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

Chandan Misra, Sourangshu Bhattacharya, and Soumya K. Ghosh

Abstract—This article 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 seven multiplications scheme in a distributed environment and thus achieves asymptotically faster execution time. It creates a distributed recursion tree of computation where each level of the tree corresponds to division and combination of distributed matrix blocks stored in the form of Resilient Distributed Datasets (RDDs). It processes each divide and combine step in parallel and memorises the sub-matrices by intelligently tagging matrix blocks in it. To the best of our knowledge, Stark is the first implementation of a distribute Strassen's algorithm on Spark platform. We also report a detailed complexity analysis for the proposed algorithm, taking into account computation and communication costs. Experimental results suggest that Stark outperforms existing distributed matrix multiplication implementations on Spark – *Marlin* and *MLLib*, for high matrix sizes ($\geq 16384 \times 16384$). Our experiments reveal optimal block sizes for each matrix size, which is also shown from theoretical analysis. We also show that the experimental and theoretical running times for Stark match closely. It has also been shown experimentally that Stark exhibits strong scalability with increasing number of executors.

Index Terms—Numerical linear algebra, distributed algorithm, matrix multiplication, strassen's algorithm, apache spark

1 Introduction

Growth in the number of massive datasets from different sources like social media, weather sensors, mobile devices, etc. has led to applications of these datasets for various data-driven research and analytics in domains such as machine learning, climate science, social media analytics, etc. These applications require large-scale data processing with minimal effort and a system which scales as data grows without any failure. Many of these applications need matrix computations on massive datasets, leading to a requirement of large-scale distributed matrix computations.

Big Data processing frameworks like Hadoop MapReduce [1] and Spark [2] have emerged as next-generation distributed programming platform for data-intensive complex analytics and developing distributed applications in above mentioned data-driven domains. Spark has gained its popularity for its in-memory data processing ability to run programs faster than Hadoop MapReduce. Its general purpose engine supports a wide range of applications, including batch, interactive, iterative algorithms and streaming. Spark also offers simple APIs and rich built-in libraries like MLLib [3], GraphX [4] for data science tasks and data processing applications. Therefore, we can get substantial gain by implementing computing intensive algorithms which consume large datasets as input. In the present work, we focus on the problem of distributed multiplication of large and possibly distributed matrices using the Spark framework. For simplicity, we focus on square matrices of form $2^n \times 2^n$. General matrix sizes can be handled by conceptually padding them with zeros to the next highest size.

Many existing works have implemented distributed matrix multiplication on Big Data frameworks. One of the early works was *HAMA* [5], which implemented distributed matrix multiplication on MapReduce. However, this scheme suffers from the shortcomings of Hadoop, i.e., communicating with HDFS for each map or reduce task. This drawback can be overcome by using the Spark framework, which supports distributed in-memory computation. The most widely used approach is the distributed matrix multiplication scheme used in its built-in machine learning library, called *MLLib*. Another recent distributed matrix multiplication scheme is *Marlin* [6], [7], which selects one of among three matrix multiplication algorithms according to the size, i.e., *Block splitting* for square, *CARMA* for rectangular and *Broadcast* for small matrices. However, both MLLib and Marlin used naïve distributed block matrix multiplication approach. Additionally, both approaches require 8 block multiplications to calculate the product matrix because the input matrix has to be further divided into 2×2 blocks, which still requires $O(n^3)$ running time. In the present work, we attempt to overcome this shortcoming by using Strassen's matrix multiplication algorithm, which was proposed by Volker Strassen in 1969 [8]. Strassen's algorithm only needs 7 block matrix multiplications for the 2×2 splitting of matrices, thus resulting in a time complexity of $O(n^{2.807})$. An interesting research question is whether this gain in complexity translates to gains in actual wall clock execution time on reasonably sized matrices when implemented using a Big Data processing platform such as Spark.

- *C. Misra is with the Xavier University Bhubaneswar, Harirajpur, Odisha 752050, India. E-mail: chandan.misra1@gmail.com.*
- *S. Bhattacharya and S.K. Ghosh are with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India. E-mail: {sourangshu, skg}@cse.iitkgp.ac.in.*

Manuscript received 1 Feb. 2019; revised 19 Feb. 2020; accepted 24 Feb. 2020. Date of publication 2 Mar. 2020; date of current version 13 May 2022. (Corresponding author: Chandan Misra.)
Recommended for acceptance by J. Tang.
Digital Object Identifier no. 10.1109/TBDATA.2020.2977326

Strassen's algorithm is inherently recursive in nature and cannot be implemented efficiently in Hadoop MapReduce, because MapReduce supports only stateless distributed operations so that fault tolerance can be ensured. Hence, for maintaining distributed states in Hadoop, one has to resort to disk-based data structures in HFDS or use external distributed key-value stores such as zookeeper, parameter server, etc. On the other hand, Spark can make recursive calls in the methods of driver program, which permits launching distributed in-memory jobs. The distributed state information can be stored as tags in the in-memory distributed data structure, thus supporting a more natural and efficient implementation for distributed recursion. Moreover, Spark programs are part of the overall Hadoop ecosystem, and are interoperable with HDFS, Cassandra, HBase, Hive, etc. Hence, our distributed matrix multiplication scheme can be used as part of larger data analytics workflows, 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.

1.1 Challenges and Contribution

There are several research challenges in developing the distributed version of Strassen's matrix multiplication algorithm, in the Map-Reduce framework. First, Strassen's algorithm is recursive in nature and thus it is not directly suitable for the Map-Reduce framework, which essentially assumes stateless functions for fault-tolerance. Hence, careful bookkeeping is needed for maintaining the state information in the global parameters, *Resillient Distributed Datasets* (RDDs) in case of Spark. Moreover, the matrix is not easily partitionable i.e., each element in the product matrix depends on multiple elements in the input / intermediate matrices. Therefore, each partition cannot be processed independently which is one of the requirements for the MapReduce programming model. 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. Also, it is necessary to keep track of the sub-matrices and matrix blocks in the intermediate map and reduce phases, so that they can be further divided, or merged and eventually stored at their final position in the product matrix.

In this paper, we address the above challenge by developing a distributed tail recursive version of the Strassen's algorithm (Algorithm 2). Our algorithm intelligently labels 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. The distributed algorithm or Stark, is implemented using the Apache Spark framework. We performed extensive experimentation on wall clock execution time of Stark vis-à-vis the state-of-the-art implementations available on Spark (both *MLLib* and *Marlin*). We show that for a large range of matrix sizes, our implementation performs 16 – 25 percent better than the nearest competitor.

Second, even though Strassen's algorithm is theoretically faster, the tradeoffs between three key elements: *computation*, *communication* (I/O), and *parallelism* (the number of actions happening in parallel) determine whether an actual speedup in wall clock execution time will be observed. In order to arrive at a suitable tradeoff, careful theoretical analysis of different stages of execution for the distributed Strassen algorithm is needed in all three aspects. We address this challenge by conducting a comprehensive analysis of the *computation complexity*, *communication complexity*, and *parallelization factor* for the implemented Strassen's algorithm as well as the baseline methods mentioned above. A similar analysis was also done by Gu *et al.* [6]. However, they did not report individual results for the different processing stages. This is critical in our case since the number of stages depends on the size of the matrix.

Since we are interested in the wall clock execution time, we do the above analysis for each of the serially executed Spark execution stages. The wall clock time for each stage is determined by the dominant component (either computation or communication), and the parallelization factor which allows the total computation (or communication) for each stage to be divided into parallel executors. The total wall clock execution time is the sum of wall clock execution times of stages. We find that our theoretical execution time model is closely matched by the empirical observations of wall clock execution time when the implemented algorithms are run in a distributed setting. Hence, this analysis helps us to pinpoint the source of the improvement in wall clock execution time. We find that the component that dominates the overall program run time in all the competing systems is the leaf node block multiplication that is executed in separate executors in parallel and our system outperformed them in the number of multiplications performed in leaf nodes. While *MLLib* and *Marlin* require s^3 ($s =$ number of splits) multiplications, our system needs only $s^{\log 7}$ multiplications.

