PARALLEL DATA PROCESSING IN MAPREDUCE – CS 6240
Prof. Mirek Riedewald
VIJET BADIGANNAVAR

Pseudocode:

For both the versions, the matrices are represented as Column Major in which for a node, the nodes that are present in the rows represent the outlinks associated with that node and the columns associated with that node represent the inlinks associated with it.
For ex:

$$\begin{array}{cccccc} & A & B & C & D & \\ A\,[& 0 & 1 & 1 & 0 &] \\ B\,[& 0 & 0 & 0 & 0 &] \\ C\,[& 0 & 0 & 0 & 0 &] \\ D\,[& 1 & 0 & 0 & 0 &] \end{array}$$

Considering A - it has the outlinks B and C. If we consider A column wise then it has an inlink from D. B and C are dangling nodes.

This matrix is represented in Sparse format as
Matrix M:
A[B,C]
D[A]

Matrix D (indicating Dangling nodes):
B [0.25]
C [0.25]

Matrix R (indicating Rank Matrix):
A [0.25]
B [0.25]
C [0.25]
D [0.25]

Step-1 : Preprocessing and Initial PR initialization.

This step is a map reduce job. The goal of this job is to find out all the links present in the graph. So the mapper is going to process each line and emits all the links and outlinks. As we need to get hold of all the links a single reducer is required. The paritioner has to send all the records to a single reducer (Default Partioner). The reducer is going to accumulate all the links and assign a unique id to each one of them. It prepares the sparse matrix by adding the outlinks Ids for each of the linkids and writes it to a file. In the cleanup process, we get to know total number of nodes in the graph, thus we can initialize the Dangling nodes and assign the default pagerank values to all the nodes in the rank matrix.

Thus, at the end of this job, Matrix M containing the linksids with all the outlinks. The matrix D containing all the dangling nodes ids as row ids. The matrix R contains all the node ids and the page rank values associated with that node.

Mapper:

// Mapper function reads the line from the input and preprocesses it. It fetches all the outlinks associated
// with that link l if link is valid and send each of those links as keys to the reducer.

```
class Map{
    map (offset B, Line l) {
        // l is a record that contains linkName:<htmlpage>
        // Preprocessing is a function that validates if the line is valid and fetches all the outlinks
        associated     // with l.
        preprocess the line l

        if(l is valid){
            fetch all the outlinks for given link l
            emit(l,"")
            // send each of the outlink to the reducer
            for each outlink o of l{
                emit(o,"");
                emit(l,o);
            }
        }
      }
}
```

Reducer:

// Reducer accumulates all the links in the graph. For each link reducer assigns an unique id to it. If the link
// is a dangling node then it will update that link Id in the dangling file with nodeId as name, 1/|V| as its
// value. If a link is a not a dangling node, then that link is written to M file in sparse format (as described in
// heading) with linkId and outlinksId. It also assigns default pagerank value to all the nodes in the graph.

```
Class Reducer{
//initialize all the data structures
setup(){
    //LinksMap : link -> ListOfIds
    //DanglingSet is a datastructure that stores all the dangling nodes
    // counter is used to assign a unique Id to each link
    initialize multiple outputs mos
    initialize counter = 0
    initialize linksMap;
    initialize danglingSet;
}

// assigns unique id to all the links. If a link is a dangling node then it updates the dangling set otherwise it
// writes the linkId with listOfOutlinksIds
reduce(link, [outlink1,outlink2,..]){
    if link is not present in the linksMap{
        put the link in linksMap with a uniqueId(counter++)
        write(mos,"MappingFile", link, uniqueId)
    }

    if(count(outlinks) == 0)
        add link to danglingSet
        else{
                for each outlink o in outlinks{
                        if (o is **not** present in the linksMap){
                                put the link in linksMap with a uniqueId(counter++)
                                write(mos,"MappingFile", o, uniqueId)
                        }
                        accumulate all the outlinksIds for link
                }
                write(mos, "M", linkId, listofoutlinksIds)
        }
}

// This funcation has access to the total number of nodes present in the graph. Thus, this will create the
// dangling matrix with value 1/|N| and also assigns default pagerank to all the nodes.
cleanup(){
        TOTAL_LINKS = size of linksMap;
        foreach (nodeId in danglingNodes){
                write(mos,"D", nodeId, 1/ TOTAL_LINKS)
        }
        foreach (nodeId in linksMap){
                write(mos,"R",nodeId, 1/ TOTAL_LINKS)
        }
}
}
```

Version A:

Step 2 is divided into 3 separate jobs. Thus there are 3 jobs per iteration and this is repeated from 10 iterations thus, there will be 10 x 2 = 20 jobs during this whole step. The first job is calculation of Dangling score. The second task is calculation of Pagerank values for all the nodes that contains 2 jobs.

