Distributed Computation(Spark) through Dense Matrix Multiplication for OLS-based Linear Regression.

Team Members: Komal Pardeshi and Sean Yu
Team Github: https://github.com/CS6240/project-cs6240project-sean-komal

Project Overview

Fast matrix multiplication and inversion are fundamental challenges in Computer Science due to their computational complexity. In the realm of big data, matrix operations are crucial for various tasks such as graph algorithms, machine learning, and network analysis. They find applications in tasks like hop-path calculations, dense matrix products, and similarity score predictions or calculating correlations for linear regression, among others. Similarly, matrix inversion is vital for solving systems of linear equations, parameter estimation in regression models, and optimization tasks. However, performing matrix inversion on large datasets poses significant challenges due to computational complexity and memory requirements of traditional inversion algorithms.

Linear regression is a fundamental statistical algorithm used to learn a linear relationship between a set of features and an output. While linear regression is simple and interpretable, it sometimes fails to prioritize significant predictors effectively. Hence, extensions like lasso regression and ridge regression have been developed as described in our supplementary material. Both linear regression and these extensions can be solved through matrix operations.

In this project, we aimed to compute Dense Matrix Multiplication and Inversion using Distributed Processing Algorithms on Spark using Scala. We carried out implementations for following tasks:

1. Dense Matrix Multiplication (H-V and V-H Partitioned Matrices): The Multiplication tasks is the major task we aim to optimize. We strictly utilize Shuffle-based strategies for matrices where the second matrix dimensions dominates over first, but our program examines the datasets to choose Broadcast where data could fit into memory.

2. Optimized Dense Matrix Multiplication for Symmetric Matrices: Correlation Matrices are alway symmetric and hence this strategy optimizes by reducing the computations to half by only calculating the lower triangular matrix.

3. Ordinary Least Squares Linear Regression: While we use decomposition methods for Inversion,which is comparatively difficult to parallelize, we achieve inherent parallelism in steps like computing Matrix Multiplcation as there are two operations for $A^T.A$ and the final $(A^T.A)^{-1}.B$

Originally, one of the tasks was to do matrix inversion, and it was submitted in the intermediate report. However after testing on bigger data on AWS there was some inconsistencies. Therefore, a linear algebra package called 'breeze' was used. The instruction to run our code are available on the github readme file.

The inputs, outputs and logs to all the experiments can be found at this one drive link.

Input Data

For the intermediate report both real and synthetic data were considered. H-V runs were conducted on the synthetic dataset generated and real data (WEC Perth 49). The data is further explained below.

1. Two programs were written in Python to create the synthetic data. One generates a random matrix for the purposes of testing matrix multiplication. The other generates a random symmetric positive definite matrix to test matrix inversion. This code is uploaded to GitHub at the following link. The data used to run the code can be found in this OneDrive

2. WEC Perth 49, developed experimentally by the University of Adelaide, comprises 148 features indicating tidal wave amplitudes at X and Y coordinates, local energy component magnitudes, and the final power output (global component in kW) for large Tidal Wave-energy generation. This dataset serves as a complex multivariate dataset for linear regression, requiring pattern identification to predict both the local energy generation component and the global component. We utilized this dataset to evaluate the scalability and runtime of the Generic Matrix Multiplication Algorithm across different "Cluster types," "Number of workers," and "Algorithm Strategies for computation."

All inputs are in the form of CSV files with each row corresponding to the rows of the matrix. For example for the following matrix $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$ the first line would be "1,2,3".

Dense Matrix Multiplication

Overview

Both horizontal-vertical (H-V) and a vertical-horizontal (V-H) matrix multiplication were implemented in scala. However, the V-H implementation is adapted for $A^T A$ such that some shortcuts are made whereas the H-V implementation can be applied to any two matrices (given that their dimensions agree).

Pseudocode & Analysis

Dense Matrix Multiplication (H-V and V-H)

The following is the pseudocode for the Dense matrix multiplication. It utilizes the horizontal partitioning on the left Matrix and vertical partitioning on the right matrix. The link to this code on github is here.
While the code for intermediate report had the fastest runtime of 37 seconds, it is to be noted that the join strategy program adds up computation overhead to prune the data and also the sorting program along with coalesce, end up increasing the runtimes. However, these steps, especially sorting, helps prune data for next stage.