Through theoretical analysis and experimental validation we find that the optimal block size for a given matrix size can be ascertained by two factors. First, the running time curve follows a U shape as a function of number of splits or block size, hence providing the user with a guideline for block size resulting in minimal running time. Second, the size of the RDD for storing the intermediate data increases at a rate of $3^l n^2$, where the depth of recursion l is determined by the block size. The user may require this to be less than total executor memory, thus avoiding disk operations.

Experimental Validation. We empirically demonstrate the effectiveness of our algorithm by comparing it to the best performing baselines over a suitable number of splits (typically between 2 to 32), for various matrix sizes. We find that our method requires 16 and 25 percent less wall clock time than *Marlin* and *MLLib* respectively. In another experiment, we vary the number of splits for each matrix size. We find that running time follows a U-shaped pattern, thus suggesting an optimal block size for each matrix size. This experiment also shows that theoretically calculated running times and empirically observed running times match closely, hence further validating our theoretical calculations. Finally, we report stage-wise breakdown of both theoretical running times and empirically observed running times. This helps us to identify the most time-consuming stage, thus re-affirming our conclusion regarding the reason for improvement of running time with Strassen's algorithm over existing baselines.

This paper is organised in five sections. 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* from Section 3.2 and provide a detailed description of the algorithm along with the data structure used. In Section 4, we evaluate the performance of our algorithm vis-a-vis two other competing approaches — *MLLib* and *Marlin* — showing that *Stark* performs better than others. The performance analysis also provides an explanation for the superior empirical performance of Stark in Section 5. Section 6 summarizes the results and discusses the future research direction.

2 Related Work

An extensive literature exists on parallelizing naïve matrix multiplication algorithms [5], [9], [10], [11], and [12]. Similarly Strassen's matrix multiplication algorithm has also been extensively studied for parallelization [13], [14], [15], [16] and [17]. The literature on parallel and distributed matrix multiplication can be divided broadly into three categories: 1) Grid based approaches, 2) Breadth First Search/ Depth First Search (BFS/DFS) based approaches and 3) Hadoop and Spark based approaches [6], [12], [16]. In our review, we will also adopt this schema.

2.1 Grid Based Approach

The grid-based approaches are particularly well suited for the processor layouts built in a two or three-dimensional grid. Grid based parallel matrix multiplication approaches are classified as $2D$, $2.5D$, and $3D$. The most common known $2D$ algorithm is Cannon's algorithm [18] and *SUMMA* [9]. $3D$ approaches [19], [20], move less data than the $2D$ algorithms by storing redundant copies of the matrices. $2.5D$ approaches in [11], [21] and [22], have been developed to interpolate between $2D$ and $3D$ approaches to attain better scaling and more efficient $2D$ matrix multiplication.

Grid based Strassen's matrix multiplication has also gone through a similar evolution and got new $2D$ and $2.5D$ approaches. Luo and Drake [14] provided two Strassen's based parallel approaches — *2D-Strassen* which uses a classical parallel matrix multiply approach for breaking the matrices and Strassen's multiply method to multiply leaf node matrices and *Strassen-2D* where Strassen's is used at the higher levels and classical parallel matrix multiplication is used at lower levels. $2.5D$ approaches [21] provide better communication efficiency than their $2D$ counterparts but still lack communication optimality (see Section 2.2). Grid-based algorithms are very efficient in a grid and torus-based topologies but may not perform well in other more general topologies [12], which is the main focus of generic Big Data computing platforms.

2.2 BFS/DFS Based Approach

BFS/DFS approaches are developed [17] for Strassen's algorithm to achieve communication optimality in distributed memory parallel architectures. Among Strassen-based parallel algorithms, Communication-Optimal Parallel Strassen's (CAPS [16]) minimizes communication costs and runs fastest in practice. Ballard *et al.* [12] determined the communication costs for Strassen methods and also provided the communication lower bound for square as well as for rectangular matrices, which proves that CAPS matches the lower bound and, therefore, provides communication optimality [10], [17].

At each level of the recursion tree of the Strassen's algorithm, *CAPS* uses either a BFS or a DFS step to traverse the tree in parallel. BFS requires more memory but incurs less communication cost as it processes all the intermediate subproblems among processors in parallel. On the other hand, DFS is more communication intensive as it processes each subproblem sequentially using all the processors. In an *Unlimited Memory* (UM) scheme, *CAPS* can employ BFS for all the levels to reach communication optimality. In a *Limited Memory* scheme, a number of DFS steps are executed serially before using all BFS steps till the leaf nodes and it is found to be optimal up to a constant factor. Though our implementation follows a similar kind of recursion tree as the *CAPS UM* scheme, it is worth evaluating the algorithm in a scalable framework where data is distributed.

2.3 Hadoop and Spark Based Approach

There are several implementations of distributed matrix multiplication using Hadoop MapReduce and Spark. John Norstad [23] presented four strategies to implement data parallel matrix multiplication using a block matrix data structure. However, all of them require 8 block multiplications to calculate the product matrix when the input matrix is further divided into 2×2 blocks and thus require $O(n^3)$ running time.

There are other distributed frameworks like *HAMA* [5] and *MadLINQ* [24], that provide massive matrix computation. Matrix multiplication in *HAMA* is carried out using two approaches — *iterative* and *Block*. In the iterative approach, each map task receives a row index of the right matrix as a key and the column vector of the row as a value. Then it multiplies all columns of row i of the left matrix with the received column vector. Finally, a reduce task collects the ith product into the result matrix. The block approach reduces the required data movement over the network by building a collection table and placing the candidate block matrix in each row. The iterative approach is not suitable in Hadoop for massive communication cost. Although *Block* approach incurs low communication cost, it does not provide faster execution as it uses require 8 block multiplications to calculate the product matrix. *MadLINQ*, built on top of Microsoft's *LINQ* framework and *Dryad* [25], are the examples of cloud-based linear algebra platform. However, they suffers from same kind of drawback as *HAMA*. Deng and Ramanan [26], [27] claimed that Strassen's algorithm can be implemented using six MapReduce passes, but did not provide the algorithm. Hence, we are not able to experimentally evaluate it.

Yu *et al.* proposed in his paper [28] an auto tuning approach which can speed up Spark programs by a factor of 30.4x on average and up to 89x. However, our present work is motivated by the challenges in implementing a distributed block recursive algorithm and its theoretical analysis and thus, is not a suitable competing approach. Rong Gu *et al.* in [6] developed an efficient distributed computation library, called *Marlin*, on top of Apache Spark. They proposed three different matrix multiplication approaches (*Block Splitting*, *CARMA*, and *Broadcast*) based on the size of input matrices and have shown that Marlin is faster than R and distributed algorithms based on Hadoop MapReduce. In the *Block Splitting* approach, used for $m \times m$ shaped square matrices, each

block of left matrix and right matrix are replicated m times. This ensures candidate blocks to be multiplied in parallel for a particular product block and requires shuffling candidate blocks to be in the same partition. The authors employed Spark's *join* transformation, which minimizes the communication cost by shuffling only one matrix. However, it still requires 8 block multiplications compared to 7 multiplications in Stark, which makes Stark faster than Marlin. The Spark MLLib library suffers from the similar shortcomings of 8 block multiplications. However, the algorithm first lists all the partitions 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. First, we describe the original Strassen's algorithm for serial matrix multiplication in Section 3.1. Next, we describe the *block* data structure, which is central to our distributed matrix multiplication algorithm, since it encapsulates both the contents of a block as well as *tags* necessary for the distributed recursive algorithm in Section 3.2. Finally, Section 3.3 describes the distributed matrix multiplication algorithm, and its implementation strategy using RDDs [29] of blocks.

