A Map Reduce Big data Approach for Sparse Matrix Data Processing

Somasekhar Giddaluri1, Dr. K. Karthikeyan2,
1Research Scholar, 2Associate Professor
1School of Computer Science & Engineering
2School of Advanced Sciences
1, 2VIT University, Vellore 632014, India.
kkarthikeyan67@yahoo.co.in
+91-9944943681

October 14, 2018

Abstract

The importance of big data is increasing day by day motivating the researchers towards new inventions. Often, a small amount of data is needed to achieve a solution or to draw a conclusion. The new big data techniques stem from the necessity to retrieve, store and process the required data out of huge data volumes. The present paper focuses on dealing with sparse matrices which is frequently needed in many sparse big data applications nowadays. It applies compact representation techniques of sparse data and moulds the required data in the mapreducible format. Then the mapreduce strategy is used to get the results quickly which saves execution time and improves scalability. Finally we established that the new algorithm performs well in sparse big data scenario compared to the state-of-the-art big data processing techniques.

Key Words: Sparse data; Sparse matrix; Matrix multiplication; Big data; Mapreduce.
1 Introduction

Nowadays, due to the evolution of social networking websites, mobile devices, sensors and cloud computing, the data is rapidly being incremented from time to time. This enormous data for which traditional RDBMS techniques become inapplicable is termed as big data. Big data processing and big data analysis are the challenges faced by the society today. New techniques, algorithms, and solutions in big data domain are the most desirable now. In recent past, sparse big data attracted the research community to innovate several big data techniques for its retrieval, storage and efficient processing. Compact storage options for sparse columns were proposed by Abadi D.J.[1]. The compact sparse matrix representation techniques for GPU architectures were proposed by Neelima B. and Prakash S.R. [2]. The above two compact representation techniques save data storage space, and reduce data retrieval time. A fast sparse matrix multiplication technique was proposed by Yuster R. and Zwick U.[3]. This technique partitions the matrices to be multiplied into a dense part and a sparse part. It uses a fast algebraic algorithm to multiply the dense parts, and the naive algorithm to multiply the sparse parts. It focussed on minimising the number of arithmetic operations involved in sparse matrix multiplication. But it is having only theoretical value. Many big data processing techniques were brought into limelight by Dean J., Ghemawat S. and White T. [4,5] which can also be used in big sparse data applications. The challenging problems in the Big Data Research were identified by Xiaolong, J. et al.[6]. A mapreduce based solution in the health care sector was given by Mohammad Hossein, B., and Mahdi, N.[7]. A scalable data mining technique based on mapreduce was proposed by Xuan, L. et al.[8]. Parallelisation and indexing techniques for sparse matrix multiplication were implemented by Buluc A. and Gilbert J.R [9]. The communication overhead problem of sparse matrix multiplication was solved by Ballard G. et.al. [10]. The parallelisation technique for sparse tensor matrix multiplication was proposed by Smith S. et.al. [11]. The approaches [9-11] are not suitable for big data applications. Proper care should be taken by the programmer regarding the data distribution, replication, load balancing, communication overhead etc. Some mapreduce based massive matrix multiplic-
tion techniques were innovated[12-15]. Though the HAMA based iterative approaches [12-13] exhibit good scalability over large data sets, they take multiple rounds for matrix multiplication. In the article[14], an efficient solution for matrix chain multiplication was proposed by Myung J. and Lee S., giving more importance to inter-operation parallelism than intra-operation parallelism during matrix multiplication. Here, matrix is represented as a relation. But this representation has redundancy problem. More memory space is needed to store each input sparse matrix which increases the data retrieval time. This results in more time for multiplication. Though multi-round matrix multiplication [15] is suitable for long running mapreduce computations in cloud systems, the management of input/output pairs in each round is a complex issue. The subsequent rounds will spend much time to read temporary files generated by the previous round. This results in extra overhead. The algorithm sparseMULovercomes the drawbacks of state-of-the-art approaches in dealing big sparse matrices. It focuses on reduction in matrix multiplication time and improvement in scalability in the sparse Big Data scenario. It uses the compact representation techniques of sparse data, converts them into a mapreducible format and performs a sparse matrix multiplication with less execution time. The results prove that the algorithm works better compared to massive sparse matrix multiplication using HAMA Hadoop and HAMA_HPMR[12-13]. It is more suitable for big sparse data applications.