Dangling Score calculation Job:
This job deals with 3 matrices. Dangling Matrix D [1 X V] in sparse format, with each element having a value of 1/V.PageRank Matrix i.e. R is [V X 1]. Multiplication of D X R gives a [1 X 1] matrix i.e. accumulated dangling score that that should be shared with all the nodes in the graph.

The job consists of two mappers one for Matrix D and other for Matrix R. The reducer will compute the DxR. For each dangling node in the reducer it increments the counter. At the end of the job, the value of the counter is the total dangling score that needs to be shared across all the nodes in the graph.

Mapper for Matrix D

```
// Emits the ((i,j,k),value) where refers to the i row of the first matrix, j refers to the columns of the second
// matrix and k is the rowId of the first matrix.
class Map {
        map (key, value){
                // value is a dLink,dValue
                obtain the dLinkId and dValue from value by splitting it on ","
                // emit ((i,j,k),value)
                emit ({0, 0, dLinkId}, dValue)   //  i=0 as D has one row, j=0 as R has one column
        }
}
```

Mapper for matrix R

// Emits the ((i,j,k),value) where refers to the i row of the first matrix, j refers to the columns of the second
// matrix and k is the rowId of the first matrix.

```
class Map {
        map (key, value){
                // value is a rLink,prValue
                obtain the rLink and prValue from value by splitting it on ","
                // emit ((i,j,k),value)
                emit ({0, 0, rNode }, prValue)   //  i=0 as D has one row, j=0 as R has one column
        }
}
```

Reducer

```
// Accumulate the dangling score coming from all the dangling nodes and increment the counter.
class Reducer{
        reduce (nodeId, values){
                dScore= 0.0
                if the nodeId has two values (v1,v2) {
                        dScore = v1 * v2;
                        increment the counter("DanglingScore) by dScore;
                }
        }
}
```

Note: Dangling score computation is same for both the versions.

COLUMN BY ROW Matrix Multiplication (Version – A)

Adjacency Matrix M is [V X V] in sparse format,
PageRank Matrix R is [V X 1]
Multiplication of M X R gives a [V X 1] matrix i.e. new Page rank values of all the nodes.
Partition by columns: For each columns emit non zero elements for each row (keys as nodeIds).

Explanation:
Matrix M:
A[B, C]
D[A, B]

Matrix D (indicating Dangling nodes):
B [0.25]
C [0.25]

Matrix R (indicating Rank Matrix):
A [0.25]
B [0.25]
C [0.25]
D [0.25]

Records emitted from M mapper are,

(0, 1) key and (1, 0.5) value (meaning 0 is contributing to 1 and the factor is 1/2)
(0, 1) key and (2, 0.5) value
(3, 1) key and (0, 0.5) value
(3, 1) key and (1, 0.5) value

Records emitted from R mapper are,

(0 ,0) key and (0, 0.25) value
(1, 0) key and (0, 0.25) value
(2, 0) key and (0, 0.25) value
(3, 0) key and (0, 0.25) value

Records coming to Reducer for index 0
(0,0) (0,0.25) parent score
(0,1) (1,0.5) outlink contribution
(0,1) (2,0.5) outlink contribution

Thus for index 0, Pagerank contribution of 0 is (0.25 * 0.5 + 0.25 * 5)

The same process is continued for other links.

This task is divided into 2 jobs. The first job does the shares its pagerank values to all the outlinks. The second job gathers all the values from the inlinks and computes the new pagerank for that nodeId.

