BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **17**, No 1
Sofia • 2017 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2017-0002

Fast Matrix Multiplication with Big Sparse Data

G. Somasekhar[1], *K. Karthikeyan*[2]

[1]*School of Computer Science & Engineering, VIT University, Vellore, India*
[2]*School of Advanced Sciences, VIT University, Vellore, India*
Emails: gidd.somasekhar2014@vit.ac.in k.karthikeyan@vit.ac.in

Abstract: *Big Data became a buzz word nowadays due to the evolution of huge volumes of data beyond peta bytes. This article focuses on matrix multiplication with big sparse data. The proposed FASTsparseMUL algorithm outperforms the state-of-the-art big matrix multiplication approaches in sparse data scenario.*

Keywords: *Sparse data, sparse matrices multiplication, Big Data, Mapreduce.*

1. Introduction

Big Data analytics and its applications attracted researchers leading to many inventions. While analysing the data, a small amount of data may be required for drawing conclusions, taking decisions or achieving the solution. As sparse data consists of large number of missing values or null values which are not useful in data analysis, the key is to store only the non-null values of it. When the sparse data becomes voluminous, so that we cannot apply any of the traditional database techniques to reach the objective, then it is known as big sparse data. An efficient sparse matrix representation and it's usage to solve the big matrix multiplication problem in sparse data scenario is our main theme. Operation with the pair of big sparse matrices, used as input in sparse matrices multiplication, involves the problems of data representation, storage, retrieval and processing. Researchers had given many solutions to solve them. The data structures for the compact representation of sparse matrices were invented by Di Felice, Agnifili and Clementini [1]. Compact storage options for sparse columns were proposed by Abadi [2]. The suitable sparse matrices representation techniques for GPU architectures were proposed by Neelima and Prakash [3]. The main advantages of the above three compact representation techniques are saving data storage space, and reducing data retrieval time. A fast sparse matrices multiplication technique was proposed by Yuster and Zwick [4]. 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 matrices multiplication. But it is having only theoretical value.

Many Big Data processing techniques were brought into limelight by Dean and Ghemawat [5] and White [6] which can also be used in big sparse data applications. Parallelisation and indexing techniques for sparse matrices multiplication were implemented by Buluc and Gilbert [7]. The communication overhead problem of sparse matrices multiplication was solved by Ballard et al. [8]. The parallelisation technique for sparse tensor matrix multiplication was proposed by Smith et al. [9]. The above approaches [7-9] 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. Several mapreduce based techniques applicable in many big sparse data scenarios were innovated [10-15]. A Big Data solution for matrix factorization was proposed by Sun, Li and Rishe [10]. It involves more computation cost. Though the HAMA based iterative approaches [11-12] exhibit good scalability over large data sets, they take multiple rounds for matrix multiplication. An efficient solution for matrix chain multiplication was proposed by Myung and Lee [13], 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. The Vector Linear Combination scheme was proposed by Zheng et al. [14]. It splits matrix multiplication in two steps, namely scalar multiplication and linear combination of weighted vectors. It gives the result in a single mapreduce job. As any special input format or layout for sparse matrices is not taken prior to multiplication process, it leads to a little bit increase in multiplication time. 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.

Some serious attempts were made (ScaLAPACK [16] and DAGuE [17]) to make matrix computation easy and simple. But they failed to solve scalability issue. ScaLAPACK is difficult to program and incurs severe synchronization overhead. DAGuE does not implement any failure handling mechanism and its performance is limited due to tile level parallelism. With the pioneer work of Qian et al. [18], MadLINQ was emerged as a unified programming model for both matrix algorithm and application developers. It is efficient in dense matrices' computations. But it is difficult to handle sparse matrices using MadLINQ. These flaws in the above approaches motivated us to improve the matrix multiplication approach in the big sparse data perspective. As mapreduce has immense impact in real time Big Data applications, we selected mapreduce and improved the algorithm for big sparse data processing. We used compact sparse representations to save the memory space needed to store large sparse matrices. The results show that the proposed approach gives significant reduction in execution time and improvement in scalability. It overcomes the drawbacks of state-of-the-art approaches in operating with big sparse matrices.

2. Problem statement

The big sparse matrices multiplication involves a pair of sparse matrices to be multiplied. Let us assume the first input sparse matrix $A_{m\times n}$ and the second input sparse matrix $B_{n\times k}$, i.e., matrix A consists of m number of rows and n number of columns, whereas matrix B consists of n number of rows and k number of columns. If we take the raw sparse matrix data, a larger amount of multiplication time is wasted in retrieving and processing null values. So, an efficient combined sparse data representation of the pair of sparse matrices, and its implementation in the matrix multiplication from the point of view of Big Data are the major problems.

