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

Team Members*:*
Kevin Shah
Hardik Shah
Ruturaj Nene

Github Repository: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project

Project Overview:
Our goal is to find which users are more important than the most in a big and convoluted graph of users following other users like that of Twitter. We thought of defining “importance” as pagerank of a user and we are using matrix multiplication to compute pagerank of the users. As the transition matrix (NxN, N is the number of vertices) used for computing pagerank is huge and very sparse, we have used Sparse H-V matrix multiplication and Dense B-B matrix multiplication using MapReduce.

We were able to successfully complete the sparse matrix multiplication partitioned horizontally for left matrix and vertically for right matrix. We conducted three experiments for speedup, scalability and partitioning.

Input Data:
We are working with Twitter dataset available at http://socialcomputing.asu.edu/datasets/Twitter which is basically a graph of users with two files - nodes.csv and edges.csv.
nodes.csv: it's the file of all the users. This file works as a dictionary of all the users in this data set. It's useful for fast reference. It contains all the node ids used in the dataset.
edges.csv: this is the friendship/followership network among the users. The friends/followers are represented using edges. Edges are directed.
Here is an example.
1,2
This means user with id "1" is following user with id "2".

The above graph (edges.csv) will be converted to a transition matrix which will be the input to our MR job for matrix multiplication, the input format for which is given below:
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 matrix has the exact same format except the first letter being ‘R’ instead of ‘L’.

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

Overview:

Inorder to find important users, the transition matrix of user's outlinks has to be multiplied iteratively with that corresponding pagerank vector. Hence, as part of an intermediate report we are multiplying the transition matrix (PxQ) with the pagerank vector (Qx1) on our sample matrix of 20k x 20k and 30k x 30k represented in block oriented format as sparse matrix.
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.

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

```
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 randowRow 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];

        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]
            }

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

   Speedup:

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:
   1. 4-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/m4.large-4-30k-SparseHV
   2. 8-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-output/m4.large-8-30k-SparseHV

   Github Log:
   1. 4-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/m4.large-4-30k-SparseHV
   2. 8-machine: https://github.ccs.neu.edu/cs6240-f19/Team-Non-Sequential-Project/tree/master/aws-logs/m4.large-8-30k-SparseHV

2. 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(Scalability):

   Scalability:

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:
   1. 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/m4.large-4-20k-SparseHV
   2. 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/m4.large-4-30k-SparseHV

   Github Logs:
   1. 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/m4.large-4-20k-SparseHV
   2. 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/m4.large-4-30k-SparseHV

3. 30k x 30k matrix, 30k x 1 matrix, 20 partitions vs 20,000 Partitions (Partition Granularity):

   Partition Granularity:

Cluster	Running time
1 master 4 node, matrix size: 30k x 30k and 30k x 1, 20 Partitions	7min 28sec
1 master 4 node, matrix size: 30k x 30k and 30k x 1, 20000 Partitions	21min 52sec

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

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

Github Log:

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

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)

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.

Conclusions

We were able to successfully complete the sparse matrix multiplication partitioned horizontally for left matrix and vertically for right matrix. We conducted three experiments for speedup, scalability and partitioning. We achieved speedup of 1.5, good amount of scalability and found that it takes more time for more fine grained partitioning due to higher data duplication.

For future work, we can make the sparse matrix multiplication generic for multiplying two large matrices rather than one NxN matrix and other Nx1. Also, we'll be working on multiplying two dense matrices using block by block partitioning as task 2,3.