3.1 Single Node Strassen's Preliminaries

Strassen's matrix multiplication, as shown in Algorithm 1, can multiply two $(n \times n)$ matrices using 7 multiplications and 18 additions of matrix blocks of size $\frac{n}{2} \times \frac{n}{2}$, thus providing much faster execution compared to 8 multiplications and 4 additions of the naïve algorithm. Note that, n should be 2^p for some integer p. However, the scheme can also be applied to rectangular matrices or matrices of general sizes by partitioning them appropriately as demonstrated by [14]. In this paper, we focus on matrices of size 2^p for mathematical brevity.

Algorithm 1. Strassen's Matrix Multiplication

Procedure `Strassen's` (A, B, *threshold*)
A = input matrix of size $n \times n$;
B = input matrix of size $n \times n$;
C = output matrix of size $n \times n$;

if *n=thresold* **then**
Multiply A and B using naïve approach;
else
Compute $A_{11}, B_{11}, \ldots, A_{22}, B_{22}$ by computing $n = \frac{n}{2}$;
$M_1 = \text{Strassen's}((A_{11} + A_{22}), (B_{11} + B_{22}))$;
$M_2 = \text{Strassen's}((A_{21} + A_{22}), B_{11})$;
$M_3 = \text{Strassen's}(A_{11}, (B_{12} - B_{22}))$;
$M_4 = \text{Strassen's}(A_{22}, (B_{21} - B_{11}))$;
$M_5 = \text{Strassen's}((A_{11} + A_{12}), B_{22})$;
$M_6 = \text{Strassen's}((A_{21} - A_{11}), (B_{11} + B_{12}))$;
$M_7 = \text{Strassen's}((A_{12} - A_{22}), (B_{21} + B_{22}))$;
$C_{11} = (M_1 + M_4 - M_5 + M_7)$;
$C_{12} = (M_3 + M_5)$;
$C_{21} = (M_2 + M_4)$;
$C_{22} = (M_1 - M_2 - M_3 + M_6)$;
end
return C;

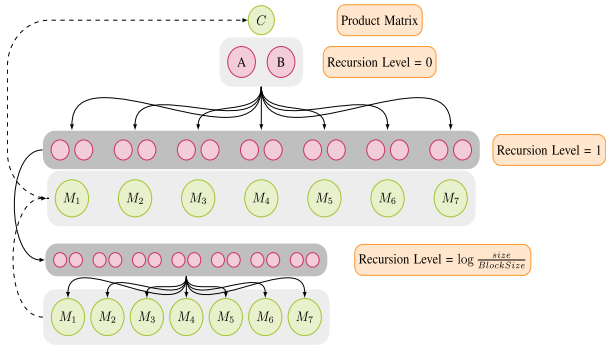

Fig. 1. The implementation flow of Stark. Dark rectangles with red circles denote RDD produced during the division and replication phase. Light rectangles with green circles denote the RDD resulted from the combination phase of the recursion algorithm. Solid and dashed lines specifies the top-down and bottom-up flow respectively. Each recursion level (from 0 to $\log(size/BlockSize)$) is executed in parallel.

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) bookkeeping information needed for running the algorithm.

Conceptually, each matrix of dimension n, is partitioned into s number of splits, giving $\frac{n}{s}$ *block rows* and $\frac{n}{s}$ *block columns*. Here, *block rows* and *block columns* are defined by the number of rows and columns of blocks. Each matrix of size n is divided into four equal square sub-matrices of dimension $\frac{n}{2}$, until it reaches *block* dimension of $b = \frac{n}{s}$. 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.

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 the algorithm, these indices can change to keep track of the current position of sub-matrix.
2) *column-index*: Analogously to the above, stores the current column index of the sub-matrix.
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. 1. 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 runs in parallel across several Spark executors inside the cluster and is described in the following sections.

Algorithm 2. Distributed Strassen's Matrix Multiplication

Procedure `DistStrass` ($RDD < Block > A$, $RDD < Block > B$, *blockSize*)
  C = product matrix $A \times B$ as an RDD of Blocks
  $size \leftarrow$ size of matrix A or B
  $blockSize \leftarrow$ size of a single matrix block
  $n \leftarrow \frac{size}{blockSize}$
  if $n = 1$ **then**
    `MulBlockMat` (*A*,*B*);
  else
    $n \leftarrow \frac{n}{2}$
    $[A_1, B_1] \leftarrow$ `DivNRep` (*A*,*B*)
    $R \leftarrow$ `DistStrass` (A_1, B_1,*n*)
    $C \leftarrow$ `Combine` (*R*)
  end
  return C;

3.3.1 Divide and Replication Phase

In the divide step, the matrices are divided into 4 sub-matrices of equal size, designated as $A_{11}, A_{12}, A_{21}, A_{22}$. These sub-matrices are needed to create seven intermediate matrices M_1 to M_7, as shown in Algorithm 1. Note that, we need 4 copies of A_{11} and A_{22} and 2 copies of A_{12} and A_{21}, hence 12 sub-matrices of size 2^{n-1}. Matrix B is divided similarly. The replication is performed using *flatMapToPair* transformation. Each sub-matrix is marked with an appropriate M-Index (M1 – M7), according to the destination where it is needed for computation. For example, the 4 A_{11}s are marked with $M1, M3, M5, M6$. Additionally, since we need to store the destination over all the recursion levels, the destination index grows exponentially with recursion level. At every recursion level, the destination index of a sub-matrix is calculated as: $parentIndex * 7 + index$. For example, after 2 recursive splits, a sub-matrix with destination index, $parentIndex = 32$ will generate sub-matrices with destination indices 224 – 230. Each sub-matrix generated after the divide and replicate stage contains the key (M-index,destination-index). This is stored in a $PairRDD$ as shown in Algorithm 3. Next we used spark $groupByKey$ operation to collect all necessary sub-matrices, and perform the necessary computations given in Algorithm 1. The multiplication operation is performed by recusively calling $DistStrass$ routine in Algorithm 2.

3.3.2 Multiplication of Block Index Matrices

When the division reaches the size of a user-defined block size, the candidate blocks are aggregated inside a single RDD. This is done using one *mapToPair*, followed by one *groupByKey* as shown in Algorithm 4. The *mapToPair* function takes each block and returns a *key-value* pair to group two potential blocks for multiplication. The *groupByKey* actually groups candidate blocks for multiplication followed by a map function which returns the product of the candidate blocks in the form of an RDD. The keys in the *mapToPair* transformation are chosen in such a way so that all the leaf level candidate blocks are multiplied in parallel whereas each such candidate blocks are multiplied serially inside individual nodes. To speed up the individual matrix multiplications, we transform the leaf node block matrices into Breeze [30] matrices, which permits using hardware optimized BLAS implementations.

