VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Topic:

Strassen's Matrix Multiplication using BlockMatrix in pyspark

Name:

Kush Desai(20MCB1002)

Raginee Titar(20MCB1004)

Shweta Shewale(20MCB1005)

Under the Guidance of:

Prof. Ramesh Ragala

Contents

ABSTRACT:

We will be working on Strassen’s matrix multiplication method. It is faster than the standard matrix multiplication algorithm and is useful in practice for large matrices, but would be slower than the fastest known algorithms for extremely large matrices. We will implement Strassen’s matrix multiplication for square matrix using pySpark.

INTRODUCTION:

Many existing works have implemented distributed matrix multiplication on Big data frameworks. One of the early works was , 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, which intelligently selects one of their three matrix multiplication algorithms according to the size of the input t matrices. However, both these schemes used naive distributed block matrix multiplication approach. This approach requires 8 block multiplications to calculate the product matrix when the input matrix is further divided in (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 ma - tix multiplication algorithm, which was proposed by Volke r Strassen in 1969. 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.

Strassen's algorithm, which is inherently recursive in nature, cannot be implemented efficiently in Hadoop MapReduce. This is because the MapReduce programming paradigm only supports 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. Spark is a natural choice since it can make recursive calls in the methods of driver program which can launch distributed in-memory jobs. The distributed state information can be stored as tag s 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, hence interoperable with HDFS, Cassan- JOURNAL OF LATEX CLASS FILES, VOL. 14, NO. 8, AUGUST 2015 2 dra, 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.

There are several research challenges in developing the distributed version of Strassen's matrix multiplication algorithm, in the Map-Reduce framework:

• Strassen's algorithm is recursive in nature and thus 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, redundant distributed datasets (RDDs) in case of Spark.

• 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 MapReduce programming model.