Procedure 1 Matrix Multiplication Algorithm

```
// Checking Ratio to Determine Algorithm Strategy
ratio ← size(0)/size(1)
num_workers ← args(5).toInt()

// Multiplication = A × B
//Read and Partition Matrix1 and Matrix2 based on DynamicPartitioner
data2 ← sc.textFile(args(1), minPartitions = myPartitionFunc(size(1), num_workers))

val numFeaturesA            // Columns in 1
val numFeaturesB            // Columns in 2

HV_Bool = if(size(1) < 60000 and numFeatures_A ≥ numFeatures_B)thentrueelsefalse

matA ← if (HV_Bool.value) then HorizontalPartitioner(data1) else VerticalPartitioner(data1)
matB ← if (HV_Bool.value) then VerticalPartitioner(data2) else HorizontalPartitioner(data1)

result ← if (numFeatures_A == size(1)) then
    if (ratio ≥ 1 and HV_Bool.value) then
        MatmulBroadcast(matA, matB, sc)
    else  MatmulShuffle(matA, matB, HV_Bool)
else
    logger.error("Arithmetic Error: Matrix inner dimensions do not match! Cannot Multiply.")
    throw new IllegalArgumentException("Matrix inner dimensions do not match! Cannot Multiply.")

logger.info("NumPartitions taken for Processing: {}")
logger.info(result.getNumPartitions)
result.sortBy(_._1)(customComparator, ClassTag[(Int, Int)](classOf[(Int, Int)])).saveAsTextFile(args(2))
```

Procedure 2 Horizontal Partitioning Function(Generic)

Input: Matrix1 or Matrix2 Data :RDD[String]
Output: RDD[(Int,Vector[Double])]

```
val matA = // RDD for distribute,Vector for faster access dataA.map(line => line.split(",")
extract elements from cols .map(_.trim.toDouble).toVector)
Get rid of whitespaces .zipWithIndex.mapcase(vec, ind) => (ind.toInt, vec)
col index
    matA // Return value // Repartition Not Required
```

Procedure 3 Vertical Partitioning Function(Generic)

Input: Matrix1 or Matrix2 Data :RDD[String]
Output: RDD[(Int,Vector[Double])]

```
val matB = // RDD for distribute,Vector for faster access
dataA.map(line => line.split(",") // extract elements from cols.map {(_.trim.toDouble).toVector) // Get rid of whitespaces.zipWithIndex.toVector} //col index
val VpartitionedData_B = matB.flatMap(array => array.map {case (value, index) => (index, value)}
// Swap to index-value
.mapValues(_.map(_._2).toVector) // Convert to Vector
VpartitionedData_B // Return value
```

Procedure 4 Matrix Multiplication with Broadcast

Input: $HPart\_data : RDD[(Int, Vector[Double])], VPart\_data : RDD[(Int, Vector[Double])], sc : SparkContext$
Output: RDD[((Int,Int),Double)]

```
logger : org.apache.log4j.Logger = LogManager.getRootLogger
{logger.setLevel(Level.INFO)}
logger.info("Strategy: H-V Partition-Broadcast")          ▷ Horizontal-Vertical Partition Broadcast Strategy
lookup = sc.broadcast(VPart_data.collectAsMap())

resultRDD : RDD[((Int,Int),Double)]
resultRDD = HPart_data.map{case (ind,vec) ⇒ lookup.value.map{x ⇒
                        ((ind,x._1), vec.zip(x._2))}}
    .flatMap(x ⇒ x.mapValues{ele ⇒ ele.map{nums ⇒ nums._1 * nums._2}})
                            .flatMapValues{x ⇒ x}.reduceByKey(_ + _)
```

Procedure 5 Custom comparator for sorting

Input: A, B - Matrices with agreeing dimensions
Output: Boolean value

```
result = x._1.compareTo(y._1) //Works on Secondary Sort Logic
if (result == 0) x._2.compareTo(y._2) else result //Compare key2 if key1 equal
}
```