Algorithm 3. Divide and Replication

Procedure `DivNRep` ($RDD < Block > A$, $RDD < Block > B$)
  Result: $RDD < Block > C$
  $RDD < Block > AunionB$ = *A*.union(*B*);
  $PairRDD < string, Block > firstMap$ = $AunionB$.flatMapToPair();
  $PairRDD < string, iterable < Block > > group$ = $firstMap$.groupByKey();
  $RDD < Block > C$ = $group$.mapToPair();
  return C;

Algorithm 4. Block Matrix Multiplication

Procedure `MulBlockMat` ($RDD < Block > A$, $RDD < Block > B$)
  Result: $RDD < Block > C$
  $RDD < Block > AunionB$ = *A*.union(*B*);
  $PairRDD < string, Block > firstMap$ = $AunionB$.mapToPair();
  $PairRDD < string, iterable < Block > > group$ = $firstMap$.groupByKey();
  $RDD < Block > C$ = $group$.map();
  return C;

3.3.3 Combining the Sub-Matrices

The combine step consists of three transformations - *mapToPair*, *groupByKey*, and *flatMap*. *MapToPair* associates each input block a key which will represent the block indexes of its immediate parent level of the recursion tree. Since each parent sub-matrix generates 7 child sub-matrices in the divide step, the key of each block in this transformation is divided by 7. *groupByKey* groups all the similar keys together to form parent sub-matrices from children. For example, blocks of 49 children sub-matrices in level 3 are grouped to form blocks of 7 parent sub-matrices in level 2. Next, each of the sub-matrix is transformed into the 4 matrices C_1 to C_4 of the same size using the *flatmap* step. Thus, the flatMap takes a list of blocks and transforms them into a list of respectively 4 sub-matrices (C_1 to C_4). It takes a list of blocks with same key and transforms into a list of blocks that denotes 4 sub-matrices C_1 to C_4. All the Block level additions and subtractions (refer to last 4 lines of the else part of Algorithm 1) are done in parallel, similar to block level multiplications, while each such additions and subtractions are done locally inside the nodes. At the last step, we make a union of 4 sub-matrices to get the product matrix C. The *combine* step is shown in Algorithm 5.

TABLE 1
Stagewise Performance Analysis of MLLib

Stage-Step	Computation	Communication	P.F.
S1-flatMap	s^3	NA	$min[s^2, c]$
S1-flatMap	s^3	NA	$min[s^2, c]$
S3-co-Group	NA	$2min[s, c]n^2$	$min[s^2, c]$
S3-flatMap	$s^3 \times (\frac{n}{s})^3$	NA	$min[s^2, c]$
S4-reduceByKey	sn^2	NA	$min[s^2, c]$

TABLE 2
Stagewise Cost Analysis of Marlin

Stage-Step	Computation	Communication	P.F.
S1-flatMap	$2s^3$	$2sn^2$	$min[2s^2, c]$
S1-flatMap	$2s^3$	$2sn^2$	$min[2s^2, c]$
S3-Join	NA	sn^2	$min[s^3, c]$
S3-mapPartition	$s^3 \times (\frac{n}{s})^3$	bs^2	$min[s^3, c]$
S4-reduceByKey	NA	sn^2	$min[s^2, c]$

This concludes the description of the distributed matrix multiplication algorithm, *Stark*. Next, we evaluate the performance of our algorithm from a series of experiments and compare it with the baselines.

Algorithm 5. Combine Phase

Procedure `Combine` ($RDD < Block > BlockRDD$)
Result: $RDD < Block > C$
$PairRDD < String, Block > firstMap$ = $BlockRDD$.map();
$PairRDD < string, iterable < Block > > group$ = $firstMap$.groupByKey();
$RDD < Block > C$ = $group$.flatMap();
return C;

4 Performance Analysis of the Competing Approaches

In this section, we attempt to estimate the performance of the proposed approach *Stark*, and state-of-the-art approaches *MLLib* and *Marlin*, for distributed matrix multiplication. In this work, we are interested in the *wall clock execution time* of the algorithms for a varying number of nodes, matrix sizes, and other algorithmic parameters e.g., partition/block sizes. The wall clock execution time depends on three independently analysed quantities: total *computational complexity* of the sub-tasks to be executed (stages in case of Spark), total *communication complexity* between executors of different sub-tasks on each of the nodes, and *parallelization factor* (PF) of each of the sub-tasks or the total number of processor cores available. Gu *et al*. [6] also follow a similar paradigm for analysis of *Marlin*.

We consider only square matrices of dimension 2^p for all of the derivations. The key input and tunable parameters for the algorithms are:

- $n = 2^p$: number of rows or columns in matrix A and B (for square matrix)
- s = number of splits
- $2^q = b = \frac{n}{s}$ = block size in matrix A and B (used for Strassen's multiplication cost analysis)
- c = Total number of physical cores in the cluster

Therefore,

- Total number of blocks in matrix A or $B = s^2$
- $s = 2^{p-q}$

The following cost analysis has been done conforming to the Spark execution model, which constitutes of two main abstractions — *RDD* and *Lineage Graph* (which is a Direct Acyclic Graph (DAG) of operations). RDDs are a collection of elements partitioned across the nodes of a cluster that can be operated on in parallel and stored either in memory or in Hadoop / Hadoop supported file system. RDDs support two types of operations — *transformations* and *actions*. A Spark program implicitly creates a *lineage*, which is a logical DAG of transformations that resulted in the RDDs. When the driver runs, it converts this logical graph into a physical *execution plan* with a set of *stages* by pipelining the transformations. Then, it creates smaller execution units, referred to as *tasks* under each stage, which are bundled up and prepared to be sent to the cluster. For brevity we provide only the computation and communication complexities and parallization factors for each stage for the three approaches. Details of the derivation of the cost analysis has been provided in the supplementary document, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TBDATA.2020.2977326. Our analysis follows the execution plan for the programs developed and compared for this paper. This allows us to directly correlate actual wall clock running times with theoretically predicted ones, hence pinpointing the stages which run faster. *Marlin* and *MLLib* job execution consists of 6 stages. However, the first four stages are part of the preprocessing stage except two flatMap transformations. Therefore, the two flatMap steps and Stage 3 and 4 are part of the actual execution. A summary of the cost analysis of *MLLib* and *Marlin* is given in Tables 1 and 2 respectively.

Unlike *MLLib* and *Marlin*, *Stark* does not posses a constant number of stages. Instead, the number of execution stages depends on the recursion depth. 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 equal to $2(p - q) + 2$. We divide the algorithm into three main sections - *Divide*, *Multiply* and *Combine*. Each such section consists of three transformations and their corresponding costs are detailed in Table 3.

5 Experiments

In this section, we report results from experiments performed to evaluate the performance of *Stark*, and compare it with that of *Marlin* and *MLLib*. First, we provide the test setup in Section 5.1. Then, in Section 5.2, we compare the fastest possible wall clock time, of the three algorithms for different input matrix sizes. After that, in Section 5.3, we conduct a series of experiments to individually evaluate the effect of partition size and the matrix size of each competing approaches. At last, we evaluate the scalability of *Stark* in Section 5.6. All the equations referred in this section are detailed in the supplemental material, available online.

TABLE 3
Stagewise Cost Analysis of Stark

Stage-Step	Computation	Communication	P.F.
S_1 to S_{p-q} - flatMap Divide	$\frac{8s^2}{3}(s^{0.8}-1)$	NA	$min[\frac{7^i}{4}(2s^2), c]$
S_2 to S_{p-q+1} - groupByKey Divide	NA	$\frac{12n^2}{5}(s^{1.8}-1)$	$min[7^{i+1}, c]$
Stage 2 to Stage $p-q+1$ - flatMap Divide	$\frac{7n^2}{3}(s^{0.8}-1)$	NA	$min[7^{i+1}, c]$
Stage $p-q+1$ - map Leaf	$2s^{2.8}$	NA	$min[s^{2.8}, c]$
Stage $p-q+2$ - groupByKey Leaf	NA	$2s^{0.8}n^2$	$min[s^{2.8}, c]$
Stage $p-q+2$ - flatMap Leaf	$s^{2.8} \times (\frac{n}{s})^3$	NA	$min[s^{2.8}, c]$
Stage $p-q+2$ to Stage $2p-2q+1$ - map Combine	$\frac{7s^2}{3}(s^{0.8}-1)$	NA	$min[7^{i+1}, c]$
Stage $p-q+3$ to Stage $2p-2q+2$ - groupByKey Combine	NA	$\frac{7n^2}{3}(s^{0.8}-1)$	$min[7^{i+1}, c]$
Stage $p-q+3$ to Stage $2p-2q+2$ - flatMap Combine	$\frac{14n^2}{s^2}(s^{2.8}-1)$	NA	$min[7^{i+1}, c]$