2 Problem statement

In general, sparse data consists of a large number of missing values which are not useful in decision making process. If the original sparse data is stored in the main memory, much of the memory goes waste by missing values. So at the pre-processing stage, it must be converted into a compact form to save memory space and to make the computation process easy. But these compact sparse data representations must be converted into a mapreducible format to make them suitable for big sparse data applications and efficient big data algorithms must be implemented to get the fast results with optimal scalability. This paper focuses on sparse matrix data
and its processing. The compact sparse matrix representations and fast sparse matrix computations are discussed with a case study on sparse matrix multiplication in the big data perspective.

3 Problem Solving and Innovative Content

Two compact representations of a sparse matrix row are presented below.

Figure 1. Positions represented using a list (Compact sparse row representation1)

![Figure 1](image1)

Figure 2. Positions represented using a bit string (Compact sparse row representation2)

![Figure 2](image2)

These compact representations (figure1-2) save memory. Compact sparse row representation2 saves more memory space than compact sparse row representation1. At present, we experimented on compact sparse row representation1 with a focus on execution time and scalability. The most popular big data technique known as mapreduce is used here. Mapreduce is a programming strategy (figure 3)
which is best suited to the big data applications where less execution time and more scalability are essential. The big input data set is first partitioned, then sent to fixed number of map functions as input and finally processed in parallel.

Figure 3. The implementation of Mapreduce programming strategy

The intermediate outputs (Local outputs) of map functions are collected as one unit and sent to each reducer function as input. The total job consists of split, sort, and merge operation sequence. Finally, the outputs from all reducer functions are collected as one final output file. SparseMUL uses this strategy to solve the big data issue. Moreover, any big data solution must satisfy the following three requirements.

i) All the data should be distributable.

ii) The global pattern (Final output) should be obtained from all the local patterns (Local outputs).

iii) The problem should be mapreducible.

So, at the pre-processing stage, each row of sparse matrix using sparse row representation is converted into map reducible format as shown in figure 4.

Matrix $M$:

Row #1:

| 1 | 0 | 0 | 0 | 2 | 0 | 1 |

Column Identification Number (non-null position in the row)

Matrix Name

$M_1, 3, 4$

$M_2, 5, 6, 7$

$M_3, 8, 9, 10$

$M_4, 11, 12, 13$

$M_5, 14, 15$

(Note: Five lines are stored for five non-null values in the row #1 of matrix $M$. Start position, end position, and number of non-null values in each row are stored explicitly. To save memory space, but size of the matrix is stored explicitly. All rows are represented in the same way.)
Figure 4. A map-reducible format of a row in the sparse matrix M.

To simplify the problem, instead of taking two input files for the pair of sparse matrices to be multiplied, only a single input file is created by appending the map-reducible format of the second input sparse matrix to that of the first input sparse matrix in the same file. The steps in the main big data algorithm SparseMUL( ) are as shown below.

```plaintext
File SparseMUL(File D) // Algorithm SparseMUL

Input: The original sparse matrices M1 and M2;

Output: The target data file F;

1: Arrange matrices A and B in the map-reducible format shown in fig. 4 and store the resultant matrices data in D.
2: F= sparseMatrixMatrixMultiplicationOneStep(D); // Initiates mapreduce job;

File MAP_ sparseMatrixMatrixMultiplicationOneStep(File D) // Map task;

Input: A source data file D;

Output: The intermediate file D_PART;

A1: for each line in D do
A2: str = line.split(‘,’);
A3: if str[0] = ‘M1’ then
A4: for j = 0 ..ndo
A5: Key = str[1] + "", "j";
A6: Value = M1["", "i" + str[2] + "", "j" + str[3];
A7: context.write(Key, Value); // Writing line to D_PART
A8: end for
A9: end if
A10: if str[0] = ‘M2’ then
A11: for j = 0 ..ndo
A12: Key = j + "", "+str[2];
A13: Value = M2["", "i" + str[1] + "", "j" + str[3];
A14: context.write(Key, Value); // Writing line to D_PART
A15: end for
A16: end if
A17: end for
```
Pseudo code for the SparseMUL approach

```java
File REDUCE SparseMatrixMatrixMultiplicationOneStep (File D_INBOX) // Reduce task;

Input: D_INBOX = Collection of all D_PAR files.
HashMap<Integer, Float> hashM1 = new HashMap<Integer, Float>();
HashMap<Integer, Float> hashM2 = new HashMap<Integer, Float>();
Float result = 0.0;
Float a_i, b_jk;

Output: RD_PAR = The output of a reduce task;

R1: for each line “Key, Value(Iterable)” do // grouped by key;
R2: str1 = value.toString().split(“,”);
R3: if str1[0].equals(“M1”) then
    R4: hashM1.put(Integer.parseInt(str1[1]), Float.parseFloat(str1[2]));
else
    R6: hashM2.put(Integer.parseInt(str1[1]), Float.parseFloat(str1[2]));
end if
end for
R8: end for
R9: for j=0..n do
    R10: a_ij = hashM1.containsKey(j) ? hashM1.get(j) : 0.0f;
    R11: b_jk = hashM2.containsKey(j) ? hashM2.get(j) : 0.0f;
    R12: result += a_ij * b_jk;
end for
R14: if result != 0.0 then
    R15: context.write(null, new Text(Key.toString()+”t”+Float.toString(result)));
end if
```