3. Problem solving and innovative content

The following two sparse row representations can be used in representing a sparse matrix as shown in Figs 1 and 2.

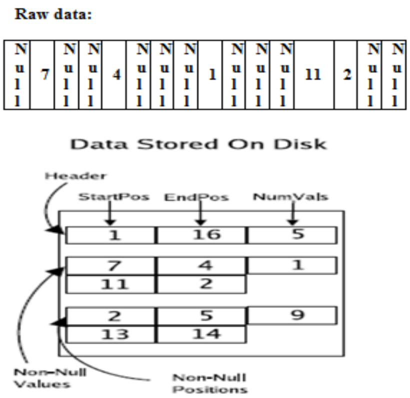


Fig. 1. Positions represented using a list (Compact sparse row representation #1)

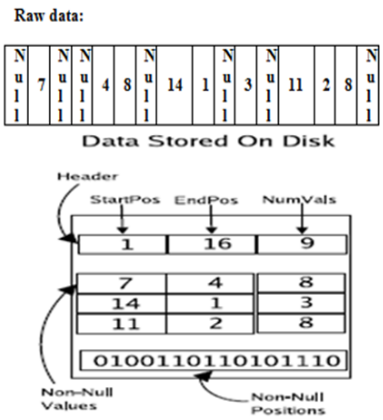


Fig. 2. Positions represented using a bit string (Compact sparse row representation#2)

Though these compact sparse data representations we solve the storage and retrieval problems to the maximum extent; the sparse matrices multiplication problem is yet to be solved in the Big Data scenario.

Any Big Data solution has to satisfy the following three requirements.

- All the data should be distributable.
- The global pattern (Final output) should be obtained from all the local patterns (Local outputs).
- The problem should be mapreducible.

Mapreduce is a programming strategy (Fig. 3) well suited to solve Big Data problems where less execution time and more scalability are essential. The big input data set is first partitioned and sent to fixed number of map functions as input and processed in parallel. 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. FAST sparse MUL uses this strategy to solve the big sparse matrices multiplication problem.

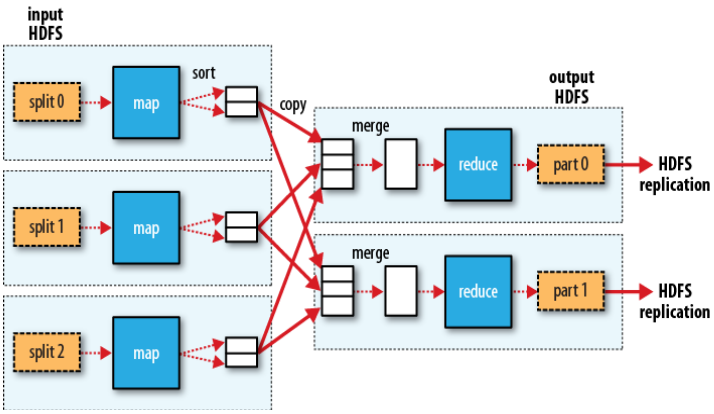


Fig. 3. The implementation of Mapreduce programming strategy

The problem is not mapreducible unless the compact sparse data representations of the two matrices involved in multiplication are converted into a mapreducible format. The sample mapreducible format of a sparse matrix row is shown below in Fig. 4.

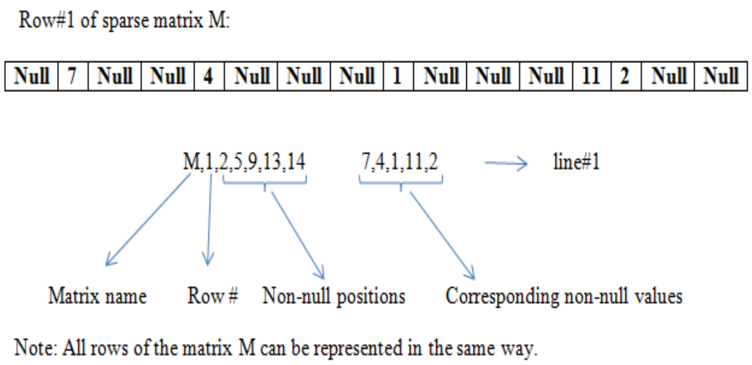


Fig. 4. The mapreducible format of a row of the sparse matrix *M* shown as a line in the input file

To simplify the problem further, instead of taking two input files for two sparse matrices which are to be multiplied, only a single input file is created by concatenating the mapreducible formats of respective matrices. This is implemented in the sub-algorithm called Combined_Sparse_Compact( ). The steps in the main Big Data algorithm FASTsparseMUL( ) are as shown below.

