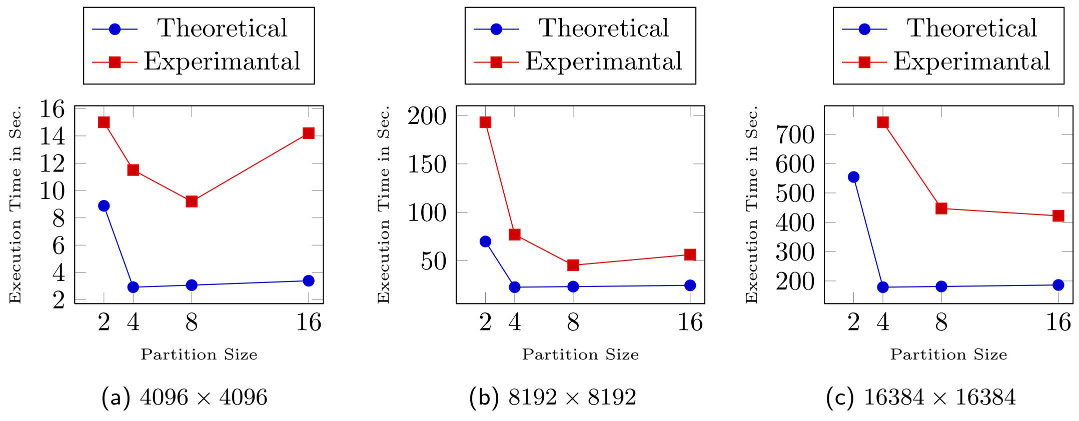
(c) 16384×16384

,

Fig. 10. Comparing theoretical and experimental running time of MLLib for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing partition size

It is clearly seen that MLLib execution time (experimental) entirely matches with the theoretical one. Both are decreasing with increasing partition size. The costliest part of MLLib is the computation of multiplication of block matrices, which is $((n/b)^3/min[b, 25])$. For partition size $b = 2$ and 4 the factor decreases drastically because, the denominator becomes b^3. After that, the parallelization factor becomes constant which is 25, the number of cores designated for the spark job. So, the denominator b affects the execution time and thus it decreases gradually.

For *Marlin*, the execution time first decreases and then increases for increasing partition size both for theoretical and experimental running time. The reason is again the term that



,

Fig. 11. Comparing theoretical and experimental running time of Marlin for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing partition size

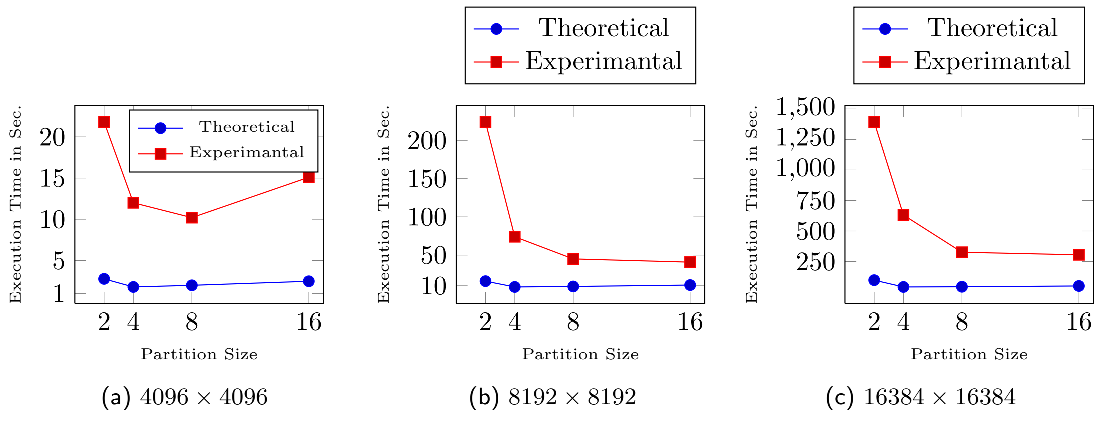
represents the most costly portion of the execution i.e. the computation of block MMs which is equal to $(b^3 \times (n/b)^3/min[b^3, 25])$. Except partition size $b = 2$, for all other b's the P.F. is 25. Therefore, this factor is same for all other partition size. Marlin has a join operation before this step whose cost is $(bn^2/min[b^3, 25])$. As this term influence very little than the earlier computation step, the execution time increases slowly with increasing partition size.