Procedure 6 Dynamic Partitioner based on File size

Input: Input rows, Number of Worker Machines
Output: numPartitions :Int

```
defaultParts = max(2, size/10) // Rows in a matrix < 100, 2 is default in Spark
morethan100rows = min(size, defaultParts) //working when one of the matrix is smaller
WorkerbasedPartitions = (morethan100rows, numworkers) //If numWorkers is limited to ¡10
numPartitions = min(WorkerbasedPartitions, 100) // Upper Limit set to 100 partitions
```

Procedure 7 Matrix Multiplication with Shuffle

Input: mat1: RDD[(Int, Vector[Double])], mat2: RDD[(Int, Vector[Double])], HV_Bool: Broadcast[Boolean]
Output: RDD[((Int,Int),Double)]

```
If HV_Bool.value
    logger.info("Strategy: H-V Shuffle")                  ▷ Horizontal-Vertical Shuffle Strategy
    return mat1.cartesian(mat2).map case (indvecA, indvecB) => ((indvecA._1, indvecB._1), indvecA._2.zip(indvecB._2))
                        .flatMap(tuple => tuple._2.map(nums => (tuple._1, nums._1 * nums._2))).reduceByKey(_ + _)
Else
    logger.info("Strategy: V-H Shuffle")                  ▷ Vertical-Horizontal Shuffle Strategy
    return mat1.map(_._2).zip(mat2.map(_._2)).map case(vec1, vec2) =¿ vec1.zipWithIndex.mapi => (i,vec2.zipWithIndex)
                        .flatMap(identity).map case(ele, vec) => vec.map(x =¿ ((ele._2,x._2), x._1 * ele._1))
                        .flatMap(identity).reduceByKey(_ + _)
```

V-H Matrix Multiplication (Lower Triangular Matrix Only)

The pseudocode outlines V-H matrix multiplication adapted for A^TA, employing hash and shuffle techniques to compute values at each coordinate. Given that the columns of A^T match the rows of A, only one input matrix is necessary. Note, the code processes inputs line by line, where each line represents a row of the input matrix A, corresponding conveniently to the columns of A^T. The outputs consist of the coordinates of A^TA and their corresponding values. For instance, if $A^TA = C$, the final output is a pair RDD with keys as (row = i, col = j) and values as C_{ij}. The link to the code on github is here.

```
val conf = new SparkConf().setAppName("ATAMatMul")      // Setup spark conference and context, various
val sc = new SparkContext(conf)                         // settings like setMaster not added in psuedocode

val inputFile = sc.textFile(args(0))                    // Read input file
val mat = inputFile.flatMap {                           // mat will be a RDD of Array[Double] with each
  case (line) =>                                        // array representing row of matrix
    line.split("\n").map(_.split(",").map(_.trim.toDouble))
}

val customPartitioner = new HashPartitioner(5)          // create custom partitioner

val numFeatures = mat.first().length                    // Get number of features from RDD

val matMul = mat.flatMap {
val indexedLine = line.zipWithIndex                     // loop through line (row) with index
  indexedLine.flatMap { case (element, index) =>
    val otherIndexedElements = line.drop(index).zipWithIndex  //  loop through every other element in row
    otherIndexedElements.map { case (otherElement, j) =>     // and multiply with the first element
      ((index, index + j), otherElement * element)
    }
  }
}.partitionBy(customPartitioner).reduceByKey((x, y) => x + y) // Sum for each coordinate in resulting matrix
matMul.saveAsTextFile(args(1))                                 // Save to output
```

Algorithm and Analysis

Dense Matrix Multiplication(Both H-V and V-H)

This algorithm works for any-sized(Rows vs Columns) Pre-multiplier and post-multiplier matrices. A generic matrix multiplication algorithm was implemented using H-V Partition.Since most problems related to Machine Learning or Finite Elements Method Formulation have higher rows(datapoints) than columns(features), we have first utilized HV Partition.