Pseudo code for the FASTsparseMUL approach

```
File  FASTsparseMUL (File D) // Algorithm FASTsparseMUL
Input: The original sparse matrices A and B;
Output: The target data file F;
1: D=Combined_Sparse_Compact(Matrix A, Matrix B); /* Converts matrix A and matrix B into
mapreducible compact format */
2: F= FAST_MR_sparseMUL(D); // Initiates mapreduce job;
```

```
File Combined_Sparse_Compact (Matrix A, Matrix B) // Algorithm Combined_Sparse_Compact
Input: The original matrices A and B;
File A consists of the original sparse matrix A of size m*n.
File B consists of the original sparse matrix B of size n*k.
Output: The target data file D; /* File D consists of the mapreducible compact form of both the
original sparse matrices A and B.*/
1: for   i = 0…m do
2:   str1=””;     // create  two empty strings.
3:   str2=””;
4:   str1+=”A, i”;
5:   for j = 0…n do
6:     if A[i] [j] = Null then // skips on reading null values of matrix A
7:       continue;
8:     else
9:       str1+=”j”;
10:      str2+=A[i] [j];
11:    end if
12: end for
13: line=str1+”\t”+str2; /* Conversion of each row of matrix A in the mapreducible compact
                            form as shown in Fig. 4.*/
14: Write line to file f1;
15: end for
16: for   i = 0...n do
17:   str1=””;     // create two empty strings.
18:   str2=””;
19:   str1+=”B, i”;
20:   for j = 0...k do
21:     if B[i][j] = Null then  // skips on reading null values of matrix B
22:       continue;
23:     else
24:       str1+=”j”;
25:       str2+=B[i] [j];
26:     end if
27:   end for
30: line=str1+”\t”+str2; /* Conversion of each row of matrix B in the mapreducible compact
                            form as shown in Fig. 4.*/
31: Write line to file f2;
32: end for
33: Concatenate f1 and f2 to get file D. // Collective compact representation of both matrices A and B
in a single file.
```

```
File FAST_MAP_sparseMUL (File D) // Map task;
Input: A source data file D;
Output: The intermediate file DPART ;
M1: for each line in D do
M2:   str = line.split (“\t”);
M3:  str1=str [0].split (“,”);
M4:  str2=str [1].split (“,”);
M5:  if str1 [0] = ‘A’ then
M6:     for j = 0…k do
M7:      for r = 0... (str2.length) do
M8:       Key = str1 [1] +”,”+j; // Representing matrix A in the (Key, Value) format.
M9:       Value = A+”,”+str1[r+2] +”,”+str2[r];
M10:       context.write (Key, Value);   // Writing line to DPART
M11:    end for
M12:   end for
M13:  end if
M14:  if str1 [0] = ‘B’ then
M15:     for i = 0…m do
M16:      for s = 0… (str2.length) do
M17:       Key = i+”,”+str1[s+2]; // Representing matrix B in the (Key, Value) format.
M18:       Value = B+”,”+str1[1]+”,”+str2[s];
M19:       context.write (Key, Value);   // Writing line to DPART
M20:      end for
M21:     end for
M22:  end if
M23: end for
```

```
File FAST_RED_sparseMUL (File  DUNION) // Reduce task ;
Input: DUNION = Collection of all DPART files.
HashMap<Integer, Float> hashA = new HashMap<Integer, Float> ( );
HashMap<Integer, Float> hashB = new HashMap<Integer, Float> ( );
Float result = 0.0;
Float a_ij, b_jk;
Output: RDPART = The output of a reduce task;
R1: for each line in DUNION  do
            // grouped by Key;
R2:     str1= Value.toString ( ).split(“,”);
// Implementing Hash Maps to store intermediate (Key, Value) pairs.
R3:      if str1 [0].equals (“A”) then
R4:       hashA.put (Integer.parseInt (str1[1]), Float.parseFloat (str1[2]));
R5:      else
R6:        hashB.put (Integer.parseInt (str1 [1]), Float.parseFloat (str1 [2]));
R7:      end if
R8: end for
/* Getting the intermediate (Key, Value) pairs from corresponding HashMaps and using them to obtain
the product matrix. */
R9: for j=0…n do
R10:   a_ij = hashA.containsKey (j)?hashA.get (j) : 0.0f;
R11:   b_jk= hashB.containsKey (j)? hashB.get (j): 0.0f;
R12:    result+= a_ij * b_jk;
R13: end for
// writing product matrix into the output file RDPART.
R14: if result! = 0.0 then
R15:  context.write (null, new Text (Key.toString () +”\t”+ Float.toString (result)));
R16: end if
```

The Cloudera Quick Start VM 5.5.0 virtual machine environment with pseudo -distributed mode Hadoop 2.6.0, and other eco system 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 sparse matrices multiplication approaches using HAMA_Hadoop, HAMA_HPMR [11, 12] and VLCA [14].