Intermediate Job - 1 :
The M mapper is going to emit all the outlinks associated with the {linkId,2}. The R mapper is going to emit all the links with {linkId,1}. Secondary design pattern is used to check if the linkId at the reducer is a dead node or not. In the reducer if the first record is not coming from R then that link is a dead node and it gets its default score. If the first record is from R then, we have to multiply that link share with its pagerank value and should be sent to that node. The grouping comparator is used that is used to achieve secondary sort that sorts by key. The partitioner is used that partitions based on the hashcode of the key alone so that all the nodes values from different mappers go to the same reducer.

Mapper for Matrix M:
```
Class MapperM{
    Map(…,line){
        // line is of linkId-outlinks
        Obtain the linkId and the outlinks associated with it
        prShare = 1/ sizeOfOutlinksList
        for each(outlink o in outlinks){
            emit({linkId,2},{o,prShare})
        }
    }
}
```

Mapper for Matrix R:
```
Class MapperR{
    Map(…,line){
        // line is of linkId-prValue
        Obtain the linkId and the prValue associated with it
        emit({linkId,1},{0, prValue })
    }

}
```

Grouping Comparator
// This comparator is used for secondary design pattern such that the reducer always receives the R matrix
// values first then the M values.

```
Comparator ({LinkId1, matrixId1},{linkId2,matrixId2}){
    Sort by linkId;
}
```

Partitioner
// Partion the keys such that all the links go to the same reducer based on the hashcode of the link.

```
Class Partitioner{
    getPartition({LinkId, matrixId},{index,value}){
        Sort by hashcode(linkId) % NUMBER_OF_PARTITIONS;
    }
}
```

Reducer
// As secondary design pattern is used, the matrixIds are sorted and if the first record is not coming from R,
// then that node is a dead node. So it writes that node to the intermediate node (indicating that node is a
// dead node). If not, then it gathers the page rank score and emits its value to that particular node.

```
Class Reducer{
    Setup(){
        Read the TOTAL_LINKS and PAGE_RANK_SHARE from the configuration.
    }

    reduce({LinkId, matrixId},[{index1,value1},{index2,value2}…]){
        score = 0.0
        // if the first record is not coming from the M matrix, then it means that node is a dead node
        if(matrixId != 1){
            score = (0.15/TOTAL_LINKS)+(0.85*PAGE_RANK_SHARE);
            //indicating this is a dead node
            emit((linkId1,0.0),null)
            //share its value with outlinks
            emit((index1,score*value1),null);
        }else{// if the first record is coming from M then get its value
            score = value1;
        }

        //for each of the values coming from M emit the outlinks share
        for each ({index,value} in inputlist){
            emit((index,score*value),null);
        }
    }
}
```

Job 2 :
This job just accumulates the values for each of the nodes and emits it. This is a similar to WordCount program. Inmapper combiner is used to

```
Mapper{
    //Map: rankId -> double
    rankMap
    Setup(){
        Initialize rankMap
    }

    Map(…,Value){
        Obtain the nodeId and pagerankshare from the Value
        If(rankMap contains nodeId)
            Add the pageRankvalue share to that pagerank value for nodeId in rankMap
        Else
            Add nodeId and pageRankshare to rankMap
    }

    cleanup(){
        for each entry e in rankMap{
            emit(id,pagerankvalue);
        }
    }

}
}

// Reducer just accumulates the pagerank values and computes the new pagerank. Similar to the word count
// design pattern
Reducer(){

Setup(){
        Read the TOTAL_LINKS and PAGE_RANK_SHARE from the configuration.
}




 Reduce(nodeId,[value1,value2…]){
  tempPrValue = compute the sum of all the values of the input list
  newPRscore = 0.15/TOTAL_LINKS + 0.85 * (PAGE_RANK_SHARE+ tempPrValue);
  emit((nodeId,newPRScore),null)
}
}
```

ROW by COLUMN Matrix Multiplication (Version – B)

Adjacency Matrix M is [V X V] in sparse format,
PageRank Matrix R is [V X 1] is distributed to each of the mappers using distributed cache
Multiplication of M X R gives a [V X 1] matrix i.e. new Page rank values of all the nodes.
Partition by rows: For each row, emit the non zero elements for each columns (keys as outlinksIds)

Explanation(with example):
Matrix M:
A[B, C]
D[A, B]

Matrix D (indicating Dangling nodes):
B [0.25]
C [0.25]

Matrix R (indicating Rank Matrix):
A [0.25]
B [0.25]
C [0.25]
D [0.25]

Records emitted from mapper
[1,(0.5*0.25)]
[2,(0.5*0.25)]

Reduces just accumulates the inlinks contributions of each node and computes the new page rank values.