The Cloudera Quick Start VM 5.5.0 virtual machine environment with pseudo distributed Hadoop 2.6.0, and other ecosystem tools like HBase, Pig, Hive etc., is used for experiments.

The results in the following section prove that the proposed approach shows better execution time and scalability compared to the big sparse matrix multiplication approaches using the recent tools HAMA_Hadoop and HAMA_HPMR [11, 12].

4 Results and Comparison

Table 1. Execution times of various sparse matrix multiplication approaches
SparseMUL is executed on single node Hadoop-pseudo distributed cluster environment with 1% sparse matrices having dimensions varying from 32 to 320. Similarly sparse matrix multiplications with HAMA_Hadoop and HAMA_HPMR are implemented in the same environment. On average, SparseMUL shows approximately 2.15 times and 1.96 times reduction in time complexity compared to HAMA_Hadoop and HAMA_HPMR respectively.

The execution times of different sparse matrix multiplication approaches are tabulated in Table 1. Though SparseMULs initial execution time for matrix dimension 32 is more, it takes less execution time for the next remaining matrix dimensions. The execution time of SparseMUL for each matrix dimension is calculated by deducing job start time from job finish time.

Scale up is calculated by using the following formula,

$$\text{scaleup}(\text{dimension}) = \log\left(\frac{T(\text{dimension})}{T(32)}\right)$$

(1)

(Here, T denotes the execution time).

Scale up is inversely proportional to the scalability.

The scalability improvement of SparseMUL compared to sparse matrix multiplication using the tools HAMA_Hadoop and HAMA_HPMR is tabulated in Table 2.

<table>
<thead>
<tr>
<th>Matrix dimension</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HAMA_Hadoop</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>85</td>
</tr>
<tr>
<td>128</td>
<td>102</td>
</tr>
<tr>
<td>192</td>
<td>131</td>
</tr>
<tr>
<td>256</td>
<td>181</td>
</tr>
<tr>
<td>320</td>
<td>228</td>
</tr>
</tbody>
</table>

Table 2. Scale up values of various sparse matrix multiplication approaches

<table>
<thead>
<tr>
<th>Matrix dimension</th>
<th>Scale up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HAMA_Hadoop</td>
</tr>
<tr>
<td>32</td>
<td>0.0</td>
</tr>
<tr>
<td>64</td>
<td>0.80</td>
</tr>
<tr>
<td>128</td>
<td>0.86</td>
</tr>
<tr>
<td>192</td>
<td>0.875</td>
</tr>
<tr>
<td>256</td>
<td>1.12</td>
</tr>
<tr>
<td>320</td>
<td>1.22</td>
</tr>
</tbody>
</table>
5 CONCLUSION

An efficient big data algorithm to deal with sparse matrices is proposed and supported with a case study on sparse matrix multiplication in this paper. Many big sparse data applications such as genomics applications, analysis of protein mass-spectrometry data, sparse brain imaging applications, sequential testing approaches, algorithmic aspects of sparse recovery, learning sparse latent models and so on may use this algorithm. It is more suitable for the big data applications showing better results in terms of scalability and execution time compared to the big matrix multiplication approaches of HAMA_Hadoop and HAMA_HPMR in sparse data scenarios. There are some future research directions possible in this problem domain. SparseMUL may be combined and implemented with HAMA_Hadoop or HAMA_HPMR to get significant improvement in the performance of sparse matrix computation. The compact sparse row representation 2 can be implemented wherever needed to deal with sparse matrices. The big data algorithms with compact representations are more desirable to improve the performance of sparse matrix data processing. Moreover, SparseMUL may be further developed to perform matrix chain multiplication. The implementations of SparseMUL with Spark and HBase are other possible research directions. The big data research needs to be encouraged in this problem domain.

References