5.1 Test Setup

All the experiments are carried out on a dedicated cluster of 3 nodes except in case of comparisons with single node systems (Section 5.2) and scalability test (Section 5.6), where a cluster of 7 nodes has been used. Software and hardware specifications are summarized in Table 4. As previously mentioned, for block-level multiplications *Stark* uses Breeze, a hardware optimized BLAS implementation to execute the linear algebra computation on a single node. The algorithms were benchmarked with matrices of increasing cardinality starting from 16×16 and up to 16384×16384. The input matrices were initialized by randomly sampling 32 bit real values (IEEE 754 floating point format) from a normal distribution. To be able to obtain the execution time in a more precise manner, we cache the input matrices first and then employ the count action on it resulting the lazy executor to store the matrix RDDs into the memory of all the nodes and execute the RDD stages one by one. We then use Java's system time to make a note of the start and finish time before and after the actual multiplication function. This allows us to execute all the computations without any intervention and provide a reasonable execution time. The reported wall-clock times for each experiment represent the arithmetic average across 10 independent runs.

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. Swapping and task failures, which require the partial reevaluation of the DAG, can drastically alter the overall execution times. To allow a fair comparison among the competing approaches, we ensured that the individual jobs do not experience thrashing and always selected parameters that provide good utilization of cluster resources, but mitigate the chance of task failures. Through experimentation, we found that restricting the memory of Spark executors to 50 GB ensures successful execution (i.e., no *out of memory* errors) of jobs for all competing approaches. *Stark* requires $3^l N^2$ space for input matrix of size N and recursion level l. To multiply large matrices of size 16384×16384 with a recursion depth of 5 (split size = 32 or block size = 512), it requires up to 243 GB memory. In this case the input data to memory size ratio is almost 1 (i.e., 250 GB or 5×50 GB per executor).

TABLE 4
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
Scheduler	NA	YARN
Java	NA	1.7.0 update 79

5.2 Practical Efficiency of Stark

In this section, we demonstrate the practical utility of *Stark* compare to distributed systems as well as single node optimized matrix multiplication approaches.

a) *Comparison with state-of-the-art distributed systems:* In this experiment, we examine the performance of Stark with other Spark based distributed matrix multiplication approaches i.e., *Marlin* and *MLLib*. We report the running time of the competing approaches with increasing matrix size. We take the best wall clock time (fastest) among all the running time taken for different block sizes. The experimental results are shown in Fig. 2. We observe that *Stark* takes time comparable to MLLib and Marlin for matrix dimensions, 4096 and 8192, but is faster for matrix size 16384. *MLLib* takes most time for matrix dimension 16384. Also, as expected the wall clock execution time increases with the matrix dimension, non-linearly (roughly as $O(n^{2.9})$). Also, the gap in wall clock execution time between both *Stark* and *Marlin*, as well as *Marlin* and

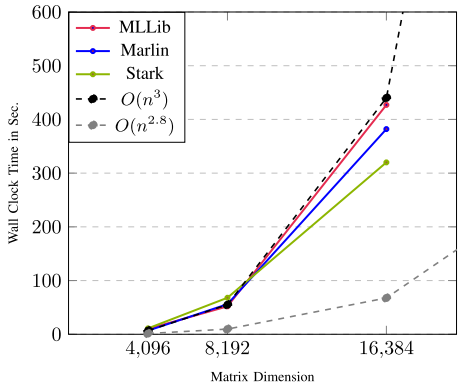

Fig. 2. Fastest running time of three systems among different block sizes. Fastest running time occurs when the number of splits is 1. 8, 16 and 32 for MLLib, 2. 4, 8, and 16 for Marlin, and 3. 4, 8, and 16 for Stark. for matrix size 4096, 8192, and 16384 respectively.

TABLE 5
Performance Comparison Among 6 Systems With Increasing Matrix Sizes (The Unit of Wall Clock Time is Second)

Matrix	SN	SS	Colt	JBlas	Intel MKL	Stark 140 cores
512×512	< 1	< 1	< 1	< 1	< 1	5
1024×1024	15	2	1	< 1	< 1	6
2048×2048	177	14	13	2	< 1	9
4096×4096	2112	100	135	16	< 1	3
8192×8192	NA	394	1163	119	5	12
16384×16384	NA	2453	NA	862	40	66
32768×32768	NA	NA	NA	NA	343	365

Here SN means Serial naïve and SS means Serial Strassen's.

MLLib increases monotonically with input matrix dimension.

b) *Comparison with state-of-the-art single node systems* : The second experiment (Table 5) examines how the runtime improves if we use Stark on the cluster compared to the matrix multiplication on a single node having similar configuration of a single node in the cluster. We report the fastest wall clock execution times for each of the methods and matrix sizes. The intention of this experiment is to demonstrate the performance of Stark compared to highly optimized linear algebra libraries, e.g., Colt [31], JBlas [32], and Intel Math Kernel Library (MKL) [33]. We use the parallel version of Colt library, named ParallelColt [34] which employs automatic multi-threading when computations are done on a machine with multiple CPUs. We also report execution times two other variations of the single node serial matrix multiplication algorithm: the three loop naïve approach and the single node Strassen's matrix multiplication algorithm. We use *NA* when the wall clock time is more than reasonable time i.e., more than 1 hour. The results also show that for matrix sizes larger than (8192×8192), *Stark*'s performance is comparable to Intel MKL (which is a native library), and better than all other java-based single node algorithms.

While the comparison here is not fair, since Stark uses much more resources, we show that there is a substantial gain in wall clock execution time, over the single node parallel options currently available, thus justifying a distributed solution.

5.3 Variation With the Number of Splits

In this experiment, we examine the performance of *Stark* with *Marlin* and *MLLib* with increasing number of splits for each matrix size. For each matrix size (from (4096×4096) to (16384×16384) we increase the number of splits until we get an intuitive change in the results as shown in Fig. 3.

We see that *Stark* takes the minimum amount of wall clock time among all the approaches for almost all the values of the number of splits and for all the matrix sizes. The costliest part of all the competing approaches is *Stage 3*, which contains the shuffle and leaf node block matrix multiplication steps. It can be easily verified that the computation cost of *Stark* (as shown in Equation (32) in the supplemental document, available online) is less than *MLLib* (Equation (5)) and *Marlin* (Equation (16)). This is studied in further detail in Section 5.4.

We see that *Stark* takes more time than *MLLib* at $s = 16$ and $s = 32$ and matrix size 4096×4096 and 8192×8192. As the divide section cost at *Stage 2* of *Stark* depends on s, increasing its value increases the number of times divide section executes which eventually adds communication cost. This suggests that too many partitions for a small matrix will increase the wall clock execution time.

It can be seen that all the approaches follow a *U* shaped curve i.e., there exists an optimal value for s, where the wall clock time required to perform the matrix multiplication is minimal. If the number of splits s is either too low or too large the wall clock running time increases (a discussed in detail in Section 5.4).

The gap between *MLLib* and *Stark* decreases as s increases. On the other hand, after an optimal point the *Stark* line overshoots but *MLLib* got more gradual tendency. This is because, there is a trade-off between the computation cost and communication cost of *Stark*, as we increase the number of splits. For a smaller number of partitions, the computation cost is comparatively higher, while communication cost is low due to short height of the recursion tree. On the other hand, large values of the number of splits increase communication cost for each level of the recursion tree, without gaining much from parallelization. This suggests that unrolling the recursion to an appropriate depth will result in an optimal gap in performance.