This approach just uses one Map reduce job. The R matrix is shared to each of the mappers using distributed cache. The M contains each of the outlinks, associated with that node. It gets the pagerank share from the R matrix that is loaded into the memory and distributes its share to each of the outlinks. It also emits the pagerank value of 0.0 for itself so that if the node is a deadNode it will get its default share and those nodeId are not lost. Default Partitioner is used.

```
Mapper{
    ranksMapping;
    localMapping;

    // Initialize the mapping
    setup(){
        get the rank file from the cache and load the rankMapping map with nodeId and
        pagerankvalue.
    }

    //Accumulate the pagerank values for all the outlinks associated with the nodeId by multiplying its
    // share with the pagerank of that node
    map(key, value){
        get the nodeId and outlinksList from value;
        prShare = 1/size(outlinksList)
        for(outlink o in outlinksList){
            if localMapping contains outlink
                update the value for the outlink in localmapping by adding
                prShare*ranksMapping(nodeId) to the existing value
            else
                add (nodeId, prShare*ranksMapping(nodeId)) to local Mapping
        }

        add (nodeId,0.0) to localMapping if localMapping does not contains an entry for nodeId
    }

    // emit all the values of the localMapping
    cleanup(){
        for each entry in rankMap{
            emit(nodeId,pagerankvalue);
        }
    }
}


// Reducer just accumulates the pagerank values and computes the new pagerank. Similar to the word count
// design pattern
Reducer(){

    Setup(){
            Read the TOTAL_LINKS and PAGE_RANK_SHARE from the configuration.
    }

    Reduce(nodeId,[value1,value2…]){
      tempPrValue = compute the sum of all the values of the input list
      newPRscore = 0.15/TOTAL_LINKS + 0.85 * (PAGE_RANK_SHARE+ tempPrValue);
      emit((nodeId,newPRScore),null)
    }
}
```

Step 4:

Explanation:
This is a separate Map reduce job. The design pattern used is top-k design pattern. There are two mappers. One mapper takes the gathers the local top 100 ranks and sends it to the reducer. The other mappers emits all the links associated with the ids.(reverse mapping to get the links names from the id). There is only one reducer and default partitioner is used. The reducer will first gather all the links that receives two paramters as values. (These two parameters are linkName and the pagerank score, indicating that link is in local 100 in some mapreduce). Once it receives all the top 100 records from all the reducers, it finds the global top-100 and prints those values to the file.

```
// Gather the top 100 records and emits them as linkId,value.
Mapper for R matrix:

localTop100
Setup(){
        Initialize localTop100
}

map(…, x){
    // x is a (linkId, PageRankValue)
    pg_x = get the pagerank value of x;
    if pg_x is in localTop100
        add x to  localTop100
}

cleanup(){
    for each x in localTop100
    emit(linkId ,pageRankValue)
}
```

Mapper for mapping File

```
// This mapper takes each link emits(linkId, linkName)
Map(…, Line){
    Split the line to obtain linkId and linkPageName
    emit(linkId,linkPageName)
}
```

Reducer:

```
// Map : LinkName => PageRank
ranksMap

Setup(){
    Initialize ranksMap;
}

// checks if the linkId receives two values indicating it appears in local top100 of any of the mappers.
reduce(link,[value1,value2])){
    // value1, value2.. refers to inputlist
    if (size of the input list is 2)
    add (linkName,pagerank) to ranksMap
}

// sorts and emits the top 100
cleanup(){
    sort the ranksMap by pagerank
    emit top 100 records of ranksMap.
}
```

Briefly explain how each matrix and vector is stored (serialized) for versions A and B. Do not just paste in source code. Explain it in plain English, using a carefully chosen example. See the above discussion about dense and sparse matrices for an example how to do this.

In both the versions the sparse matrix representation is used. In this matrix representation, the row indicates the outlinks associated with that node(represented by the row number).

Let assume this is the example:

A -> B, C
D -> A

Mapping
A – 0
B – 1
C – 2
D – 3

Matrix M

0 [1 2]
3 [0]

0 is the node id and it has two outlinks, 1 and 2. Thus, the rows in this representation represent the outlinks associated with 0.