Rows(M1)	Columns(M1)	Rows(M2)	Columns(M2)	HV_Bool (rows<60000)	Columns(M1)> =Columns(M2)	Strategy Chosen
148	13052	13052	148	1	1	H-V Broadcast
148	23000	23000	148	1	1	H-V Broadcast
148	36043	36043	148	1	1	H-V Broadcast
100	1200000	1200000	100	0	1	V-H Shuffle

Rows	Columns	Elements
13052	148	1931696
23000	148	3404000
36043	148	5334364
1200000	100	120000000

1. Size ratio: A Ratio of $\frac{|A|}{|B|}$ is used to check the correct strategy between Partition-Broadcast and Shuffle for larger datasets while giving preference to Broadcast for data size up to 60000 rows.The ratio size(A)size(B), if lower, suggests that the first matrix has less rows and more columns and this favours Vertically Partitioning A and Horizontally Partitining B. Link to Code

2. The tricky aspect of Vertical Partitioning was achieved by first partitioning the matrix horizontally and then repartitioning based on Features. Link to Code

3. This code is based on Dynamic Partitioning for datasize i.e is max(2, size/10) holds for features as low as 10, min(size/10,size) holds for features less than 100, otherwise for large data sizes, partitions default to 100. Link to Code

4. There is a custom Camaparator similar to secondary sort logic, which sorts on first rows of the output matrix followed by columns.Link to Code

V-H Matrix Multiplication for A^TA

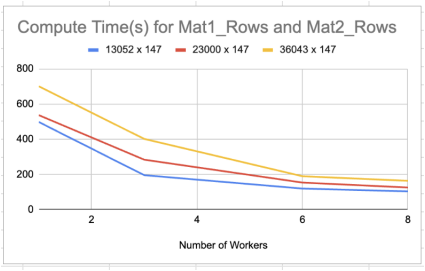
Real data usually have more rows than columns, making the left matrix A^T wide and short when computing A^TA. Employing V-H matrix multiplication is advantageous in such cases since both the vertical (column of A^T) and horizontal (row of A) dimensions are smaller.

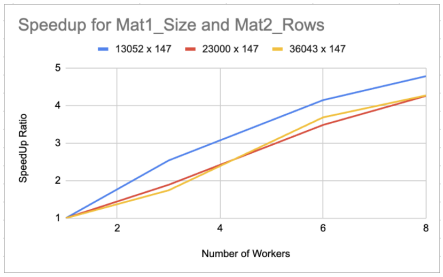
The algorithm also leverages the equivalence between the columns of A^T and the rows of A, resulting in a symmetrical matrix multiplication. Consequently, the program only computes the lower triangle of the resulting matrix multiplication.

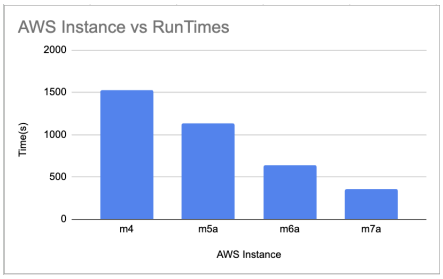
The above implementation partitions each matrix by individual rows and columns which is the most fine grain partitioning possible. However, more coarse-grained partitioning can be implemented and may be considered for the final submission.

Experiments

Dense Matrix Multiplication







1. The first graph above shows similar trends in runtimes for different matrix size(rows) given same. The limit then for computation is around 120 seconds for the file size as there is aminimal time required for reading the data into RDD, saving time, sorting, cleanup and other overheads.
2. The speedup which can also be intimated from the previous graph is has a decreasing gradient that shows, eventually there are less operations that can be parallized i.e. the limit of parallelization is being reached from the jump from 6 to 8 workers. Thus, it is to be noted that if workers are increased beyond this limit, it would give suboptimal performance for that data size.
3. The third image above shows the decreasing runtimes with respect to AWS instances owing to larger memory overheads and transfer bandwidths and more advanced architectures for computation going from m4(older) to m7(newest).
4. The scalability shows that the ratio of work scales linearly to the work, as given in the table below.
5. The matmulDense program outputs an custom datatype (Int,Int) as Key and a Double as value ie. final value is RDD[((Int,Int),Double)]. Int class for key was decided keeping in line with the scala max value for int (2147483647). This is a practical upper bound as big data matrix multiplication for values of rows higher than the above limit is beyond the scope of this subject.