One more thing that we need to explain is that the partition size at which minimum execution time happens, is not constant. For matrix size 4096×4096 and 8192×8192, the partition size is 8, but for matrix size 16384×16384 it is 16. Once again the $(b^3 \times (n/b)^3/min[b^3, 25])$ factor comes into play. According to Table 6, the difference between execution time of matrix size 1024×1024 and 2048×2048 is almost 2 seconds. However, the P.F. does not change when we increase the partition size from 8 to 16. This means larger number of 1024×1024 multiplications are faster than smaller number of 2048×2048 multiplications. That is why, the curve dips again from $b = 8$ to $b = 16$. This phenomenon is not captured in the theoretical analysis.

The cost analysis of *Stark* is almost same as *Marlin*, except the fact that, the P.F. is different and it uses the cores as much as possible while increasing partition size. The P.F. for most costly operation i.e. block MM $((7)^{p-q}((n/b)^{2.8}) \times min[7^{p-q}, 25])$, has a P.F. of $(min[7^{p-q}, 25])$. This means it uses the number of available cores except $b = 2$. That is why we get minimum at $b = 4$. The experimental execution time is lowest at $b = 8$ for (4096×4096). The reason is that the time difference between multiplying two (256×256) and (512×512) is very small. Now, the number of (256×256) multiplications performed for $b = 16$ is more than the number of (512×512) multiplications performed for $b = 8$. Similar to *Marlin*, the minimum shifts right and reaches at $b = 16$ in the rest of the cases.

5.2.5 *Stage-wise Comparison.*

In this experiment, we compare the running time of three systems stage-wise. From this test we can infer the time consuming stage for each approach. We perform this test for increasing matrix size and for each matrix size with increasing partition size. As the number od stages of *Stark* grows on the the order of $(p - q)$, for large partition size, its value becomes too large to compare with other approaches. For this reason,



,

Fig. 12. Comparing theoretical and experimental running time of Stark for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing partition size

we merge the stages of *Stark* to form 3 stages, comprising stages related to divide, leaf node multiplication and combine phases. Fig. 13 shows the comparison.

From figure, it is clearly understood that, the primary bottleneck for *MLLib* is computation in Stage 3 which involves block matrix multiplication step on a single executor. It takes more or less (60-85)% of the total execution time and the percentage increases gradually from (60-70)%, (75-80)%, to (70-86)% for matrix size (4096×4096), (8192×8192) and (16384×16384) respectively. The costliest part of *Stage 3* is $(n^{2.8}/min[b, 25])$, which explains above findings easily. Also, it explains gradual increase in execution time when matrix size is increased.

For *Marlin*, it is again block matrix multiplication step which provides the costliest part. Like *MLLib*, the cost of *Stage 3* first decreases, but increases at $b = 16$ for matrix (4096×4096) and (8192×8192). For matrix (16384×16384), its always decreasing. Moreover, this cost increases with increasing matrix size. The costliest term for *Stage 3* is $(b^3 \times (n/b)^{2.8}/min[b^3, 25])$, again explains above.

For *Stark*, the most expensive part is same as the other two approaches discussed above. The main factor that makes *Stark* to be faster is its ability to preserve the number of multiplications to be smaller than the other two approaches. To elaborate the explanation we have tabulated the stage-wise running time of the approaches with increasing partition size for all matrix sizes. The communication intensive stages are colored as green while computing intensive stages are filled with red. It is clear from tables 7, 8 and 9, the most costly stage is the stage that contains the block MM calculation. This cost is almost same for smaller partition size, but the gap increases as we move to larger partition size. The reason is that, $((7)^{p-q}((n/b)^{2.8}) \times min[7^{p-q}, 25])$ factor in *Stark* grows slowly than the factor $(b^3 \times (n/b)^{2.8}/min[b^3, 25])$ in *Marlin*. As we increase the partition size b, the number of multiplications for *Marlin* to be carried out grows in b^3 order, while *Stark* grows in 7^{p-q} order, which is less than the earlier. Again, *Stark* makes the division and combination phase to be parallel enough, making it faster compared to any other approaches.

