Parallel Matrix Multiplication

Zexi Han, Chengyang Chen
https://github.ccs.neu.edu/cs6240f18/Voyager

December 9, 2018

Abstract

In this project we studied the different parallelization mechanisms for matrix multiplication. The regular structure of matrices makes it comparably easy to identify opportunities of parallelism. Horizontal-Vertical Partitioning and Vertical-Horizontal Partitioning were implemented and tested for both dense and sparse matrix multiplication in MapReduce. The goal is to measure the speedup and observe the scalability of different approaches, so as to determine the best approach for the matrix multiplication.

1 Input Data

Synthetic square matrices were used as the input data. Dense matrix are stored in rows in the format of (matrixID,rowID,arrayOfValues), while sparse matrix are stored in rows/cols in the format of (matrixID,rowID/colID,[(colID/rowID:val),...]) where [(colID/rowID:val)...] are sorted by colID/rowID. The sparsity is defined as the percentage of cells whose value is zero in a matrix. For example, $s = 0.9$ indicates that a sparse matrix has 90% of cells whose values are zeros.

2 Horizontal-Vertical Partitioning

2.1 Overview

The simple solution for H-V partitioning is to partition matrix M_A into rows and use distributed file cache to send the whole copy of matrix M_B to each reducer. In a single reduce function call, perform the matrix multiplication on a row from M_A and cols from M_B. However, the solution will not work if the matrix M_B is too big to be stored in the memory. In addition, if matrix M_A has only one row and M_B has tens of thousands rows, all computations will be performed on a single reduce all which result poor parallelization. Therefore, we decide to use 1-Bucket-Random algorithm to do matrix multiplication.

Job1: Use 1-Bucket-Random algorithm to compute row-by-col matrix multiplication. The job will take two matrices stored in rows and cols and compute the matrix multiplication and emit the result. Each record in the result represents rowID, colID, val for a single cell in the result matrix.

Job2: Use Secondary Sort to merge cell records to rows with list of (colID:val) sorted by colID.

2.2 Pseudo-Code

2.2.1 Job 1: Use 1-Bucket-Random algorithm to do matrix product

```
// randomly assign row and cols to different regions.
map(partition, ID, cells = [rowID/colID:val ...]) {
    if (isRow(partition))
        regionKey = randomInt(0,A-1)
        // Emit the cell for all regions in the selected "row"
        for (int regionKey = (row * B); regionKey < ((row + 1) * B); regionKey++)
            emit(regionKey, (partition, ID, cells = [rowID/colID:val ...]))
    else
        regionKey = randomInt(0,B-1)
        // Emit the cell for all regions in the selected "col"
        for (int regionKey = col; regionKey < (A * B + col); regionKey += B)
            emit(regionKey, (partition, ID, cells = [rowID/colID:val ...]))
}

//emit matrix multiplication result for given row and col
reduce(regionKey, values = [(partition, ID, cells = [rowID/colID:val ...])...]) {
    do loop through values to build rows and cols list.
    // loop though
    for each row in rows
        for each col in cols {
            val = dotProduct(row,col)
            if val != 0
                emit(row.getID, colID, val)
        }
}

// Since all cell in row and col are soted by colID/rowID, use two pointer to scan each
    list and calculated the dot product.
dotProduct(row, col) {
    row_ptr = 0; // row pointer
    col_ptr = 0; // col pointer
    sum = 0;
    while row_ptr < row.size && col_ptr < col.size {
        if row_ptr < col_ptr
            row_ptr++
        else if row_ptr > col_ptr
            col_ptr++
        else
            // cells have the same index
            sum += (row_ptr * col_ptr)
    }
    return sum
}
```

2.2.2 Job 2: Use secondary sort to merge the cell in rows and sort the cell by colID

- map, emit key = (rowID,colID), value = (cellValue)
- partition, partition on rowID only
- keyComparator, sorts on rowID first then colID
- groupingComparator, consider rowID only

- reduce, build the list from input values and emit the result.

2.3 Algorithm and Program Analysis

For sparse matrices, some regions may have dense cells, rows, cols. The randomization of the 1-Bucket-Random algorithm will effectively removes output skew. To determine the number of partitions for the algorithm, We have following [1]:

Input = two $|n \times n|$ square matrix
A = # horizontal partition
B = # vertical partition
r = # worker machine
$|S|$ = # row in matrix $M_A = n$
$|T|$ = # col in matrix $M_B = n$
$C = |S|/|T|$
If $r|S| < |T|$ that is $|S| < |T|/r$ (S is way smaller than T) then, $A = 1$ and $B = r$
Otherwise, $A = \lfloor \sqrt{C \cdot r} \rfloor, B = \lfloor \sqrt{C^{-1} \cdot r} \rfloor$

In our experiments, matrix M_A and matrix M_B are both square matrix, which means that $|S| = |T|$, that is $C = 1$. Therefore, $A = B = \lfloor \sqrt{r} \rfloor = \lfloor \sqrt{\#WorkerMachines} \rfloor$

2.4 Experiments

2.4.1 Speedup

Input Data Size	Cluster Size	Running Time
Sparse $s = 0.9$ $n = 5,000$	m4.large 4 workers	12 min
	m4.large 9 workers	6 min

Table 1: H-V - Speedup Test

The speed-up of the matrix multiplication of two $|5000 \times 5000|$ matrices, on 4 workers and 9 workers is $12/6 = 2$. The theoretical max speed-up is $9/4 = 2.25$. The experiment measured speed-up is pretty close to the optimal speed-up.