Scalability Ratio for 8 workers (Mat1_rows x Mat2_rowsl)

Data size	Time(s)	Scalability
13052	104	1
23000	142	1.37
36043	164	1.58
1200000	1536	14.77

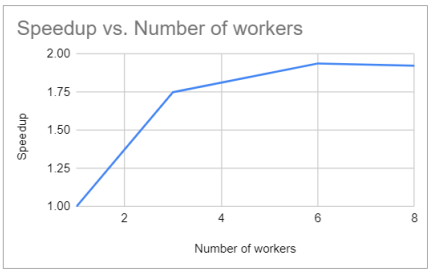
```
((0,0),2.066648764E9)
((0,1),1.7787265E8)
((0,2),2.897284344870014E9)
((0,3),2.322869460900001E8)
((0,4),3.394855331270025E9)
((0,5),3.408901373299996E8)
((0,6),3.6039448311100216E9)
((0,7),3.093561139399998E8)
((0,8),4.193945559E9)
((0,9),3.6237525E8)
((0,10),4.445760890020019E9)
```

The data shows that the program scales well when the data increases. The logs from aws can be found here and the outputs can be found here.

V-H Matrix Multiplication for A^TA

To test for speedup the algorithm was run on a subset of the Wave Energy Converter with a matrix size of 13000x148. All the experiments were run using the m4.xlarge machine on AWS. Several numbers of workers were tested and the following table summarizes the results.

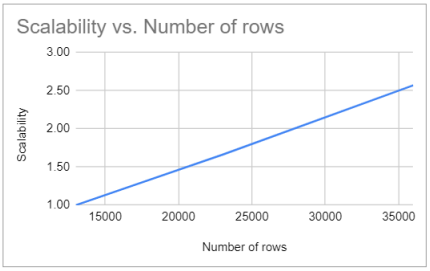
Number of workers	Time (s)	Speedup
1	252	1
3	144	1.75
6	130	1.94
8	131	1.92



The graph shows considerable speedup when increasing the number of workers from 1 to 3. However, as the number of workers increases the increase of speedup decreases. This implies that the parallelibility of the code is not that high. This could be due to the program having to loop through the same row since the row of A^T is the column of A.

The scaleup was also tested by increasing the number of rows in the matrix.

Number of rows	% increase data	Time (s)	Scalability
13000	1.00	130	1.00
23000	1.77	216	1.66
36000	2.77	334	2.57



The data shows that the program scales well when the data increases. The logs from aws can be found here and the outputs can be found here. For reference, the output of this program is a pairRDD with the key being a tuple containing the row and column (0 indexed) of the resulting matrix and the value being the result of matrix multiplication.

Linear Regression

The matrix multiplication and matrix inversion programs were combined to perform linear regression. The formulation of the solution to linear regression can be found in the supplementary materials but essentially if given a set of training features X and training values y the predictors of the OLS model is given by:

$$\beta = (X^T X)^{-1} X^T y \tag{1}$$

Given test data (x) and test values ($\tilde{y}$), the prediction ($\hat{y}$) and mean squared prediction error ϵ can be given by the following matrix operations

$$\hat{y} = \beta x \tag{2}$$

$$\epsilon = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - \tilde{y}_i)^2 \tag{3}$$

Pseudocode

The following is the pseudocode for linear regression using equations 1, 2 and 3. The code on github can be found here:

Procedure 8 Solving Linear Regression using parallel matrix multiplication

Input: X - train feature matrix, y - train values, x - test feature matrix, $\tilde{y}$ - test values
Note: all matrices are RDDs with ordered Array[Double] representing the rows.
Output: ϵ - error for OLS estimator