5.4 Comparison Between Theoretical and Experimental Results

In this experiment, we compare the theoretical cost of all the approaches with the experimental wall clock execution time to validate our theoretical cost analysis. Fig. 4 shows the comparison for three matrix sizes (from (4096×4096) to (16384×16384)) and for each matrix size as the number of splits increases.

As expected, both the theoretical and experimental wall clock execution time for all the three approaches shows a *U* shaped curve as the number of splits increases. The reason

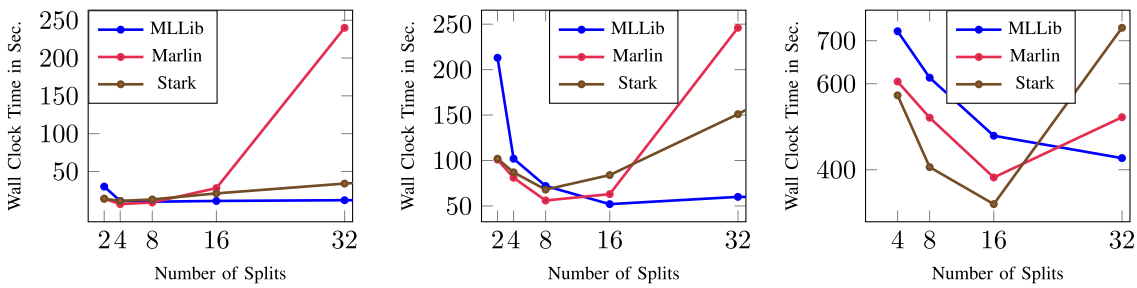

Fig. 3. Comparing running time of MLLib, Marlin, and Stark for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing number of splits. The minimum wall clock execution times for the methods correspond to the fastest running time as shown in Fig. 2.

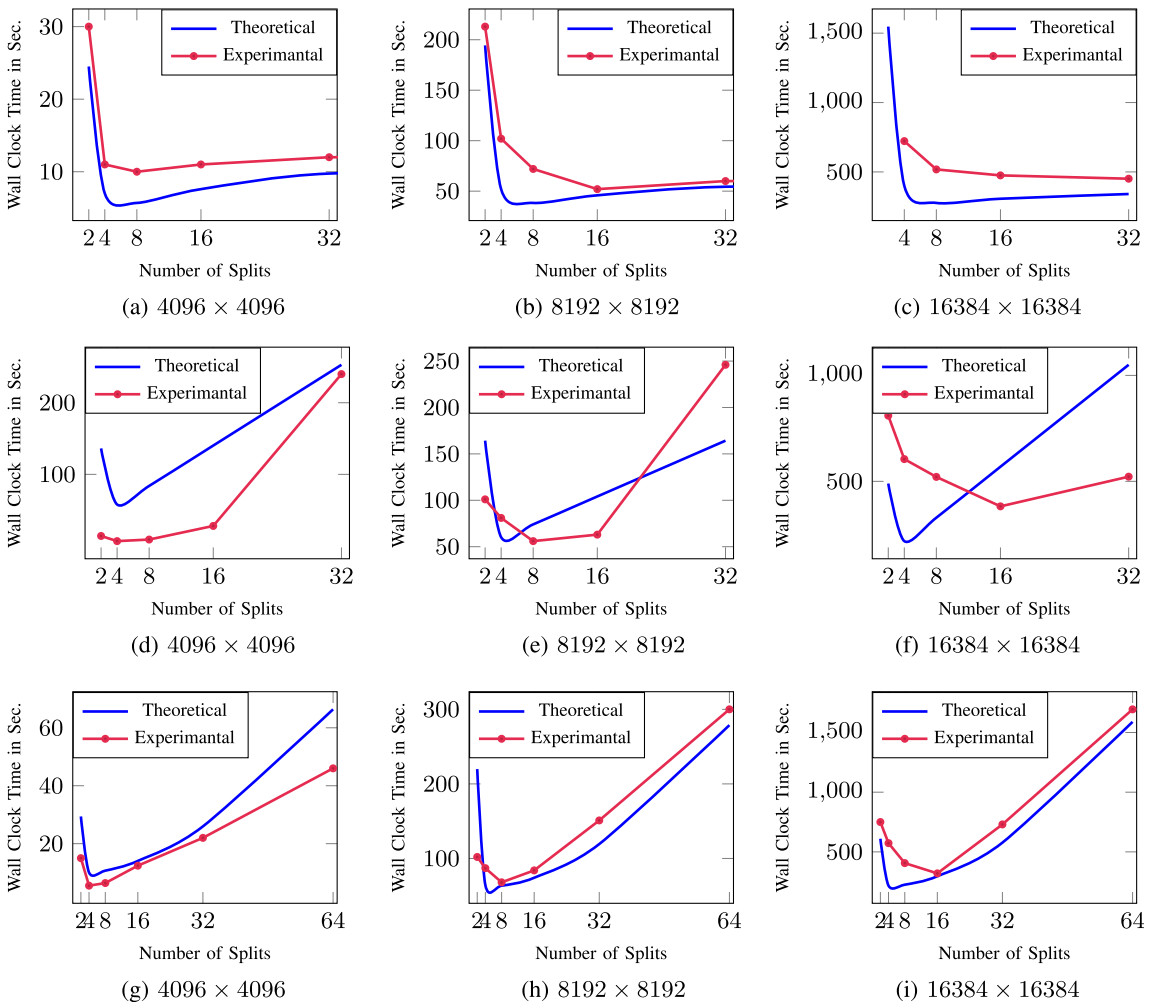

Fig. 4. Comparing theoretical and experimental running time of *MLLib* (a), (b) and (c), *Marlin* (d), (e) and (f) and *Stark* (g), (h) and (i) for matrix size (4096×4096), (8192×8192), (16384×16384) for increasing number of splits.

is that the size of matrix blocks becomes very large when it is split into a smaller number of splits for very large matrices. As a result, the single node matrix multiplication execution time is found to be large at the beginning which results in overall large execution time. As the number of splits increases, the parallelization factor increases until it reaches the actual number of physical cores available in the cluster. As a result, the overall cost decreases gradually. After the optimal point is reached, the computation cost stabilizes and communication cost starts to increase as the factor sn^2 for *MLLib* and *Marlin* and $\sum_{i=0}^{p-q-1}(7/4)^i(2s^2)$ for *Stark*, increases.

From Fig. 4 we observe that the minimum actual time and minimum theoretical time occur at different values of the number of splits. For example in *MLLib* for matrix size 8192×8192, the minimum theoretical time occurs at number of splits equals 8, whereas the experimental minimum occurs when the number of splits is 16. In order to explain this discrepancy, we measure the computation times for leaf-node block matrix multiplication only (reported in Table 6). These costs are extracted by caching matrix blocks to be multiplied at the leaf nodes into the executor memory and calculating the execution time just for the transformations used for leaf node block multiplications. For brevity, we report times only for *Marlin* and *Stark* as *MLLib* follows similar pattern as *Marlin*. Green and red cells mark the minimum of theoretical and experimental computation times divided by parallelization factor, respectively.