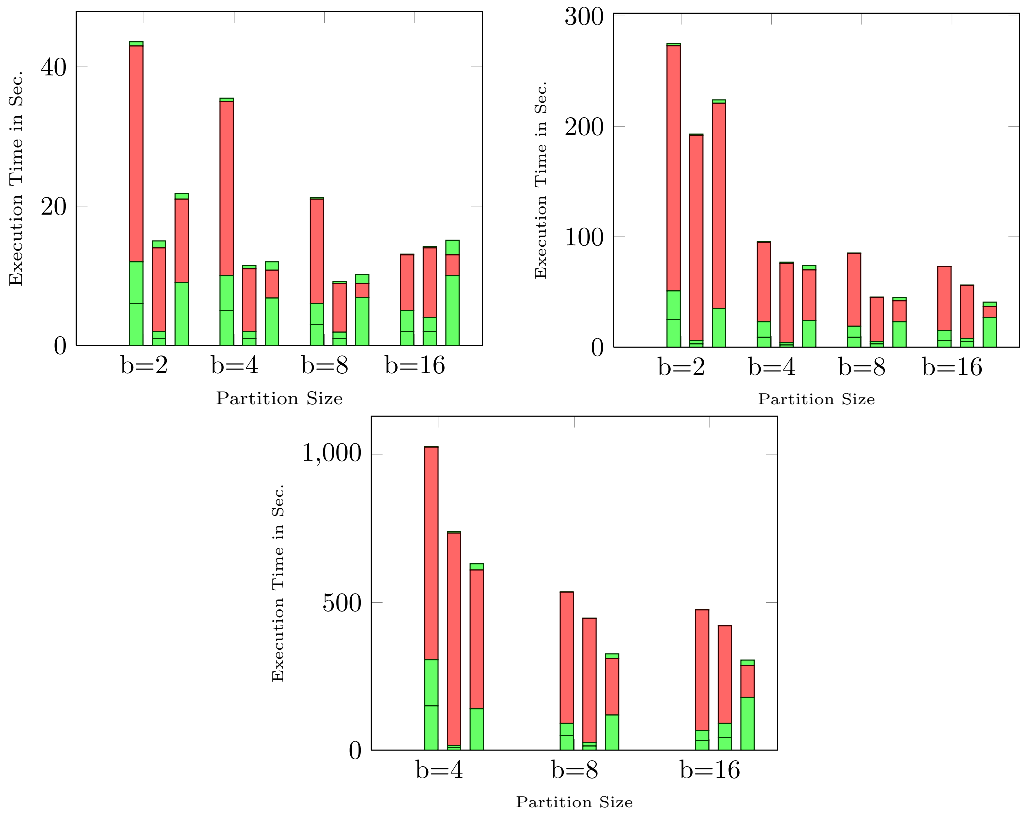


Fig. 13. Comparing stage-wise running time of MLLib, Marlin and Stark for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing partition Size

5.3 Scalability

In this section, we investigate the scalability of *Stark*. For this, we generate three test cases, each containing a different set of two matrices of dimensions equal to (4096×4096), (8192×8192) and (16384×16384). The running time vs. the number of spark executors for these 3 pairs of matrices is shown in Fig. 14. The ideal scalability line (i.e. $T(n) = T(1)/n$ - where n is the number of executors) has been over-plotted on this figure in order to demonstrate the scalability of our algorithm. We can see that *Stark* has a very good strong scalability, with a minor deviation from ideal scalability when the size of the matrix is low (i.e. for (8192×8192) and (16384×16384)).

6 CONCLUSIONS AND FUTURE WORK

With *Stark*, we developed an algorithm for MM which implemented Strassen's MM algorithm distributedly in a commodity cluster. *Stark* provides a faster, scalable and massively parallel approach to multiply two large matrix in Spark compared to state-of-the-art *MLLib* and *Marlin*. It accomplishes that using creating a recursive tree of computation and performing parallel computation on each node of a level of that tree. Additionally, it tags the block matrices intelligently to memorize them in the subsequent recursive calls. Our theoretical

Table 7. Stage-wise performance comparison of three systems with increasing partition (The unit of execution time is second) for matrix size 4096×4096

Stage	Execution Time											
	b=2			b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S	M1	M2	S
Stage 1	6	1	8	5	1	4	3	1	4	2	2	4
Stage 2	6	1	1	5	1	2	3	0.9	1	3	2	1
Stage 3	31	12	12	25	9	0.8	15	7	1	8	10	1
Stage 4	0.6	1	0.8	0.5	0.5	4	0.2	0.3	0.9	0.1	0.2	2
Stage 5						0.5			2			2
Stage 6						0.7			0.5			3
Stage 7									0.5			0.9
Stage 8									0.3			0.5
Stage 9												0.4
Stage 10												0.3