1: $A^T A$ = vh-ATA-matmul(X) // use v-h dense matrix multiplication optimized for $A^T A$
2: $A^T A\text{Inv}$ = matInv($A^T A$) // matrix inverse function
3: A^T = matrixTranspose(X) // matrixTranspose function transforms RDD
4: $A^T A\text{Inv} A^T$ = matmul($A^T A\text{Inv}$, A^T) // use dense matrix multiplication to multiply $A^T A\text{Inv}$ and A^T
5: β = matmul($A^T A\text{Inv} A^T$, y) // use dense matrix multiplication to multiply $A^T A\text{Inv} A^T$ and y
6: $\hat{y}$ = matmul(β, x) // use dense matrix multiplication to multiply β and x to get predictions
7: ϵ = MSE($\hat{y}$, $\tilde{y}$) // use function to calculate MSE
8: **Return** ϵ

Algorithm and Analysis

In terms of analysis, the choices to be made here are which algorithm to use for matrix multiplication. For the X^TX the V-H optimized for A^TA should be used. As explained before, for most linear regression problems the number of data (rows) is much larger than the number of features (columns). Therefore, A^T is more likely to less rows than columns hence it is better to vertically partition this matrix. The opposite is true for A making it beneficial to use horizontal partitioning. When multiplying $(X^TX)^{-1}$ by X^T the left matrix will be square but the right matrix will most likely have more columns than rows. Therefore, H-V matrix multiplication is beneficial. Finally, for $(X^TX)^{-1}X^T$ and y, the left matrix will have more columns making V-H more preferable. However, depending on the size of the dataset, V-H matrix multiplication may have to be used for all matrix multiplications since H-V relies on the broadcasting of the right matrix. Which is to say, if a single row of the feature matrix X (with the number of elements equaling the number of columns) cannot be broadcasted, then V-H matrix multiplication must be used. In the psuedocode above the 'matmul' function is used for matrix multiplication but the program will decide based on the size of the matrix which matrix multiplication to use.

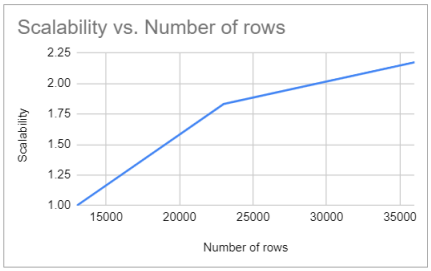
Experiments

The wave energy conversion dataset was used to test the code. Different number of workers The size dataset was modified for experimentation, similar to the matrix multiplication. To test the scaleup the program was run using m4.xlarge with 3 workers and 6 workers. With the full dataset of 36000x147. The results are shown below:

Number of workers	Time (s)	Speedup
3	824	1
6	648	1.27

The scale up is not that significant which is somewhat expected due to the fact that all the matrix multiplications have to be performed sequentially. The following table and plot shows the results for changing the size of the input data.

Number of rows	% increase data	Time (s)	Scaleability
13000	1.00	298	1
23000	1.77	546	1.83
36000	2.77	648	2.17



The scalability of the program is better than the speedup as the proportion of time increase is less than the proportion of data increase.

The links to the aws outputs are here and the logs can be found here. The outputs of this program is the mean squared error of the features which is printed to the logs.

Conclusion

In this project several implementations for matrix multiplications were completed, specifically Dense V-H and H-V matrix multiplication. Then these implementations were combined to perform linear regression.

In conclusion, the analysis presented underscores the critical importance of appropriately selecting between Partition-Broadcast and Shuffle strategies for optimal performance in handling large datasets. The delineation based on the size ratio between matrices A and B serves as a guiding principle, with Broadcast being favored for data sizes up to 60000 rows. Furthermore, incorporating Dynamic Partitioning tailored to varying data sizes and features. The custom Comparator ensures precision in sorting, complementing the overall efficiency of the system.

Moreover, the empirical findings reveal key insights into runtime behaviors and scalability dynamics. These trends highlight the significance of computational limits, parallelization thresholds, and resource scalability, notably emphasizing the diminishing returns beyond a certain point in worker count. The analysis of AWS instance performance helped us understand the significance of memory overheads, transfer bandwidths, and architectural advancements on runtime efficiencies, and design our algorithms to cater to such needs.