CS 6240:Large Scale Parallel Data Processing
Section: Mon-Wed
Project Final Report

Team Members*:* Hardik Shah, Kevin Shah, Ruturaj Nene
Github Repository: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project

Project Overview:
Our goal for the project is to multiply two huge matrices using MapReduce of which one task (1 task) is multiplying sparse matrices using H-V partitioning and other (2 tasks) is multiplying dense matrices using B-B partitioning.
We were able to successfully complete the Sparse H-V and Dense B-B matrix multiplication and applied Sparse H-V matrix multiplication to find out pagerank of nodes in any graph.
For Sparse H-V, we conducted four experiments for speedup, scalability, partitioning and comparison of pagerank for a synthetic graph using the Homework4 MR algorithm and matrix multiplication.
For Dense B-B, we conducted three experiments for speedup, scalability and partitioning.

Input Data:

- For Sparse H-V matrix multiplication task, input is one matrix and one vector with the format: L,rowId,columnId1:value,columnId2:value…
  where 'L' is the matrixId denoting that this is a record from the left matrix and the rest of the record represents the row with only columns having non-zero values. The right vector has the exact same format except the first letter being 'R' instead of 'L'.
- For Dense B-B matrix multiplication task, input are two matrices with the format: L,rowId, value1, value2, value3….
  where 'L' is the matrixId denoting that this is a record from the left matrix and the rest of the record represents the row with all columns one after other. The right matrix has the exact same format except the first letter being 'R' instead of 'L'.

Sparse matrix product H-V (MapReduce) (1 task)

Overview:
As part of this task we have performed matrix multiplication on our generated sample matrix of size P x Q and a vector Q x 1 for two cases: P = 20k, Q = 20k and P = 30k, Q = 30k both of them represented in sparse format. Although this task is stated as two tasks in the project guide, as per professor's advice we are considering it as one task because of multiplying matrix with vector instead of generic matrix multiplication.
In order to find pagerank in a graph, the transition matrix has to be multiplied iteratively with the pagerank vector. Hence, as part of this task we have multiplied the transition matrix (NxN) with the pagerank vector (Nx1) on generated synthetic graph (k=500) from Homework 4 and analysed it with the Homework 4 output (for iterations=10)

Pseudo-code: Github:
https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/Sparse-hv

```
Class Mapper {
   setup(context):
      // H-partitioning over Left Matrix, H is the number of partitions in
      // left matrix
```

```
      H = context.getConf().get("H")

   map(matrixId, rowId, columns=[colId1:val1, colId2:val2 ...]):
         if(matrixId=L):
            // Select random integer from range (1,H)
            randomRow= random(1,H)

            // for all columns in the row emit the randomRow as key, we do
            //not need to consider the partitions in right matrix as there
            //is only 1 partition since the right matrix is a vector
            for all col, val in columns:
                  emit(randomRow, (matrixId,rowId,col,val))
         else:
             // emit this tuple for all the partitions in the left matrix
             for i in range(1,H):
                 emit(i,(matrixId,rowId,col,val))
}

Class Reducer {
     setup(context):
          // Q: no. of rows of the right matrix = no of cols of left matrix
          Q = context.getConf().get("Q")

     reduce(regionId, values=[(matrixId,rowId,colId,cellValue),()...]):
           // A HashMap to store the left matrix(transition matrix)
           leftMatrixMap;
           // A vector to store right matrix(pagerank values) of size Q
           rightVector= Array[Q];
           ALPHA = 0.15 // Random Surfer probability to jump to random page

           for val in values:
                 matrixId,rowId,colId,cellVal= val.split(",")
                 // If matrixId==Right store value in vector for right
                // Matrix else store in the HashMap both indexed by rowID
                 if (matrixId=R):
                       rightVector[rowId]=cellVal
                 else:
                       leftMatrixMap[rowId].add((colId,cellVal))

           // Iterate over entries for each row in Left and compute the
             matrix product with right
           for rowId in leftMatrixMap:
                columns=rowId.getValues()
                sum=0
                for col, cellVal in columns:
                     sum += cellValue*rightVector[col]

                sum = ALPHA/Q + (1-ALPHA)*sum // Calculating new pagerank
```

```
            // Emit the matrix product for a (row,col) pair only if
               Non-zero, as the output matrix will have only one column,
              we emit for every (row,1) pair with non-zero value
             if (sum!=0):
                 emit(Null, (R,rowId,1:sum))

}
```