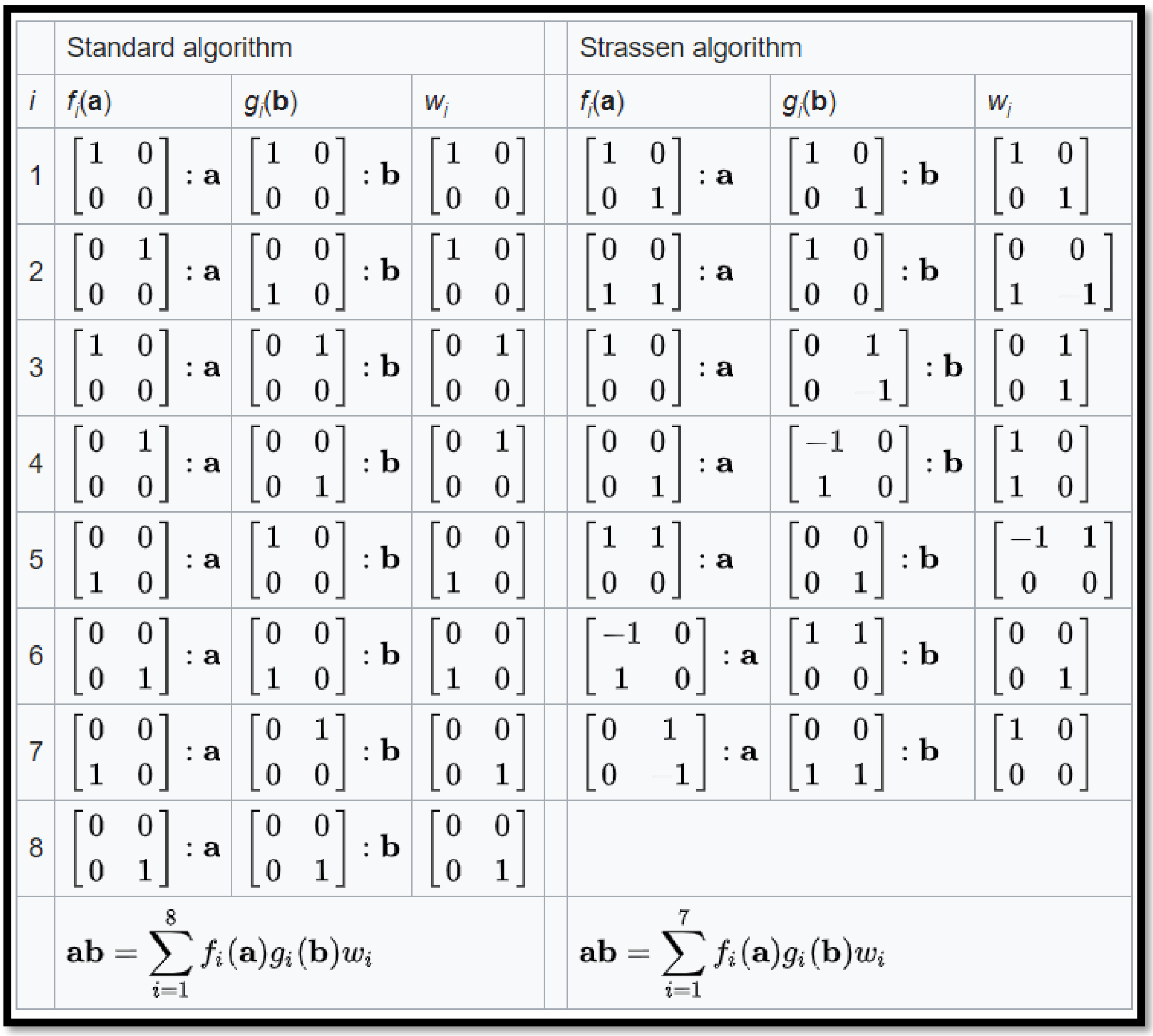
• Even though Strassen's algorithm is theoretically faster, the trade-offs between three key elements: computation, communication (I/O), and parallelism (the number of actions happening parallelly) determine whether an actual speedup in wall clock time will be observed. In order to arrive at a suitable trade off, careful theoretical analysis of different stage of execution for the distributed Strassen algorithm is needed in all three aspects.

PROBLEM STATEMENT:

Strassen’s Matrix Multiplication using Block Matrix in pyspark

PROPOSED SOLUTION:

The rank of a bilinear map is the length of its shortest bilinear computation.[5] The existence of Strassen's algorithm shows that the rank of 2×2 matrix multiplication is no more than seven. To see this, let us express this algorithm (alongside the standard algorithm) as such a bilinear computation. In the case of matrices, the dual spaces A* and B* consist of maps into the field F induced by a scalar double-dot product.



	Standard algorithm			Strassen algorithm		
i	$f_i(\mathbf{a})$	$g_i(\mathbf{b})$	w_i	$f_i(\mathbf{a})$	$g_i(\mathbf{b})$	w_i
1	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}$	$\begin{bmatrix}1&0\\0&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}1&0\\0&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$
2	$\begin{bmatrix}0&1\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\1&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}$	$\begin{bmatrix}0&0\\1&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\1&1\end{bmatrix}$
3	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&1\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&1\\0&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&1\\0&1\end{bmatrix}$
4	$\begin{bmatrix}0&1\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}-1&0\\1&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}1&0\\1&0\end{bmatrix}$
5	$\begin{bmatrix}0&0\\1&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\1&0\end{bmatrix}$	$\begin{bmatrix}1&1\\0&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}-1&1\\0&0\end{bmatrix}$
6	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\1&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\1&0\end{bmatrix}$	$\begin{bmatrix}-1&0\\1&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}1&1\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}$
7	$\begin{bmatrix}0&0\\1&0\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&1\\0&0\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}$	$\begin{bmatrix}0&1\\0&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\1&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}1&0\\0&0\end{bmatrix}$
8	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{a}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}:\mathbf{b}$	$\begin{bmatrix}0&0\\0&1\end{bmatrix}$			
	$\mathbf{ab}=\sum_{i=1}^{8} f_i(\mathbf{a})g_i(\mathbf{b})w_i$			$\mathbf{ab}=\sum_{i=1}^{7} f_i(\mathbf{a})g_i(\mathbf{b})w_i$		