4. Results and comparison

Table 1. Analytical comparison of FASTsparseMUL with various matrix multiplication approaches in the big sparse data scenario

Approach/Algorithm	Advantages	Limitations
ScaLAPACK (HPC Solution)	High expressiveness	• Difficult to program • Problem size bounded by total memory size • Synchronization overhead
DAGuE (Tiles & DAG)	High expressiveness	• Programmer must annotate data dependencies explicitly • Problem size bounded by total memory size • Performance bound by parallelism at tile level • No failure handling
HAMA based iterative approach (MapReduce)	No constraint on problem size	• Takes multiple rounds for matrix multiplication
MadLINQ	• High expressiveness • No constraint on problem size	• Performance bounded by tile level parallelism, improved with block-level pipelining • Handling sparse matrices is very difficult and creates severe load imbalance
VLCA (MapReduce)	• No constraint on problem size • Reduction in execution time • Takes single mapreduce job	• No pre-processing to remove null values in the input sparse matrices • No use of any special format for input sparse matrices • No focus on null values in second input matrix • Unnecessary computation overhead which includes null values of second input matrix
FASTsparseMUL (MapReduce)	• No constraint on problem size • Maximum reduction in execution time • Takes single mapreduce job • Shows maximum scalability • Makes best use of a special format for input sparse matrices	• Pre-processing overhead • Mainly intended for sparse matrices multiplication • Application of the algorithm to dense matrices is yet to be studied

Table 1 shows comparative analysis of FASTsparseMUL with state-of-the-art matrix computation approaches in the big sparse data scenario. It compares with non-mapreduce based approaches as well as mapreduce based approaches. The non-mapreduce based approaches like ScaLAPACK and DAGuE do not solve scalability issue of matrix computation. Though MadLINQ shows a little bit improvement in scalability, it has difficulties in handling sparse matrices. In particular, MadLINQ creates severe load imbalance problem while processing big sparse matrices. Our main focus is on improving scalability and reducing execution time of big sparse matrices multiplication. For the moment, we are skipping the discussion of the above three approaches as they show deviation from the focused objectives. HAMA uses both iterative and block based approaches for matrix multiplication. As our focus is on sparse data case only and iterative approach of HAMA is better than its block based approach in sparse data applications, we compared the proposed algorithm with iterative HAMA approaches only. HAMA based iterative approaches take less execution time leading to further improvement in scalability. But they take multiple rounds to give the result. The iterative approach of HAMA requires N rounds for multiplying a matrix of size $N \times N$ [15]. Compared to HAMA based iterative approaches, FASTsparseMUL takes single round only. VLCA approach shows improvement in scalability and reduction in multiplication time. As it does not use any special format for input sparse matrices, there exists some multiplication time overhead. No pre-processing is performed in VLCA to remove null values and there is no focus on null values in second input matrix. In addition, it creates m number of copies of each null value present in each row vector of second input matrix. As a result, a significant number of additional multiplication operations are performed without considering the presence of null values in second input matrix. This incurs computation overhead and increase in multiplication time of sparse matrices. The proposed FASTsparseMUL makes best use of a special input format or layout for sparse matrices. It removes null values while pre-processing and avoids multiplication operations with null values to the maximum extent. It results in more reduction of execution time and improvement of scalability compared to HAMA based iterative approaches as well as VLCA approach.

Table 2. Execution times of various matrix multiplication approaches for sparse data

Matrix dimension	Execution time (sec)			
	HAMA_Hadoop	HAMA_HPMR	VLCA	FASTsparseMUL
32	16	16	12	41
64	85	71	48	37
128	102	101	69	35
192	131	115	79	42
256	181	172	103	47
320	228	202	125	52

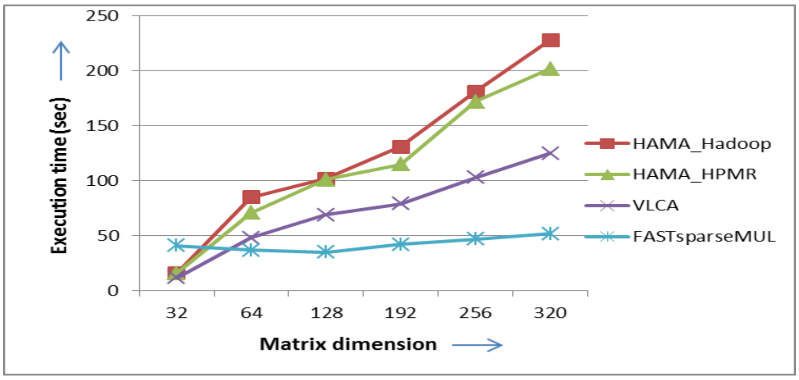


Fig. 5. Execution time comparison of FASTsparseMUL with sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA

FASTsparseMUL is executed on single node Hadoop-pseudo distributed cluster environment with 1% sparse matrices having dimensions varying from 32 up to 320. Similarly sparse matrices multiplications with HAMA_Hadoop, HAMA_HPMR and VLCA are implemented in the same environment. On average, FASTsparseMUL shows approximately 2.8 times, 2.6 times and 1.7 times reduction in time complexity compared to sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA respectively.

The execution times of different sparse matrices multiplication approaches are tabulated in Table 2 and compared in Fig. 5. Though FASTsparseMUL's initial execution time for matrix dimension 32 is more, it takes less execution time for the next remaining matrix dimensions. The sample input file and overviews of the FASTsparseMUL's mapreduce job execution are as shown below from Fig. 6 to Fig. 11b.

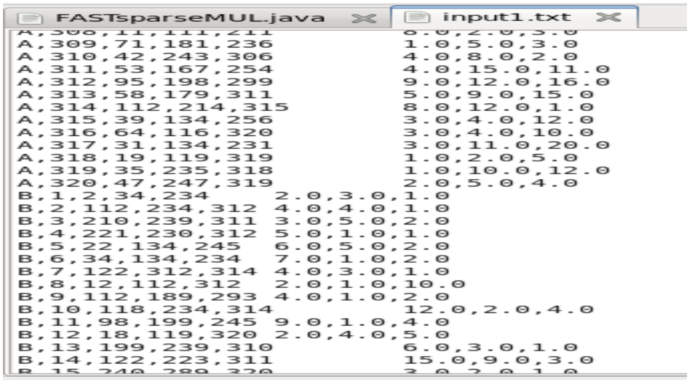


Fig. 6. Snapshot of part of the sample input file for the matrix dimension 320

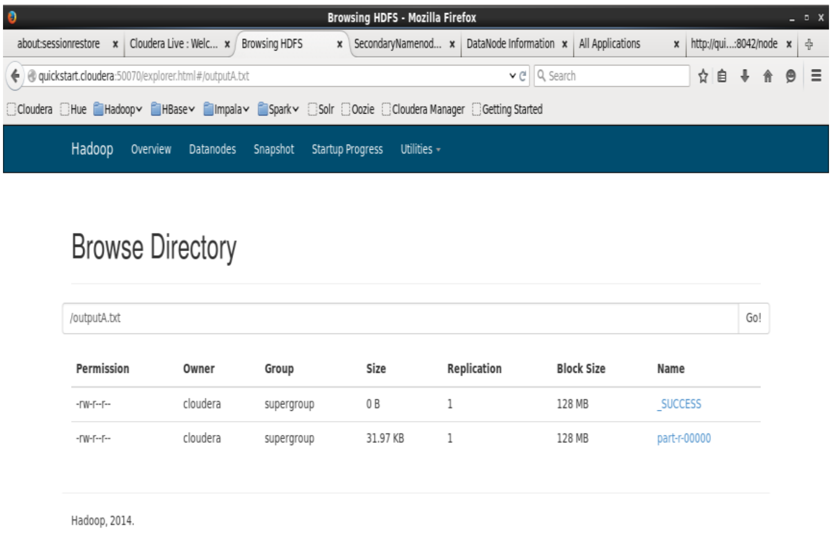
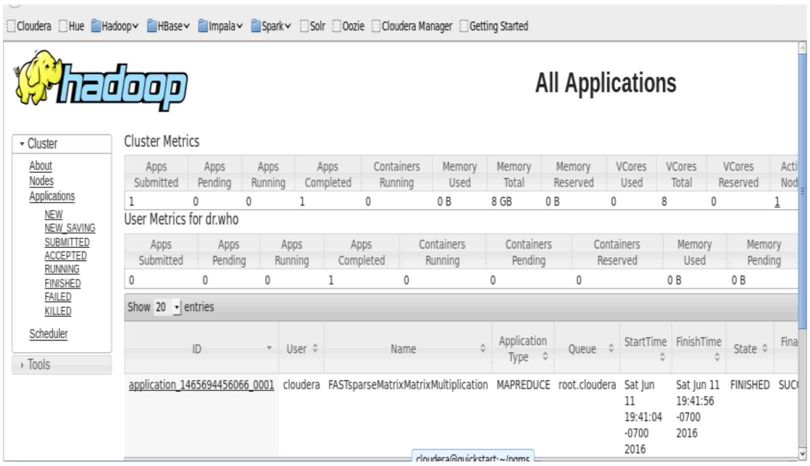
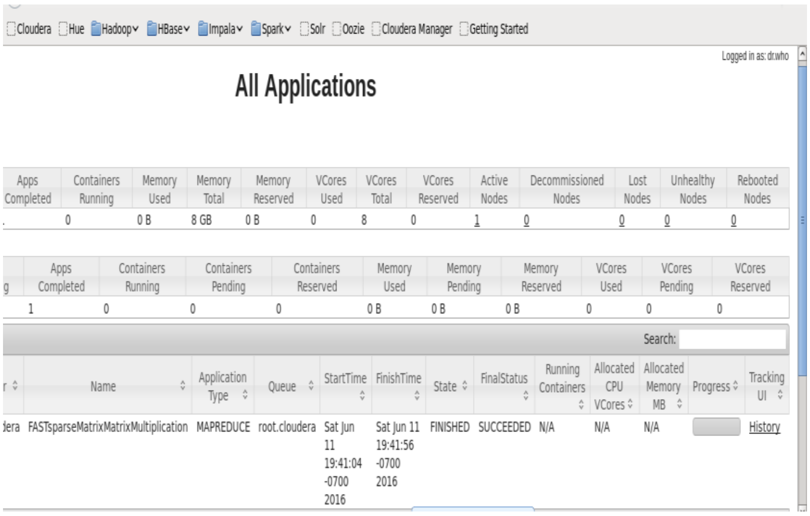