2.4.2 Scalability

Input Data Size	Cluster Size	Running Time
Sparse $s = 0.9$ $n = 5,000$	m4.large 9 workers	6 mins
Sparse $s = 0.9$ $n = 7,000$		14 mins

Table 2: H-V - Scalability Test

Change the size of the square matrix from $n = 5k$ to $n = 7k$. As the result, 1.2 times the size of input matrix cause the running time increase from 6 mins to 14 mins. That is 2.33 times running time increased. Therefore, the program does not scale well with large size of input. One of the reason could be that the nature of Matrix multiplication algorithm, the running time of the algorithm is $O(n^3)$, which indicates that changing the size of input will cause the cubical increasing of the running time.

2.4.3 Result Sample

The result has the same format as the input data. Each record which represents a single cell in the matrix is in the following format: dense ouput (matxiID, rowID, arrayOfValues) and sparse output (matxiID, rowID, [colID:val, ...]). Below is a sample of the sparse result:

```
// 1-Bucket-Random matrix multiplication result
// Each record has rowID, colID, val
0,1,2
0,2,33
1,2,35
...
// Secondary sort result
// matrix are in rows, with list of (colID:val) sorted by colID
result,0,0:28,1:2,2:33,3:40
result,1,1:30,2:35,3:40
...
```

3 Vertical-Horizontal Partitioning

3.1 Overview

Matrix multiplication can also be parallelized by partitioning the first matrix by column, and the second by row. Note that the value in row i and column j of the result matrix is equal to $A[i,0] \times B[0,j] + A[i,1] \times B[1,j] + ... + A[i,r_a] \times B[c_b,j]$, where r_a and c_b are the number of As rows and Bs columns, respectively. (These numbers have to be identical for the matrix product to be defined.) Hence the product of column vector $A[*,k]$ and row vector $B[k,*]$ produces the individual terms of type $A[*,k] \times B[k,*]$ needed for the summation [1].

3.2 Pseudo-Code

Since processing different formats of input and output is the only difference between the V-H Partitioning programs of dense matrix product and sparse matrix product, only pseudo code of the sparse matrix product is shown in the report as below.

3.2.1 Job 1: V-H Partitioning

```
// Sparse Input: matrix cell in the following format (matrixID,rowID,[colID:val,...])
map(matrixID, rowID, [colID:val, ...]) {
   for (colID,val) in input list
      if (matrixID = A) // Partition A into columns
         emit(colID, (matrixID, rowID, val))
      else // Partition B into rows
         emit(rowID, (matrixID, colID, val))

// Reduce receives entries A[i,k] and B[k,j] for different i and j
reduce(common_A_col_B_row, [(matrixID, index, val), ...]) {
   for each (matrixID, index, val)
      if (matrixID = A) then
         A_list.add(index, val)
      else
```

```
            B_list.add(index, val)
    // Emit all products A[i,k]*B[k,j] with key (i,j), because this is
    // the contribution for result cell [i,j].
    for each Aik in A_list
        for each Bkj in B_list
            emit((Aik.index, Bkj.index), Aik.val * Bkj.val) }
}
```

3.2.2 Job 2: Matrix Sum

```
// Input: intermediate output of job 1: (Aik.index,Bkj.index,Aik.val*Bkj.val)
map(rowID, colID, val) {
    emit((rowID, colID), val)
}

// Combiner is set as the same as reducer

reduce((rowID, colID), [val1, val2, ...]) {
    sum = 0
    for val in input list
        sum += val
    emit((rowID, colID), sum)
}
```

3.2.3 Job 3: Matrix Format Output

Convert (rowID,colID,val) to (“result”,rowID,[colID:val,...]). To group same-row values, we use rowID as key. But we also want the colID:val is output in ascending order of colID in each row, so the best solution is to use secondary sort. Since the algorithm doesn’t scale to very large input and the maximum endurable number of values in a row is just a few thousand, we sort colID locally in each reduce function call instead. We also notice that the result of the sparse matrix multiplication can be either sparse or dense. Thus it is necessary to choose a proper format of the output to store the result according to the sparsity of the result matrix. As time is limit, we leave this to the future work. Since this job is not the main task, its pseudo code is omitted in the report.

3.3 Algorithm and Program Analysis

The algorithm first performs a distributed equi-join of any $A[i, x]$ with any $B[y, j]$, using condition $x = y$. The corresponding product is emitted with key (i, j). Generally, if A is $m \times k$ and B is $k \times n$, there will be k partitions. The consequence is that if k is really small, the efficiency of the parallel program will be close to the sequential one which is certainly undesirable. Another MapReduce job then processes the join output by grouping by key and adding all values in each group.