It can be shown that the total number of elementary multiplications L required for matrix multiplication is tightly asymptotically bound to the rank R, i.e. , or more specifically, since the constants are known, One useful property of the rank is that it is sub multiplicative for tensor products, and this enables one to show that $2^n \times 2^n \times 2^n$ matrix multiplication can be accomplished with no more than 7^n elementary multiplications for any n.

METHODS USED:

<u>NumPy:</u>

NumPy is a python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices. NumPy stands for Numerical Python. In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists. The array object in NumPy is called Nd array, it provides a lot of supporting functions that make working with Nd array very easy. Arrays are very frequently used in data science, where speed and resources are very important.

<u>Pyspark:</u>

PySpark is a Python API for Spark released by the Apache Spark community to support Python with Spark. Using PySpark, one can easily integrate and work with RDDs in Python programming language too. There are numerous features that make PySpark such an amazing framework when it comes to working with huge datasets.

<u>pyspark.mllib.linalg.distributed module:</u>

Represents a distributed matrix in blocks of local matrices.

Parameters:

•blocks – An RDD of sub-matrix blocks ((blockRowIndex, blockColIndex), sub-matrix) that form this distributed matrix. If multiple blocks with the same index exist, the results for operations like add and multiply will be unpredictable.

•rowsPerBlock – Number of rows that make up each block. The blocks forming the final rows are not required to have the given number of rows.

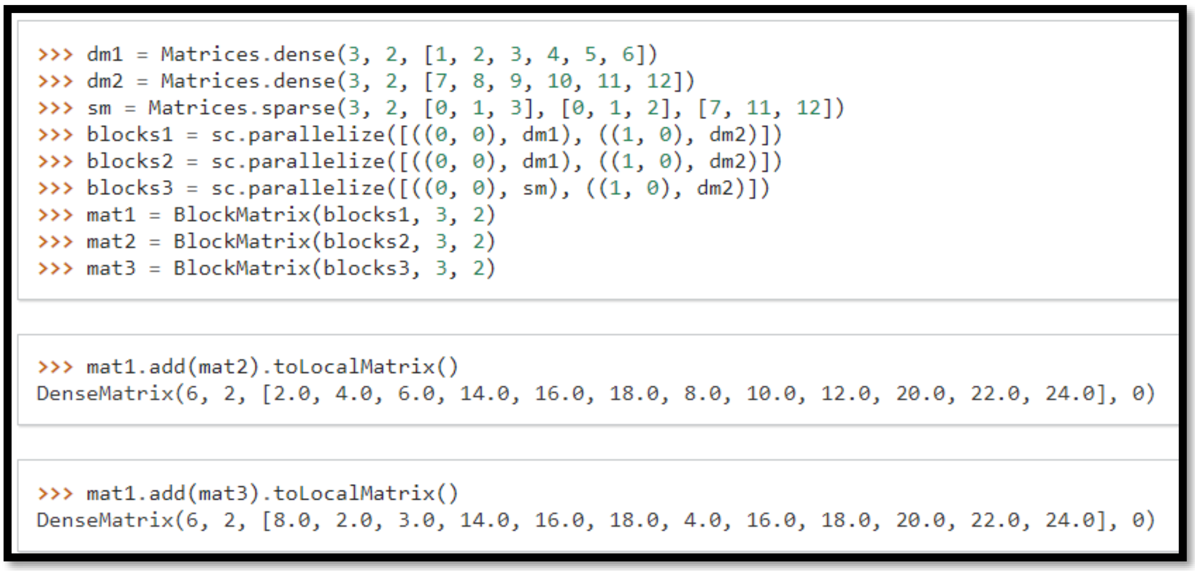
•colsPerBlock – Number of columns that make up each block. The blocks forming the final columns are not required to have the given number of columns.