Fig. 7. Snapshot of the output file contents for the matrix dimension 320



(a)



(b)

Fig. 8. Overview of the mapreduce application of FASTsparseMUL for matrix dimension 320, displaying execution time of FASTsparseMUL for the matrix dimension 320. (Finish Time – Start Time = 19:41:56 – 19:41:04 = 52 sec (shown in Table 2))

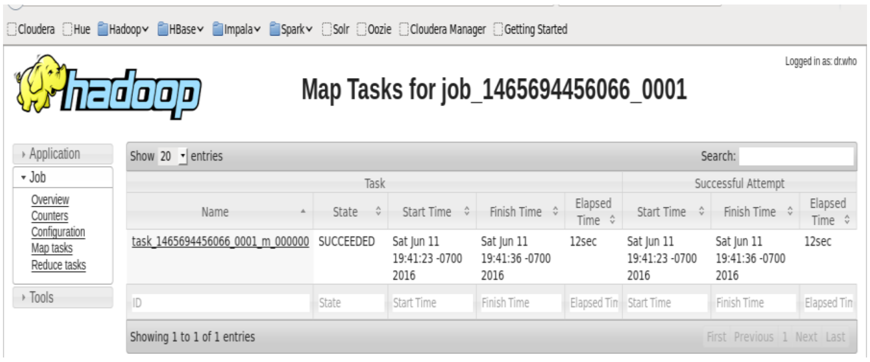
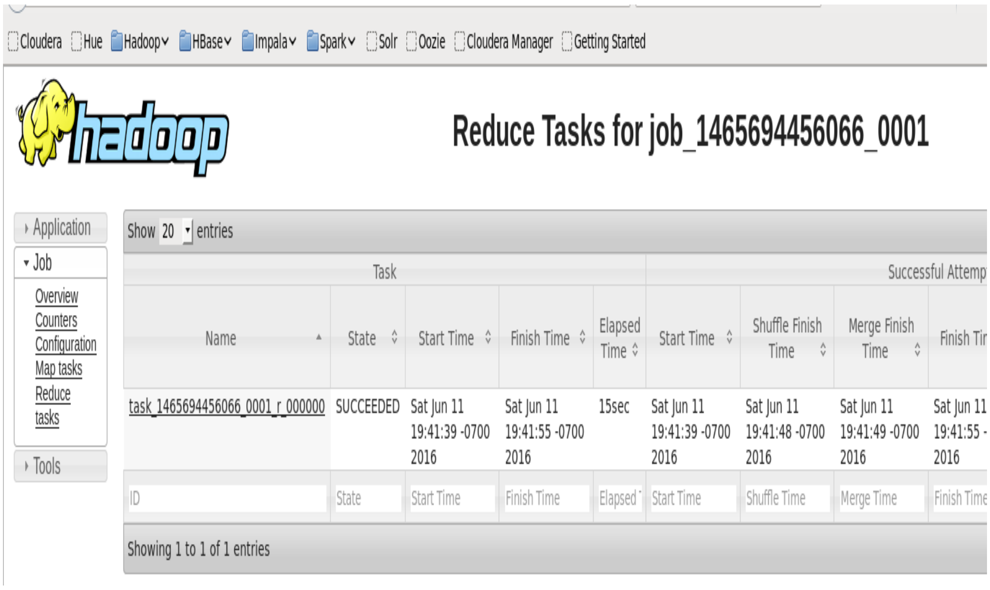
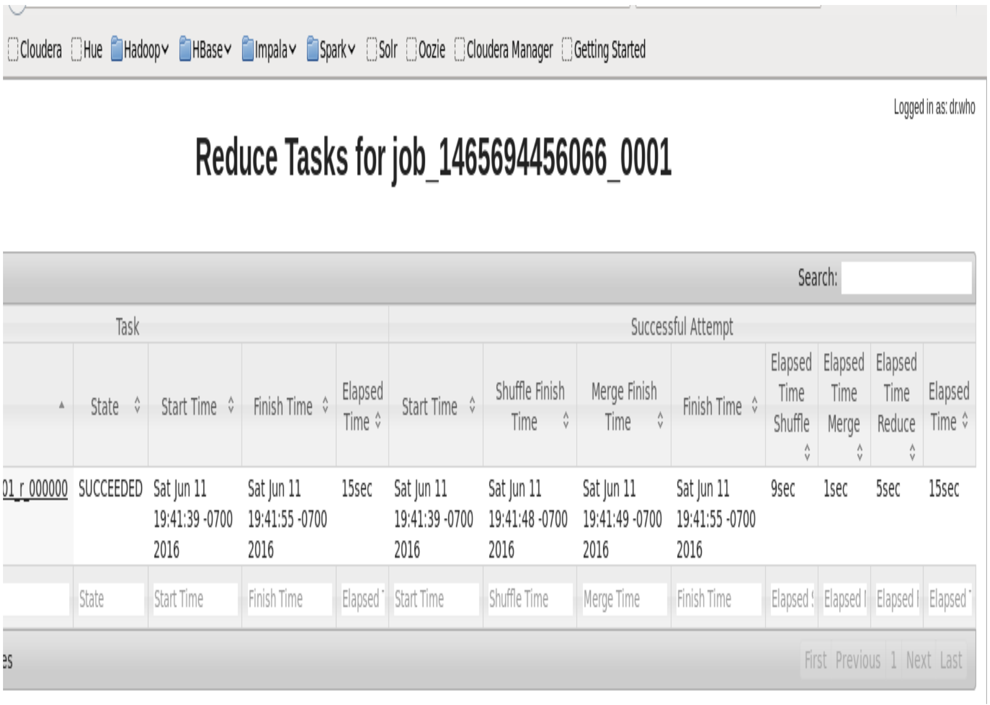