Matrix R
0 [0.25]
1 [0.25]
2 [0.25]
3 [0.25]

This matrix contains all the nodes containing the pagerank values. There will be only one column. And no of rows will be equal to TOTAL no of nodes in the graph.

Matrix D
1 [0.25]
2 [0.25]

This matrix represents the dangling nodes and there page rank values. There will be only one column.

Discuss how you handle dangling nodes. In particular, do you simply replace the zero-columns in   and then work with the corresponding ', or do you deal with the dangling nodes separately? Make sure you mention if your solution for dangling nodes introduces an additional MapReduce job during each iteration. (5 points)

Dangling nodes can be handled in different ways. If we choose dense representation of matrix, then we can get hold of all the nodes that have all the columns as 0 and replace them with 1/V. Then multiply this matrix with R. which will give the share that this dangling node will be contributing to all the nodes in the graph. But, the disadvantage of this approach is, since the matrix is dense, for each dangling node, it must emit V number of records thus it is going to take a lot of time.

Second approach, is to take advantage of the sparse representation. We already have a D matrix that only contains the dangling nodes. And, we have R matrix that contains Pagerank values of all the nodes, that includes these dangling nodes. So, if we multiply this D with R we get the sum of the total dangling score that should be shared across all the nodes.

So, I have used the second approach. Dangling score computation is a separate Map Reduce job. In this approach, I compute the D x R using the algorithm mentioned in the module. The D matrix is going to emit (row, "D", col, page_rank_share) and R will emit (col,"R",row, value). Thus in reducer, for the dangling nodes, I receive two records (multiply the values of two records) which indicates the pagerank of that dangling node that should be shared with the all the nodes in the graph. If only one value is received, then ignore that reducer record. The value obtained is updated to the counter and at the end of this job is the value that each of the node in the graph will get as a page rank share of all the dangling nodes in the graph. This value can be set as a configuration and will be used in the Pagerank computation.

Performance Comparison

Version A:

6 Machines:

Step 1 & 2	: 21.35 minutes
Step 3	: 54.12 minutes
Step 4	: 0.77 minutes

11 Machines:

Step 1 & 2	: 10.68 minutes
Step 3	: 37.60 minutes
Step 4	: 0.77 minutes

Version B:

6 Machines:

Step 1 & 2	: 17.17 minutes
Step 3	: 19.52 minutes
Step 4	: 0.83 minutes

11 Machines:

Step 1 & 2	: 11.15 minutes
Step 3	: 17.12 minutes
Step 4	: 0.75 minutes

Both the versions use the original input files, parse it and then run the 10 iterations. Thus, the above time includes the time involved in preprocessing by parsing the input files.

Comparison

	Adjacency Lists	Matrix – Version A	Matrix – Version B
6 – Machines	32 minutes	54.12 minutes	19.52 minutes
11-Machines	19 minutes	37.60 minutes	17.12 minutes

Explanation:
The Matrix version takes much less time when compared to the Adjacency list version.

In case of Adjacency list representation, we were storing the actual page names, but in case of matrix representation, we are using sparse matrix. So the amount of data that is transferred between the mapper and reducer is less. The data that is written in Map reduce for each iteration is somewhat 5864518163 bytes, in case of matrix this is much less (44589096 + 157615326) bytes.

In case of Row By column versions, I am using distributed cache to refer to the rank values. As, this will reduce a complete job, the time taken in case of this approach is much less when compared to that of Hadoop version.

Incase of the Hadoop version, the size of the intermediate files were also large, because we are using sparse representation in this approach, the size of the intermediate files is also less.

Top-100 records

Top 100 pages for both versions on both machines are attached with the submission.

The Hadoop version of top-100 records have the highest value as 0.0023, while this version has the highest value of 0.0019.

In case of the Hadoop version, the preprocessing job does not identify all the dangling nodes. The dangling nodes are added to the node list after the first iteration. Thus, in Hadoop version, the page rank of dangling nodes of the previous iteration was considered for the current iteration.

In this version, the preprocessing job collects all the dangling nodes and thus we have all the nodes in the graph. Thus, the dangling nodes page rank values are calculated and the same iteration and the same value is used for the page rank computation. Thus, there is a small change in the values of the pages. But, mostly the page names are the same.