•numRows – Number of rows of this matrix. If the supplied value is less than or equal to zero, the number of rows will be calculated when numRows is invoked.

•numCols – Number of columns of this matrix. If the supplied value is less than or equal to zero, the number of columns will be calculated when numCols is invoked.

<u>add(*other*):</u>

Adds two block matrices together. The matrices must have the same size and matching rowsPerBlock and colsPerBlock values. If one of the sub matrix blocks that are being added is a SparseMatrix, the resulting sub matrix block will also be a SparseMatrix, even if it is being added to a DenseMatrix. If two dense sub matrix blocks are added, the output block will also be a DenseMatrix.



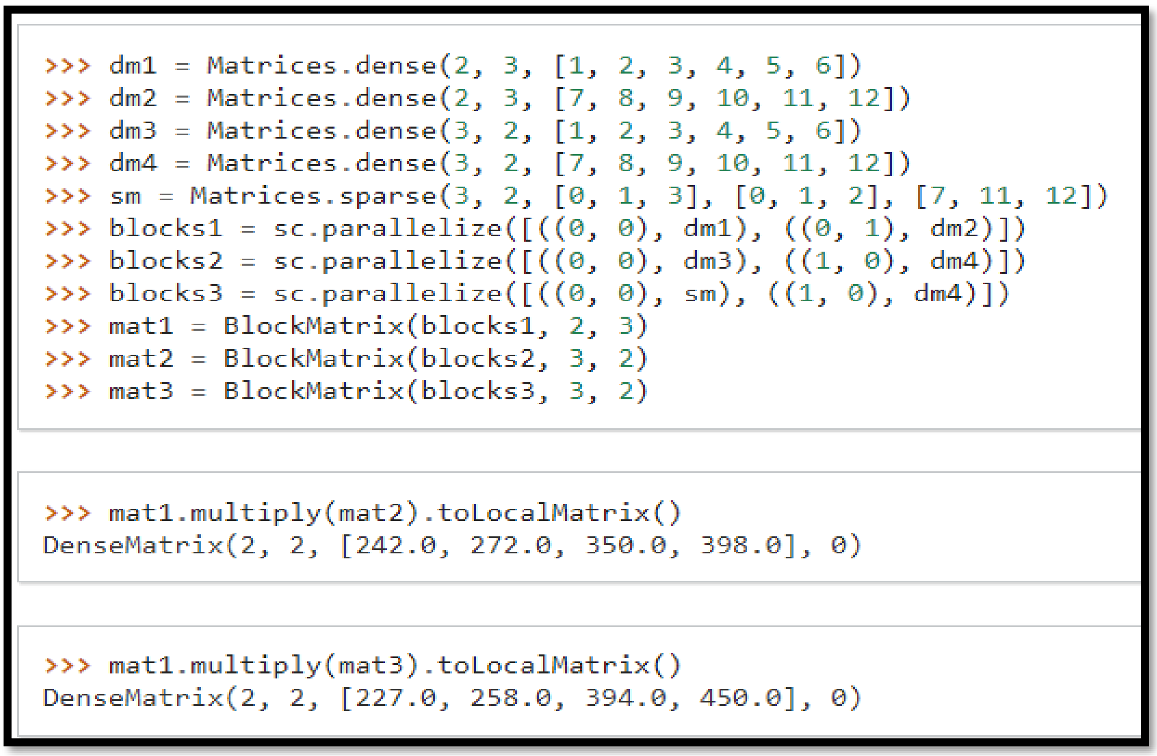
```
>>> dm1 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
>>> dm2 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [7, 11, 12])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm2)])
>>> mat1 = BlockMatrix(blocks1, 3, 2)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.add(mat2).toLocalMatrix()
DenseMatrix(6, 2, [2.0, 4.0, 6.0, 14.0, 16.0, 18.0, 8.0, 10.0, 12.0, 20.0, 22.0, 24.0], 0)
```

```
>>> mat1.add(mat3).toLocalMatrix()
DenseMatrix(6, 2, [8.0, 2.0, 3.0, 14.0, 16.0, 18.0, 4.0, 16.0, 18.0, 20.0, 22.0, 24.0], 0)
```