As observed in Fig. 4, the experimental running time minimum comes later than the theoretical one. We observe that the shifts in computation cost minimum (Table 6) roughly correspond to the shifts in overall execution time

TABLE 6
Observed Leaf-Node Block Matrix Multiplication Cost (in Milliseconds) for Marlin and Stark

Matrix Size: 4096 × 4096					
Method	Number of Partitions				
	2	4	8	16	32
Marlin	10225	6578	5059	7012	15598
Stark	12533	4715	3724	3368	3123
Matrix Size: 8192 × 8192					
Method	Number of Partitions				
	2	4	8	16	32
Marlin	93293	63050	42201	34777	42191
Stark	99185	64715	31348	23945	35662
Matrix Size: 16384 × 16384					
Method	Number of Partitions				
	2	4	8	16	32
Marlin	681401	433659	325698	335291	413648
Stark	680896	412706	246895	175346	186747

Green denotes theoretical minimum, Red denote experimental minimum.

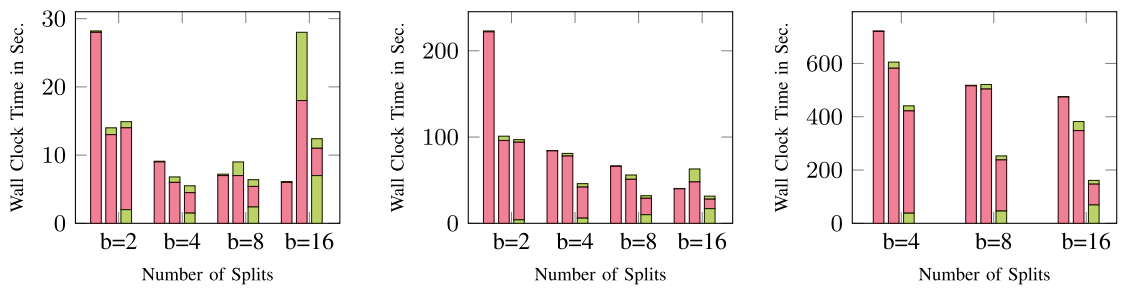

Fig. 5. Comparing step-wise running time among MLLib, Marlin,. and Stark for matrix size (4096×4096), (8192×8192), (16384×16384) and for increasing number of splits. Here red bar denotes the cost incurred during leaf node multiplication step and green bar signifies the cost incurred during divide and combine step. The cost signifies the dominant cost for each step i.e., computation cost for multiplication step and communication cost for divide and combine step.

(Fig. 4). For example, for *Stark* the minimum overall computation cost can be achieved if number of splits is 16. Since we use the Breeze library for multiplication of matrix blocks, we attribute this discrepancy to internal optimizations of the Breeze package. Hence, we conclude that modulo this discrepancy, the theoretical and experimental wall clock times match, thus justifying our analysis.

5.5 Stage-Wise Comparison

In this experiment, we compare the running time of three approaches stage-wise and thereby infer the time-consuming stage of each approach. We perform this test for increasing matrix sizes and for each matrix size with a varying number of splits. As the number of stages of *Stark* grows on the order of $p - q$, when there is a large difference between matrix size and the number of splits, the value of $p - q$ becomes too large to compare with other approaches. For this reason, we merge the stages of *Stark* to form 3 stages, comprising stages related to divide, leaf node multiplication and combine phases. Fig. 5 shows the comparison.

It is clearly seen that, *Stage 3* is the costliest stage for *MLLib* and *Marlin*. *Stage 3* in *MLLib* consists of *coGroup* and *flatMap* transformations, while in *Marlin* it is *partitionBy*, *join* and *mapPartitions* transformations. These contribute to the replication, shuffling and multiplications of blocks resulting cost far more than *Stage 4* having only one transformation *reduceByKey*.

For *Stark*, on the other hand, the most expensive stage changes as we increase the number of splits. For smaller number of splits *Stage 3* or leaf node multiplication computation cost dominates while for larger partitions *Stage 2* or communication cost of matrix division dominates. This is because, the communication cost corresponding to the divide section dominates as s is increased and as the height of the recursion tree increases, the communication cost accumulates and surpasses the computation cost of leaf node multiplications. 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. Tables 7, 8 and 9 also clarify that the computation cost in *Stage 3* is arguably closer for smaller number of splits than when we move to a larger number of splits

TABLE 8
Stagewise Performance Comparison Among Three Systems With Increasing Partition (The Unit of Execution Time is Second) for Matrix Size 8192×8192

Stage	Wall Clock Time											
	b=2			b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S	M1	M2	S
S_1	108 96	3 3	29	25 29	2 2	15	21 23	3 2	12 -	16 15	5 3	12
S_2	- - - -	- - - -	4 - - -	- - - -	- - - -	4 2 - -	- - - -	- - - -	4 3 3 -	- - - -	- - - -	4 3 5 5
S_3	222	96	90	84	78	66	66	51	35	40	48	26
S_4	1 - - -	5 - - -	3 - - -	0.4 - - -	3 - - -	1 3 - -	0.5 - - -	5 - - -	1 1 1 -	0.4 - - -	15 - - -	1 0.8 0.7 1

Here S_1 to S_4 corresponds to Stage 1 to Stage 4.

TABLE 7
Stagewise Performance Comparison Among Three Systems With Increasing Partition (The Unit of Execution Time is Second) for Matrix Size 4096×4096

Stage	Wall Clock Time											
	b=2			b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S	M1	M2	S
Stage 1	4 4	1 1	8	5 5	1 1	4	1 2	1 0.9	4	2 2	2 2	4
Stage 2	- - - -	- - - -	2 - - -	- - - -	- - - -	1 0.5 - -	- - - -	- - - -	1 0.7 0.7 -	- - - -	- - - -	1 1 2 3
Stage 3	28	13	12	9	6	7	7	7	6	6	18	6
Stage 4	0.2 - - -	1 - - -	0.9 - - -	0.1 - - -	0.8 - - -	0.4 0.6 - -	0.2 - - -	2 - - -	0.3 0.3 0.4 -	0.1 - - -	10 - - -	0.6 0.3 0.2 0.3

TABLE 9
Stagewise Performance Comparison Among Three Systems With Increasing Partition (The Unit of Execution Time is Second) for Matrix Size 16384×16384

Stage	Wall Clock Time								
	b=4			b=8			b=16		
	M1	M2	S	M1	M2	S	M1	M2	S
Stage 1	150 156	9 6	90	49 42	14 12	48	33 34	43 48	43
Stage 2	- - - -	- - - -	29 9 - -	- - - -	- - - -	27 10 9 -	- - - -	- - - -	20 14 17 18
Stage 3	720	582	300	516	504	168	474	348	162
Stage 4	2 - - -	23 - - -	8 11 - -	2 - - -	17 - - -	3 7 5 -	1 - - -	34 - - -	4 2 3 5

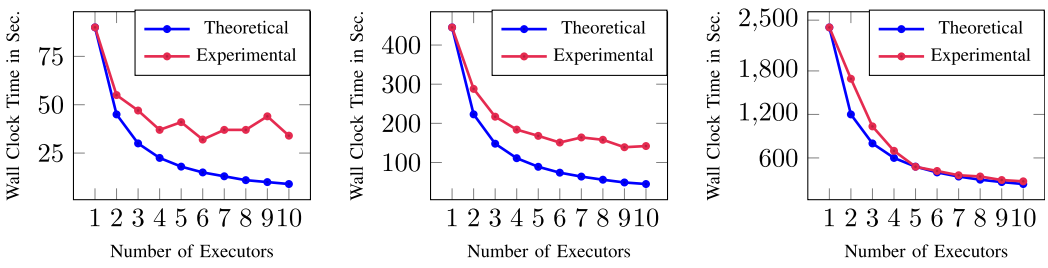

Fig. 6. The scalability of *Stark* , in comparison with theoretical scalability (blue line), on matrix (4096×4096), (8192×8192) and (16384×16384). The wall clock execution time is recorded against increasing number of executors (the number of cores is 5 for all the executors i.e., $1 - 10$. Here theoretical scalability is $T(1)/n$, where $T(1)$ is wall clock execution time when number of executors $= 1$ and n is total number of executors.

for *Marlin* and *Stark*. The reason is that, as we increase the number of splits s, the number of multiplications for *Marlin* to be carried out grows in s^3 order, while *Stark* grows in 7^{p-q} or $s^{2.8}$ order, which is less than the earlier.