Algorithm and Program Analysis:

Experiments:

Total 3 experiments were conducted: (H-the number of partitions in the left matrix)

1. **Speedup:** 30k x 30k matrix, 30k x 1 matrix, 4-machine m4.large vs 8-machine m4.large , H = 20 for both cases:

Cluster	Running time
1 master 4 node, matrix size: 30k x 30k and 30k x 1	7 min 28 sec
1 master 8 node, matrix size: 30k x 30k and 30k x 1	5 min 4 sec

The speedup is around 1.5 (448sec/304sec) which seems to be pretty good when the cluster size is doubled.

Github Output:

- 4-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-4-30k-SparseHV
- 8-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-8-30k-SparseHV

Github Logs:

- 4-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-4-30k-SparseHV
- 8-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-8-30k-SparseHV

2. **Scalability:** 30k x 30k matrix, 30k x 1 matrix vs 20k x 20k matrix, 20k x 1 matrix, both on 4 m4.large machines, H = 20 for both cases:

Cluster	Running time
1 master 4 node, matrix size: 20k x 20k and 20k x 1	3 min 26 sec
1 master 4 node, matrix size: 30k x 30k and 30k x 1	7 min 28 sec

Github Output:

- Small cluster (4 machines) 20k x 20k matrix, Partitions(H) = 20: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-4-20k-SparseHV
- Small cluster (4 machines) 30k x 30k matrix, Partitions(H) = 20: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-4-30k-SparseHV

Github Logs:

- Small cluster (4 machines) 20k x 20k matrix, Partitions(H) = 20: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-4-20k-SparseHV
- Small cluster (4 machines) 30k x 30k matrix, Partitions(H) = 20: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-4-30k-SparseHV

3. **Partition Granularity:** 30k x 30k matrix, 30k x 1 matrix, 20 partitions vs 20k Partitions:

Cluster	Running time
1 master 4 node, matrix size: 30k x 30k and 30k x 1, 20 Partitions	7 min 28 sec
1 master 4 node, matrix size: 30k x 30k and 30k x 1, 20k Partitions	21 min 52 sec

The partition granularity affects the running time greatly as more fine grained partitions lead to higher data duplication and shuffling which acts as a bottleneck in-spite of the parallelisation offered by fine granularity.

Github Output:

- Coarse-Grained Partition (20 Partitions): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-4-30k-SparseHV
- Fine-Grained Partition (20,000 Partitions): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-4-30k-SparseHV-FineGrainedPartition

Github Logs:

- Coarse-Grained Partition (20 Partitions): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-4-30k-SparseHV
- Fine-Grained Partition (20,000 Partitions): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-4-30k-SparseHV-FineGrainedPartition

4. **Pagerank Analysis:** Pagerank on 250k x 250k matrix, 250k x 1 matrix, H = 25 (Sparse H-V) vs Synthetic graph k=500 on Homework 4 pagerank, both on 10 m4.large machines

Cluster	Running time
1 master 10 node, matrix size: 250k x 250k and 250k x 1, H=25	24 min 8 sec
1 master 10 node, synthetic graph k=500, HW4	10 min 56 sec

Github Output:

- PageRank Matrix Multiplication (250k x 250k and 250k x 1, H=25): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/SparseHV/m4.large-10-250k-pageRank-SparseHV-part-25
- PageRank Synthetic Graph, HW4 (k=500): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/m4.large-10-250k-pageRank-HW

Github Logs:

- PageRank Matrix Multiplication (250k x 250k and 250k x 1, H=25): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/SparseHV/m4.large-10-250k-pageRank-SparseHV-part-25
- PageRank Synthetic Graph, HW4 (k=500): https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/m4.large-10-250k-pageRank-HW

Result Sample:
R,8194,1:20596792
R,12290,1:4575338
R,7171,1:25208113
...
The output multiplication result of PxQ and Qx1 will be Q x 1 vector in the same format as input represented as a sparse matrix.