multiply(*other*):

Left multiplies this BlockMatrix by other, another BlockMatrix. The colsPerBlock of this matrix must equal the rowsPerBlock of other. If other contains any SparseMatrix blocks, they will have to be converted to DenseMatrix blocks. The output BlockMatrix will only consist of DenseMatrix blocks. This may cause some performance issues until support for multiplying two sparse matrices is added.



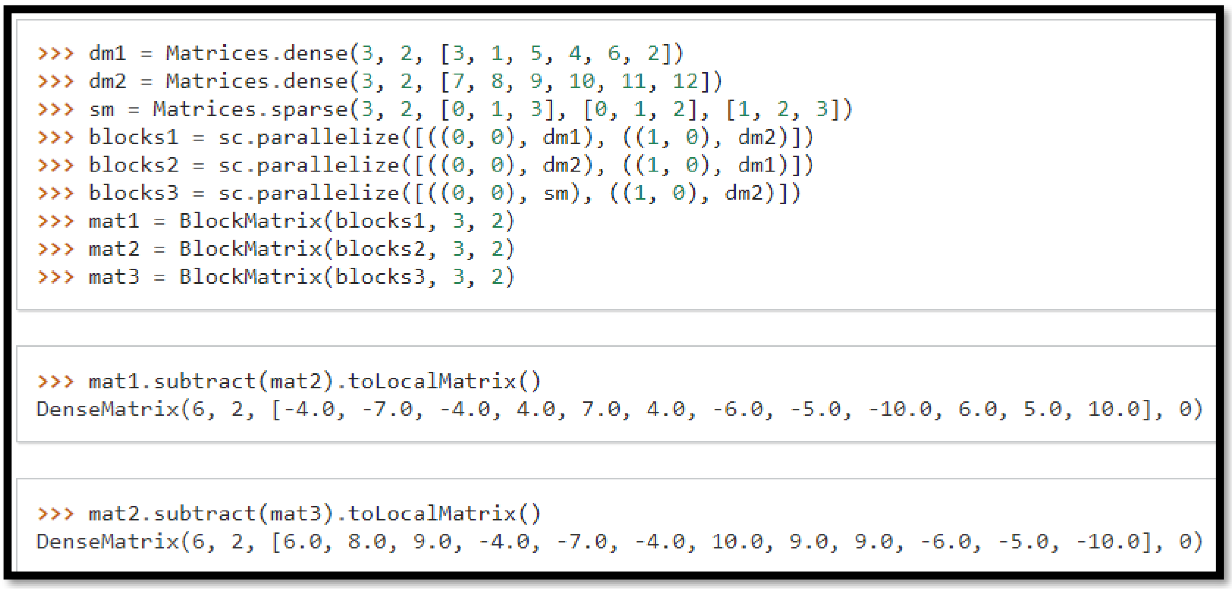
```
>>> dm1 = Matrices.dense(2, 3, [1, 2, 3, 4, 5, 6])
>>> dm2 = Matrices.dense(2, 3, [7, 8, 9, 10, 11, 12])
>>> dm3 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
>>> dm4 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [7, 11, 12])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((0, 1), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm3), ((1, 0), dm4)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm4)])
>>> mat1 = BlockMatrix(blocks1, 2, 3)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.multiply(mat2).toLocalMatrix()
DenseMatrix(2, 2, [242.0, 272.0, 350.0, 398.0], 0)
```

```
>>> mat1.multiply(mat3).toLocalMatrix()
DenseMatrix(2, 2, [227.0, 258.0, 394.0, 450.0], 0)
```

subtract(*other*):