Table 8. Stage-wise performance comparison of three systems with increasing partition (The unit of execution time is second) for matrix size 8192×8192

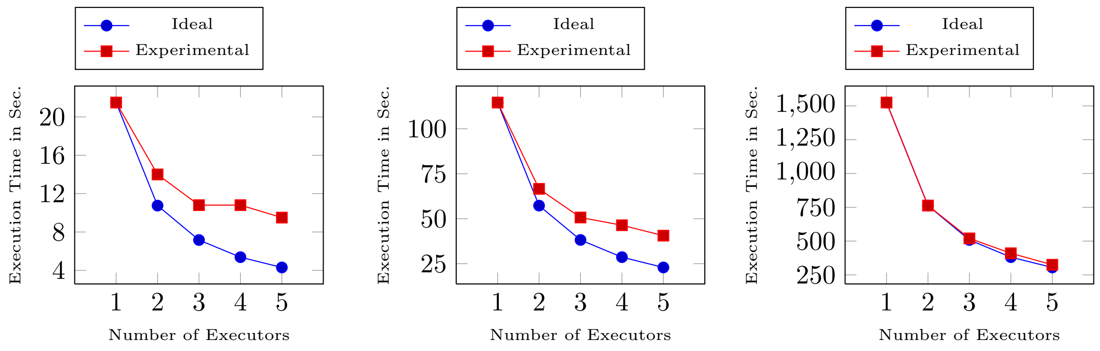
Stage	Execution Time											
	b=2			b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S	M1	M2	S
Stage 1	25	3	29	9	2	15	9	3	12	6	5	12
Stage 2	26	3	6	14	2	6	10	2	4	9	3	4
Stage 3	222	186	186	72	72	3	66	40	4	58	48	3
Stage 4	2	1	3	0.6	1	46	0.3	0.4	3	0.2	0.3	4
Stage 5						2			19			4
Stage 6						2			1			10
Stage 7									1			1
Stage 8									1			0.9
Stage 9												0.9
Stage 10												1

analysis shed light on the time consuming stages of the competing approaches and experimental results matches with the earlier. The experimental results show that the scalability of *Stark* becomes highly robust when matrix size is increased.

The next step from the practical side are experiments with massive matrix sizes (for example, $2^{17} \times 2^{17}$ using more number of nodes and executors. On the theoretical side, we could look for faster efficient multiplication algorithm such as Winograd variation [11] of

Table 9. Stage-wise performance comparison of three systems with increasing partition (The unit of execution time is second) for matrix size 16384×16384

Stage	Execution Time								
	b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S
Stage 1	150	9	90	49	14	48	33	43	43
Stage 2	156	6	37	42	12	26	34	48	36
Stage 3	720	720	13	444	420	24	408	330	21
Stage 4	2	6	470	0.5	1	21	0.2	1	46
Stage 5			9			192			33
Stage 6			12			4			108
Stage 7						6			6
Stage 8						5			3
Stage 9									4
Stage 10									5



(a) 4096×4096 (b) 8192×8192 (c) 16384×16384

,

Fig. 14. The scalability of *Stark*, in comparison with ideal scalability (blue line), on matrix (4096×4096), (8192×8192) and (16384×16384)

Strassen's MM algorithm. In future, we will look into implementing Strassen's and Winograd variation on rectangular matrices.

REFERENCES

[1] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, M Joshi, and P Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.

[2] Apache hadoop project 2016. Apache hadoop project. https://hadoop.apache.org/. (2016). [Online; accessed 23-December-2016].

[3] Apache spark, lightning-fast cluster computing 2016. Apache spark, lightning-fast cluster computing. https://spark.apache.org/. (2016). [Online; accessed 23-December-2016].

[4] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 193–204.

[5] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2012. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)* 59, 6 (2012), 32.

[6] Jarle Berntsen. 1989. Communication efficient matrix multiplication on hypercubes. *Parallel computing* 12, 3 (1989), 335–342.

[7] Lynn E Cannon. 1969. *A CELLULAR COMPUTER TO IMPLEMENT THE KALMAN FILTER ALGORITHM.* Technical Report. DTIC Document.