From Table 3 we can clearly see that the drawback of the algorithm is that the huge amount of intermediate records produced by the equi-join have to be written to and read from S3, which costs a lot of time and disk space. On the other hand, the strength of the algorithm is that it applies to both dense and sparse matrix multiplication tasks. Taking sparse matrices as input, the mapper of the first job only emits non-zero $A[i, x]$ or $B[y, j]$, and hence no redundant product ($A[i, x] \times 0$ or $0 \times B[y, j]$ or 0×0, $A[i, x]$ and $B[y, j]$ are non-zero) is performed in the reducer.

V-H Input	V-H Shuffle Output	V-H Reduce Output (Sum Input)	Sum Shuffle Output	Sum Reduce Output
Dense $n = 500$ 1,000	500,000	125,000,000	6,750,000	250,000
Dense $n = 1,000$ 2,000	2,000,000	1,000,000,000	189,000,000	1,000,000
Dense $n = 2,000$ 4,000	8,000,000	8,000,000,000	5,996,874,253	4,000,000
Sparse $s = 0.9$ $n = 1,000$ 2,000	198,144	9,813,216	5,996,639	999,947
Sparse $s = 0.9$ $n = 2,000$ 4,000	791,465	78,303,325	47,574,963	4,000,000
Sparse $s = 0.9$ $n = 4,000$ 8,000	3,169,216	627,716,862	546,367,773	16,000,000

Table 3: V-H - Cardinality of V-H Partitioning and Matrix Sum in records

3.4 Experiments

3.4.1 Speedup

Input Data Size	Cluster Size	Running Time
Dense $n = 1,000$ 5.8 MB	m4.large 5 workers	24 min
	m4.large 10 workers	13 min
Sparse $s = 0.9$ $n = 4,000$ 24.2 MB	m4.large 5 workers	21 min
	m4.large 10 workers	13 min

Table 4: V-H - Speedup Test

So Speedup$_{Dense} \doteq 24/13 = 1.85$ and Speedup$_{Sparse} \doteq 21/13 = 1.62$. Since the theoretically best possible speedup is $10/5 = 2$ for the above setting, the speedup of our V-H Partitioning for both dense and sparse matrix multiplication is close to the ideal scenario.

3.4.2 Scalability

Input Data Size	Cluster Size	Running Time
Dense $n = 500$ 1.5 MB	m4.large 5 workers	4 min
Dense $n = 1,000$ 5.8 MB		24 min
Dense $n = 2,000$ 23.2 MB		3 h 43 min
Sparse $s = 0.9$ $n = 1,000$ 1.4 MB		2 min
Sparse $s = 0.9$ $n = 2,000$ 5.8 MB		4 min
Sparse $s = 0.9$ $n = 4,000$ 24.2 MB		21 min

Table 5: V-H - Scalability Test

From the table we can easily notice that the running time of the dense matrix multiplication jumps from 24 min to 3 h 43 min while the dimension of the input matrix is just doubled. We also experimented on sparse $s = 0.9$ $n = 10,000$ matrix multiplication, the running time of the first job exceeded 11 hours so we terminated the cluster. This is owing to the reason that the V-H Partitioning requires writing and reading a huge amount of intermediate output from S3. Therefore, this partitioning does not scale well.

3.4.3 Result Sample

```
// Job 1 V-H Partitioning result
// Each record has rowID, colID, val
310,338,1600
310,366,672
310,367,2044
...
// Job 2 Matrix Sum result
// Each record has rowID, colID, val
0,0,40924
0,1,12546
0,10,19197
...
// Job 3 Matrix Format Output result
// matrix are in rows, with list of (colID:val) sorted by colID
result,0,0:40924,1:12546,2:28450,3:12251,...
result,1,0:5319,1:9343,2:21396,3:19411,...
...
```

4 Conclusion

For a square matrix with thousands of rows and columns, the H-V Partitioning with 1-Bucket-Random algorithm run generally faster than V-H approaches. The reason is that V-H Partitioning might create lots of intermediates results that need to be write to S3 and read back as input to the post-processing job. Reading and writing a large amount of intermediates might cost a lot of time. Also, the number of partitions of V-H relies on the dimension of the input matrices, which is obviously undesirable for a parallel program. In contrast, H-V partitioning with 1-Bucket-Random algorithm does create duplicate copies of input. Choosing an appropriate number of partitions for 1-Bucket-Random algorithm based on input matrix properties and the number of workers available does give better performance than the V-H approach.

For further analysis, we can create more test runs changing different variables such as, increasing/decreasing sparsity, determining partition numbers with experimental result, increasing size of input, increasing number of workers, etc, to see how different matrix properties and Cluster configurations affect the performance of H-V and V-H approaches.

References

[1] M. Riedewald, *Lecture Slides: Intelligent Partitioning*, (Northeastern University, Boston, 2018).