Subtracts the given block matrix other from this block matrix: this - other. The matrices must have the same size & matching rowsPerBlock and colsPerBlock values. If one of the sub matrix blocks that are being subtracted is a SparseMatrix, the resulting sub matrix block will also be a SparseMatrix, even if it is being subtracted from a DenseMatrix. If two dense sub matrix blocks are subtracted, the output block will also be a DenseMatrix.



```
>>> dm1 = Matrices.dense(3, 2, [3, 1, 5, 4, 6, 2])
>>> dm2 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [1, 2, 3])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm2), ((1, 0), dm1)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm2)])
>>> mat1 = BlockMatrix(blocks1, 3, 2)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.subtract(mat2).toLocalMatrix()
DenseMatrix(6, 2, [-4.0, -7.0, -4.0, 4.0, 7.0, 4.0, -6.0, -5.0, -10.0, 6.0, 5.0, 10.0], 0)
```

```
>>> mat2.subtract(mat3).toLocalMatrix()
DenseMatrix(6, 2, [6.0, 8.0, 9.0, -4.0, -7.0, -4.0, 10.0, 9.0, 9.0, -6.0, -5.0, -10.0], 0)
```

Distributed matrix:

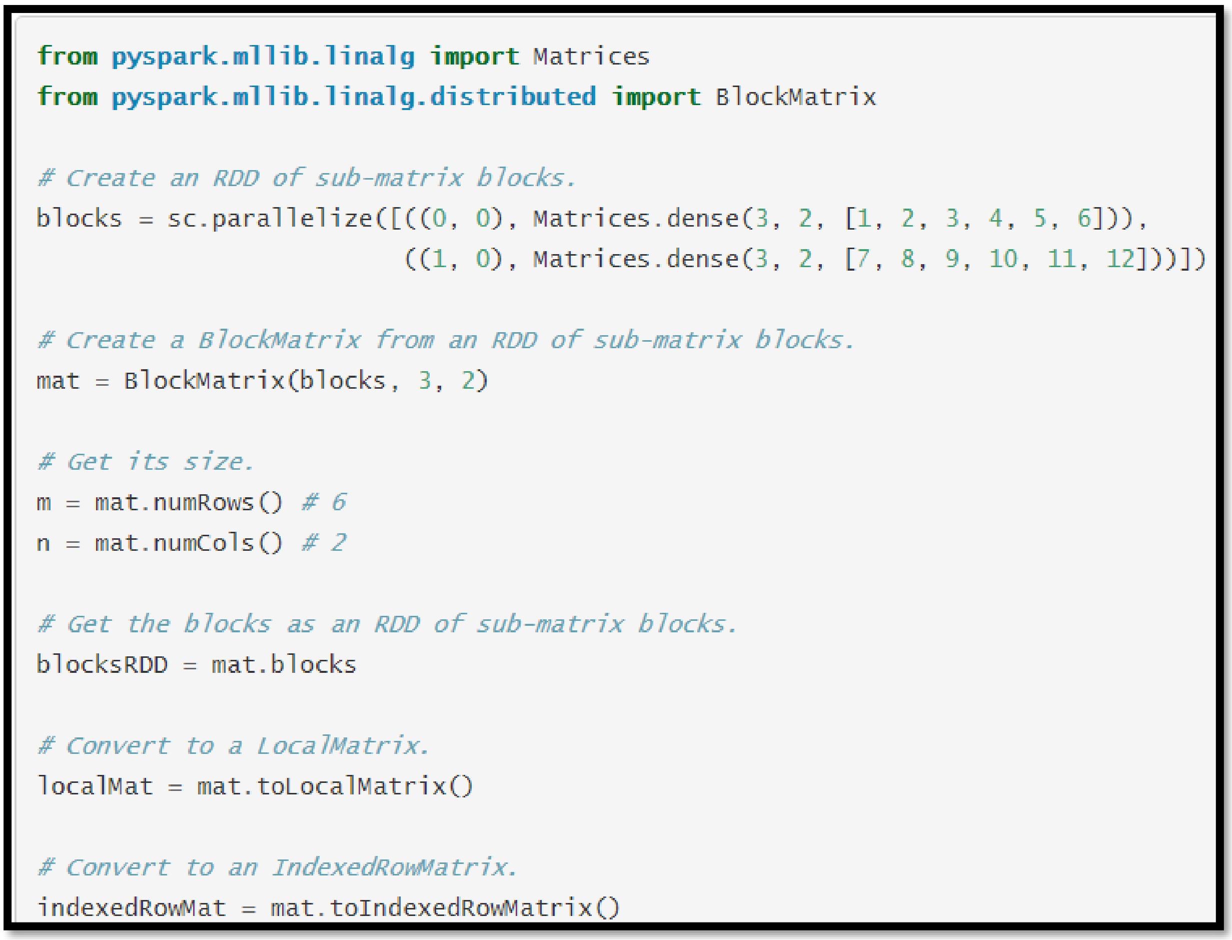
A distributed matrix has long-typed row and column indices and double-typed values, stored distributively in one or more RDDs. It is very important to choose the right format to store large and distributed matrices. Converting a distributed matrix to a different format may require a global shuffle, which is quite expensive. Three types of distributed matrices have been implemented so far.