5.6 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 versus the number of Spark executors for these 3 pairs of matrices is shown in Fig. 6. 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 strong scalability, with a minor deviation from ideal scalability when the size of the matrix is low (i.e., for (4096×4096)).

6 Conclusion

In this paper, we have focused on the problem of distributed matrix multiplication of large and distributed matrices using the Spark framework. Here, we have overcome the shortcomings in the state-of-the-art distributed matrix multiplication approaches requiring $O(n^3)$ running time. We have accomplished that by providing an efficient distributed implementation of the sub-cubic $O(n^{2.807})$ time, Strassen's multiplication algorithm. A key novelty is to simulate the distributed recursion by carefully tagging the matrix blocks and processing each level of the recursion tree in parallel.

We have also reported a comprehensive theoretical analysis of the computation and communication costs and a parallization factor associated with each stage of *Stark* as well as other baseline approaches. Through extensive experiments on the wall clock execution time of the competitive approaches we found that the theoretical analysis matches with the empirical one and also pinpoint the actual source of improvement in wall clock time of *Stark*.

An important drawback of the current approach is its high space complexity, which is $O(3^l N^2)$, where l is the recursion level (this is because at each recursion level the size of the data grows 3 times than the previous level). The Spark cluster consumes huge amount of physical memory for very large matrices and therefore cannot execute in situations when other approaches can be run smoothly. It will be interesting to find a way to circumvent this problem.

Acknowledgments

This paper is a part of the Ph.D. research work of Chandan Misra, being conducted at the Advanced Technology Development Centre, IIT Kharagpur, India. The authors would like to thank IIT Kharagpur for providing all the necessary support required to conduct the research.

References

[1] Apache hadoop project, Accessed: Dec. 23, 2016, 2016. [Online]. Available: https://hadoop.apache.org/
[2] Apache spark, lightning-fast cluster computing, Accessed: Dec. 23, 2016, 2016. [Online]. Available: https://spark.apache.org/
[3] X. Meng *et al.*, "MLlib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 34, pp. 1–7, 2016.
[4] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 2.
[5] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the MapReduce framework," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 721–726.
[6] R. Gu *et al.*, "Efficient large scale distributed matrix computation with spark," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 2327–2336.
[7] R. Gu *et al.*, "Improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2539–2552, Sep. 2017.
[8] V. Strassen, "Gaussian elimination is not optimal," *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.
[9] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency-Practice Experience*, vol. 9, no. 4, pp. 255–274, 1997.
[10] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Graph expansion and communication costs of fast matrix multiplication," *J. ACM*, vol. 59, no. 6, 2012, Art. no. 32.
[11] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms," in *Proc. Eur. Conf. Parallel Process.*, 2011, pp. 90–109.
[12] J. Demmel *et al.*, "Communication-optimal parallel recursive rectangular matrix multiplication," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 261–272.
[13] B. Grayson and R. Van De Geijn, "A high performance parallel strassen implementation," *Parallel Process. Lett.*, vol. 6, no. 01, pp. 3–12, 1996.
[14] Q. Luo and J. B. Drake, "A scalable parallel strassen's matrix multiplication algorithm for distributed-memory computers," in *Proc. ACM Symp. Appl. Comput.*, 1995, pp. 221–226.
[15] W. F. McColl and A. Tiskin, "Memory-efficient matrix multiplication in the BSP model," *Algorithmica*, vol. 24, no. 3, pp. 287–297, 1999.
[16] B. Lipshitz, G. Ballard, J. Demmel, and O. Schwartz, "Communication-avoiding parallel strassen: Implementation and performance," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 101.
[17] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for strassen's matrix multiplication," in *Proc. 24th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2012, pp. 193–204.

[18] L. E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Doctoral dissertation, Montana State University-Bozeman, College of Engineering, 1969.
[19] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM J. Res. Develop.*, vol. 39, no. 5, pp. 575–582, 1995.
[20] J. Berntsen, "Communication efficient matrix multiplication on hypercubes," *Parallel Comput.*, vol. 12, no. 3, pp. 335–342, 1989.
[21] E. Solomonik, A. Bhatele, and J. Demmel, "Improving communication performance in dense linear algebra via topology aware collectives," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 77.
[22] W. F. McColl, "A BSP realisation of strassen's algorithm," Oxford Univ. Comput. Lab., Tech. Rep., 1995.
[23] J. Norstad, "A MapReduce algorithm for matrix multiplication," Accessed: Dec. 23, 2016, 2009. [Online]. Available: http://norstad.org/matrix-multiply/index.html
[24] Z. Qian *et al.*, "MadLINQ: Large-scale distributed matrix computation for the cloud," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 197–210.
[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.
[26] M. Deng and P. Ramanan, "MapReduce implementation of strassen's algorithm for matrix multiplication," in *Proc. 4th ACM SIGMOD Workshop Algorithms Syst. MapReduce Beyond*, 2017, Art. no. 7.
[27] P. Ramanan, "Six pass MapReduce implementation of strassen's algorithm for matrix multiplication," in *Proc. 5th ACM SIGMOD Workshop Algorithms Syst. MapReduce Beyond*, 2018, Art. no. 7.
[28] Z. Yu, Z. Bei, and X. Qian, "Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 564–577, 2018.
[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010, Art. no. 95.
[30] GitHub - Scalanlp/breeze, Accessed: Jan. 28, 2020, 2020. [Online]. Available: https://github.com/scalanlp/breeze
[31] Colt - Welcome, Accessed: Apr. 6, 2017, 2017. [Online]. Available: https://dst.lbl.gov/ACSSoftware/colt/
[32] Linear Algebra for Java, Accessed: Apr. 6, 2017, 2017. [Online]. Available: http://jblas.org/
[33] Intel Math Kernel Library, Accessed: Jan. 28, 2020, 2020. [Online]. Available: https://software.intel.com/en-us/mkl
[34] P. Wendykier and J. G. Nagy, "Parallel colt: A high-performance java library for scientific computing and image processing," *ACM Trans. Math. Softw.*, vol. 37, no. 3, 2010, Art. no. 31.

Chandan Misra received the BTech degree in information technology from the West Bengal University of Technology, West Bengal, India, and the MS degree in computer science from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India. He is currently working toward the PhD degree in Advanced Technology Development Centre, IIT Kharagpur, Kharagpur, India. He is currently serving as a lecturer with Xavier University Bhubaneswar. His research interests include big data, and distributed linear algebra.

Sourangshu Bhattacharya is currently an assistant professor with the Department of Computer Science and Engineering, IIT Kharagpur. He was a scientist with Yahoo! Labs from 2008 to 2013, where he was working on prediction of Click-through rates, Ad-targeting to customers, etc. on the Rightmedia display ads exchange. He was a visiting scholar with the Helsinki University of Technology from January - May 2008. His current research interests include applications of machine learning in opinion dynamics in online social networks, online advertising, information extraction, and transportation science; as well as in distributed machine learning.

Soumya K. Ghosh received the MTech and PhD degrees in computer science and engineering from the Indian Institute of Technology (IIT) Kharagpur, Kharagpur, India. He is currently a professor with the Department of Computer Science and Engineering, IIT Kharagpur. Before joining IIT Kharagpur, he was with the Indian Space Research Organization, working in the area of satellite remote sensing and geographic information system (GIS). His research interests include geoscience and spatial web services, mobile GIS, spatial information retrieval, and knowledge discovery.