Dense matrix product B-B (MapReduce) (2 tasks)

Overview:
The goal of this task is to find the important users from twitter data similar to task 1 but this time on dense matrix and partitioning block by block.
The input matrix now will be a dense matrix (PxQ) containing 0 values as well and another matrix vector (QxR) will also be a dense matrix. The first and second matrix now will be partitioned into blocks rather than H-V. This task will multiply two matrices of generic sizes (PxQ and QxR) and thus we can apply this to compute pagerank with matrices of size NxN and Nx1.

Pseudo-code: Github:
https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/Dense-BB

```
Class DenseBBMapper {
   setup(context):
      // Get number of left-horizontal, left-vertical, right-vertical
      // partitions from the context

   map(matrixId, rowId, columns=[val1, val2 ...]):
```

```
        if (matrixId=L):
           // Select partition number I for this row
           I = rowId % leftHorizontalPartitions

    // for all columns in the row emit the key as (I,J,k) where I
    // is the partition number for this row, J for the column and k
    // is the partition number for all vertical partitions in right matrix.
             for all col, val in columns:
                  // Select partition number J for this column
                  J = col % leftVerticalPartitions
                  for all k in rightVerticalPartitions:
                        emit((I,J,k), (matrixId,rowId,col,val))
        else:
            // Select partition number J for this row
            J = rowId % leftVerticalPartitions
            // for all columns in the row emit the key as (i,J,K) where J
            // is the partition number for this row, K for the column and
            // i is the partition number for all horizontal partitions in
            // left matrix
            for all col, val in columns:
                // Select partition number K for this column
                K = col % rightVerticalPartitions
                    for all i in leftHorizontalPartitions:
                       emit((i,J,K), (matrixId,rowId,col,val))
}

Class DenseBBReducer {
     reduce(regionId, values=[(matrixId,rowId,colId,cellValue),()...]):
           // A map (key:row) of map (key:col) for the left sub-matrix
           leftRowsToColMap;
           // A map (key:col) of map (key:row) for the right sub-matrix
           rightColsToRowsMap;

           for val in values:
                matrixId,rowId,colId,cellVal= val.split(",")
                if (matrixId=R):
                      rightColsToRowsMap[colId][rowId]=cellVal
                else:
                      leftRowsToColMap[rowId][colId]=cellVal

           // Iterate over entries in left and right maps and compute the
           // partial matrix product for the set of j columns (of left
matrix)
           for i in leftRowsToColMap.keys():
               for k in rightColsToRowsMap.keys():
                     rightRowKeys=rightColsToRowsMap[k].keys()
                     sum=0
                     for j in rightRowkeys:
                          sum += leftRowsToColsMap[i][j] *
```

```
                                    rightColsToRowsMap[k][j]
                        emit(Null, (i,k,sum))
}

Class PartialSumMapper {
   map(rowId, colId, partialSum):
        emit(rowId, (colId,partialSum))
}

Class PartialSumReducer {
     reduce(rowId, values=[(colId1,partialSum1),(colId2,partialSum2)...]):
           // A HashMap to aggregate the partial sum for each column
           colIdtoSumMap;

           for colId, partSum in values:
                colIdtoSumMap[colId]+=partSum
           for colId,sum in colIdtoSumMap.keys():
                emit(Null, (rowId,colId,sum))
}
```

Algorithm and Program Analysis:

Experiments:

Total 3 experiments were conducted:
H1- number of horizontal partitions in left matrix
V1 - number of vertical partitions in left matrix = number of horizontal partitions in right matrix
V2 - number of vertical partitions in right matrix

1. **Speedup:** 6k x 6k matrix, 6k x 6k matrix, 5-machine m4.large vs 10-machine m4.large , for both cases (H1, V1, V2 = 10 for both cases):

Cluster	Running time
1 master 5 node, matrix size: 6k x 6kand 6k x 6k	73 min 18 sec
1 master 10 node, matrix size: 6k x 6kand 6k x 6k	34 min 52 sec

The speedup is around 2 which seems to be pretty good when the cluster size is doubled.