The basic type is called RowMatrix. A RowMatrix is a row-oriented distributed matrix without meaningful row indices, e.g., a collection of feature vectors. It is backed by an RDD of its rows, where each row is a local vector. We assume that the number of columns is not huge for a RowMatrix so that a single local vector can be reasonably communicated to the driver and can also be stored / operated on using a single node. An IndexedRowMatrix is similar to a RowMatrix but with row indices, which can be used for identifying rows and executing joins. A CoordinateMatrix is a distributed matrix stored in coordinate list (COO) format, backed by an RDD of its entries.

Block Matrix:

A BlockMatrix is a distributed matrix backed by an RDD of MatrixBlocks, where a MatrixBlock is a tuple of ((Int, Int), Matrix), where the (Int, Int) is the index of the block, and Matrix is the sub-matrix at the given index with size rowsPerBlock x colsPerBlock. BlockMatrix supports methods such as add and multiply with another BlockMatrix. BlockMatrix also has a helper function validate which can be used to check whether the BlockMatrix is set up properly.

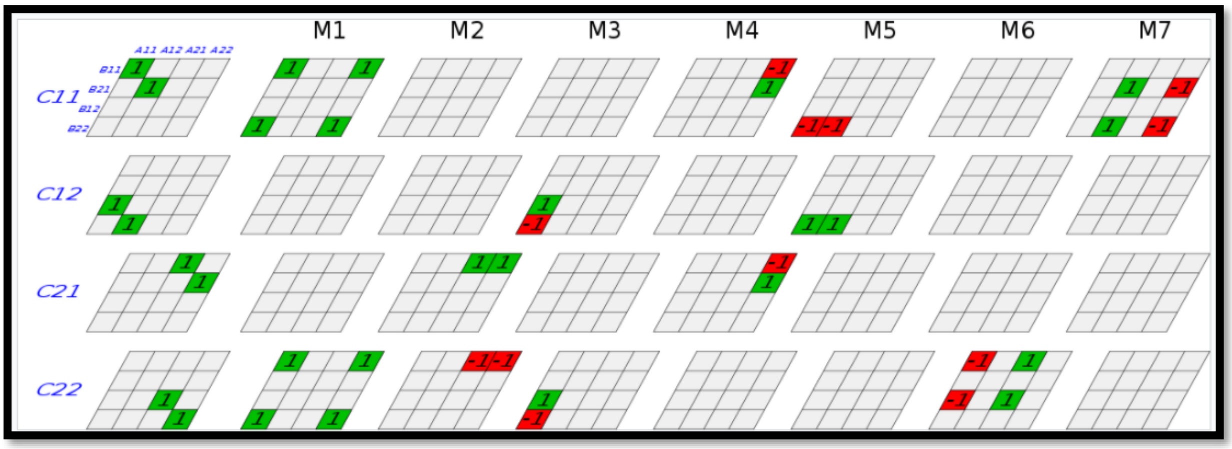
A BlockMatrix can be created from an RDD of sub-matrix blocks, where a sub-matrix block is a ((blockRowIndex, blockColIndex), sub-matrix) tuple.



```python
from pyspark.mllib.linalg import Matrices
from pyspark.mllib.linalg.distributed import BlockMatrix

# Create an RDD of sub-matrix blocks.
blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
                         ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])

# Create a BlockMatrix from an RDD of sub-matrix blocks.
mat = BlockMatrix(blocks, 3, 2)

# Get its size.
m = mat.numRows() # 6
n = mat.numCols() # 2

# Get the blocks as an RDD of sub-matrix blocks.
blocksRDD = mat.blocks

# Convert to a LocalMatrix.
localMat = mat.toLocalMatrix()

# Convert to an IndexedRowMatrix.
indexedRowMat = mat.toIndexedRowMatrix()
```

Algorithm Used:

Strassen's matrix:



Let *A*, *B* be two square matrices over a ring *R*. We want to calculate the matrix product *C* as

$$\mathbf{C} = \mathbf{A}\mathbf{B} \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{2^n \times 2^n}$$

If the matrices *A*, *B* are not of type $2^n \times 2^n$ we fill the missing rows and columns with zeros. We partition *A*, *B* and *C* into equally sized block matrices.

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

The naive algorithm would be:

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{aligned}$$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the $C_{i,j}$ matrices, the same number of multiplications we need when using standard matrix multiplication.

The Strassen algorithm defines instead new matrices:

$$
\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}
$$

only using 7 multiplications (one for each M_k) instead of 8. We may now express the $C_{i,j}$ in terms of M_k:

$$
\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
$$

We iterate this division process *n* times (recursively) until the submatrices degenerate into numbers (elements of the ring *R*). The resulting product will be padded with zeroes just like *A* and *B*, and should be stripped of the corresponding rows and columns.

Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which those algorithms are more efficient. The particular crossover point for which Strassen's algorithm is more efficient depends on the specific implementation and hardware. Earlier authors had estimated that Strassen's algorithm is faster for matrices with widths from 32 to 128 for optimized implementations.[1] However, it has been observed that this crossover point has been increasing in recent years, and a 2010 study found that even a single step of Strassen's algorithm is often not beneficial on current architectures, compared to a highly optimized traditional multiplication, until matrix sizes exceed 1000 or more, and even for matrix sizes of several thousand the benefit is typically marginal at best (around 10% or less).[2] A more recent study (2016) observed benefits for matrices as small as 512 and a benefit around 20%.

Conclusion:

we have focused on the problem of distributed matrix multiplication of large and distributed matrices using 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.

References:

1. https://spark.apache.org/docs/2.0.0-preview/mllib-data-types.html#local-matrix

2. https://spark.apache.org/docs/2.2.0/api/python/pyspark.mllib.html#pyspark.mllib.linalg.distributed.BlockMatrix

3. https://spark.apache.org/docs/2.0.0-preview/mllib-data-types.html#local-matrix

4. https://en.wikipedia.org/wiki/Strassen_algorithm

5. https://spark.apache.org/docs/2.2.0/api/python/_modules/pyspark/mllib/linalg/distributed.html