[8] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*. IEEE, 120–127.

[9] Jaeyoung Choi, David W Walker, and Jack J Dongarra. 1994. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience* 6, 7 (1994), 543–570.

[10] Colt - Welcome 2017. Colt - Welcome. https://dst.lbl.gov/ACSSoftware/colt/. (2017). [Online; accessed 6-April-2017].

[11] Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation* 9, 3 (1990), 251–280.

[12] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 261–272.

[13] Frédéric Desprez and Frédéric Suter. 2004. Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms. *Concurrency and Computation: practice and experience* 16, 8 (2004), 771–797.

[14] Craig C Douglas, Michael Heroux, Gordon Slishman, and Roger M Smith. 1994. GEMMW: a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *J. Comput. Phys.* 110, 1 (1994), 1–10.

[15] Bogdan Dumitrescu, Jean-Louis Roch, and Denis Trystram. 1994. Fast matrix multiplication algorithms on MIMD architectures. *Parallel algorithms and applications* 4, 1-2 (1994), 53–70.

[16] Brian Grayson and Robert Van De Geijn. 1996. A high performance parallel Strassen implementation. *Parallel Processing Letters* 6, 01 (1996), 3–12.

[17] Rong Gu, Yun Tang, Zhaokang Wang, Shuai Wang, Xusen Yin, Chunfeng Yuan, and Yihua Huang. 2015. Efficient large scale distributed matrix computation with spark. In *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2327–2336.

[18] Botong Huang, Nicholas WD Jarrett, Shivnath Babu, Sayan Mukherjee, and Jun Yang. 2015. Cümülön: Matrix-based data analytics in the cloud with spot instances. *Proceedings of the VLDB Endowment* 9, 3 (2015), 156–167.

[19] Sascha Hunold, Thomas Rauber, and Gudula Rünger. 2008. Combining building blocks for parallel multi-level matrix multiplication. *Parallel Comput.* 34, 6 (2008), 411–426.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[21] Bharat Kumar, C-H Huang, P Sadayappan, and Rodney W Johnson. 1995. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming* 4, 4 (1995), 275–289.

[22] Linear Algebra for Java 2017. Linear Algebra for Java. http://jblas.org/. (2017). [Online; accessed 6-April-2017].

[23] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel strassen: Implementation and performance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 101.

[24] Qingshan Luo and John B Drake. 1995. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings of the 1995 ACM symposium on Applied computing*. ACM, 221–226.

[25] William F McColl. 1995. *A BSP realisation of Strassen's algorithm.* Technical Report. Citeseer.

[26] William F McColl and Alexandre Tiskin. 1999. Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24, 3 (1999), 287–297.

[27] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, and others. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 34 (2016), 1–7.

[28] John Norstad. 2009. A MapReduce algorithm for matrix multiplication. http://norstad.org/matrix-multiply/index.html. (2009). [Online; accessed 23-December-2016].

[29] Yuhsuke Ohtaki, Daisuke Takahashi, Taisuke Boku, and Mitsuhisa Sato. 2004. Parallel implementation of Strassen's matrix multiplication algorithm for heterogeneous clusters. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International.* IEEE, 112.

[30] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 197–210.

[31] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. 2010. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on.* IEEE, 721–726.

[32] Edgar Solomonik, Abhinav Bhatele, and James Demmel. 2011. Improving communication performance in dense linear algebra via topology aware collectives. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 77.

[33] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[34] Edgar Solomonik and James Demmel. 2012. Matrix multiplication on multidimensional torus networks. In *International Conference on High Performance Computing for Computational Science*. Springer, 201–215.

[35] Fengguang Song, Jack Dongarra, and Shirley Moore. 2006. Experiments with Strassens algorithm: from sequential to parallel. *Ratio* 2, 3 (2006), 3–5.

[36] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4 (1969), 354–356.

[37] Mithuna Thottethodi, Siddhartha Chatterjee, and Alvin R Lebeck. 1998. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1–14.

[38] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience* 9, 4 (1997), 255–274.

[39] Piotr Wendykier and James G Nagy. 2010. Parallel colt: A high-performance java library for scientific computing and image processing. *ACM transactions on mathematical software (TOMS)* 37, 3 (2010), 31.

[40] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2.

Received February 2007; revised March 2009; accepted June 2009