Github Output:

- 5-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-5-6k-DenseBB-part-10
- 10-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-6k-DenseBB-part-10

Github Logs:

- 5-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-5-6k-DenseBB-part-10
- 10-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-6k-DenseBB-part-10

2. **Scalability:** 5k x 5k matrix, 5k x 5k matrix vs 6k x 6k matrix, 6k x 6k matrix vs 7k x 7k matrix, 7k x 7k matrix, all on 10 m4.large machines (H1, V1, V2 = 10 for all cases):

Cluster	Running time
1 master 10 node, matrix size: 5k x 5k matrix, 5k x 5k matrix	27 min 14 sec
1 master 10 node, matrix size: 6k x 6k matrix, 6k x 6k matrix	34 min 52 sec
1 master 10 node, matrix size: 7k x 7k matrix, 7k x 7k matrix	44 min

Github Output:

- 5k x 5k matrix, 5k x 5k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-5k-DenseBB-part-10
- 6k x 6k matrix, 6k x 6k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-6k-DenseBB-part-10
- 7k x 7k matrix, 7k x 7k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-7k-DenseBB-part-10

Github Logs:

- 5k x 5k matrix, 5k x 5k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-5k-DenseBB-part-10
- 6k x 6k matrix, 6k x 6k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-6k-DenseBB-part-10
- 7k x 7k matrix, 7k x 7k matrix: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-7k-DenseBB-part-10

3. **Partition Granularity:** H1 = V1 = V2 = 5, 10, 20 all for 6k x 6k, 6k x 6k matrices with 10 workers:

Cluster	Running time
1 master 10 node, matrix size: 6k x 6k and 6k x 6k, H1 = V1 = V2 = 5	24min 30sec
1 master 10 node, matrix size: 6k x 6k and 6k x 6k, H1 = V1 = V2 = 10	34min 52sec
1 master 10 node, matrix size: 6k x 6k and 6k x 6k, H1 = V1 = V2 = 20	62min 20sec

The partition granularity affects the running time greatly as more fine grained partitions lead to higher data duplication and shuffling which acts as a bottleneck in-spite of the parallelisation offered by fine granularity.

Github Output:

- H1 = V1 = V2 = 5, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-6k-DenseBB-part-5
- H1 = V1 = V2 = 10, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-6k-DenseBB-part-10
- H1 = V1 = V2 = 20, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/DenseBB/m4.large-10-6k-DenseBB-part-20

Github Logs:

- H1 = V1 = V2 = 5, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-6k-DenseBB-part-5
- H1 = V1 = V2 = 10, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-6k-DenseBB-part-10
- H1 = V1 = V2 = 20, 6k x 6k, 6k x 6k matrices https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/DenseBB/m4.large-10-6k-DenseBB-part-20

Result Sample:
194,205:6000,302:6000,450:6000...
10,33:6000,65:6000,780:6000...
711,21:6000,2:6000...
...
The output multiplication result of PxQ and QxR will be PxR matrix in the row-major order format where the first value is rowId followed by comma separated columnId:value pairs.

Conclusions
We were able to successfully complete the sparse matrix multiplication partitioned horizontally for left matrix and vertically for right matrix. We conducted 4 experiments for speedup, scalability, partitioning and comparing pagerank via matrix multiplication with HW4 algorithm. We achieved speedup of 1.5, good amount of scalability, found that it takes more time for more fine grained partitioning due to higher data duplication and also that HW algorithm for computing pagerank beats the matrix multiplication algorithm.
We were able to successfully complete the dense matrix multiplication partitioned block by block for matrices of any generic size. We conducted 3 experiments for speedup, scalability and partitioning. We achieved speedup of 2, good amount of scalability, found that it takes more time for more fine grained partitioning due to higher data duplication.
For future work, we can use the dense B-B matrix multiplication for many applications which requires multiplying two huge matrices such as computing pagerank just like we did for sparse H-V matrix multiplication.