Fig. 9. Overview of map tasks for the FASTsparseMUL's mapreduce job



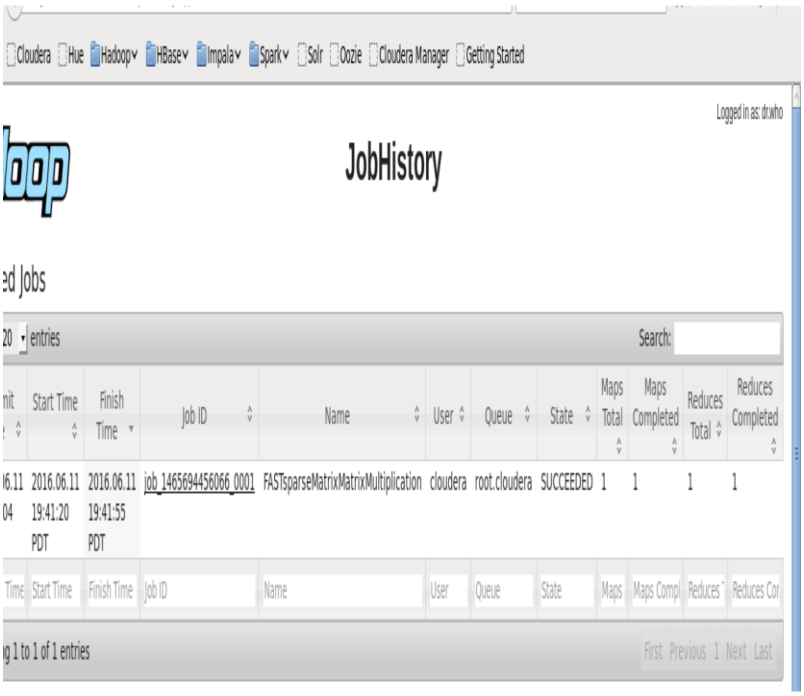
(a)



(b)

Fig. 10. Overview of reduce tasks for the FASTsparseMUL's mapreduce job

(a)



(b)

Fig. 11. Overview of the history of FASTsparseMUL's mapreduce job

Scale up is calculated by using the following formula,

(1) $$\text{Scale up(dimension)} = \log(T(\text{dimension})/T(32)),$$

where T denotes the execution time.

Scale up is inversely proportional to the scalability. The scalability improvement of FASTsparseMUL compared to sparse matrices multiplication using HAMA_Hadoop, HAMA_HPMR and VLCA approach is depicted in Fig. 12 and tabulated in Table 3.

Table 3. Scale up values of various sparse matrices multiplication approaches

Matrix dimension	Scale up			
	HAMA_Hadoop	HAMA_HPMR	VLCA	FASTsparse MUL
32	0.0	0.0	0.0	0.0
64	0.73	0.65	0.6	–0.05
128	0.80	0.80	0.76	–0.07
192	0.91	0.86	0.82	0.01
256	1.05	1.03	0.93	0.06
320	1.15	1.10	1.02	0.1

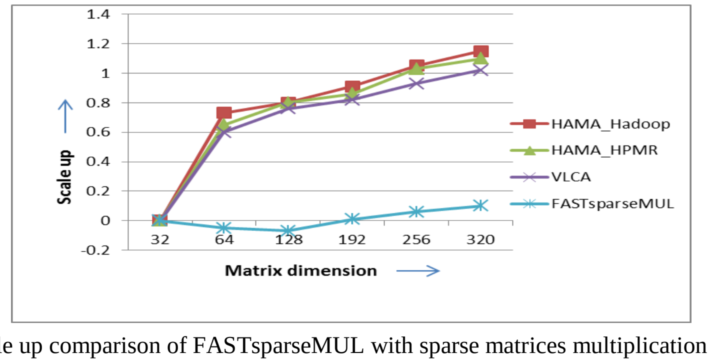


Fig. 12. Scale up comparison of FASTsparseMUL with sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA

5. Discussion

The proposed FASTsparseMUL algorithm is compared with sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA on a single node hadoop pseudo distributed environment. Though the algorithm initially takes more time for execution, it takes less time for other matrix dimensions afterwards, as shown in Fig. 5 and Table 2. There is improvement in the scalability also. Scale up values are low for FASTsparseMUL as shown in Fig. 12 and Table 3, which means that the scalability is high comparing to the sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA, as scale up is inversely proportional to scalability. Possible decrement in execution time and increment in scalability prove that the algorithm is more suitable for Big Data applications. The algorithm may be combined with HAMA_Hadoop or HAMA_HPMR or VLCA in the fully distributed cluster environment to get the results still better in the big sparse data perspective.

6. Conclusion

An efficient Big Data algorithm for the multiplication of a pair of sparse matrices is proposed. In the sparse data case, the experiments prove that the algorithm outperforms the state-of-the-art big matrices multiplication approaches. It is more suitable for the Big Data applications showing better results in terms of scalability and execution time compared to the sparse matrices multiplication approaches of HAMA_Hadoop, HAMA_HPMR and VLCA. Application of the algorithm to dense matrices is yet to be studied. There are some future research directions possible in this problem domain. FASTsparseMUL may be combined and implemented with HAMA-Hadoop or HAMA-HPMR or VLCA to get significant improvement in the performance of sparse matrices multiplication. Moreover, FASTsparseMUL may be further developed to perform sparse matrices chain multiplication. The implementations of FASTsparseMUL with Spark and HBase are other possible research directions. The Big Data algorithms with compact representations of matrices are more desirable to improve the performance of sparse matrices data processing. The Big Data research needs encouragement in this problem domain.

References

1. Di Felice, P., A. Agnifili, E. Clementini. Data Structures for Compact Sparse Matrices Representation. – J. Adv. Eng. Software, Vol. **11**, 1989, No 2, pp. 75-83.
2. Abadi, D. J. Column-Stores for Wide and Sparse Data. – In: Proc. of 3rd Biennial Conference on Innovative Data Systems Research (CIDR), January 2007, pp. 1-6.
3. Neelima, B., S. R. Prakash. Effective Sparse Matrix Representation for the GPU Architectures. – Int. J. of Computer Science, Engineering and Applications (IJCSEA), Vol. **2**, 2012, No 2, pp. 151-165.
4. Yuster, R., U. Zwick. Fast Sparse Matrix Multiplication. – J. ACM Transactions on Algorithms (TALG), Vol. **1**, 2005, No 1, pp. 2-13.
5. Dean, J., S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. – Communications of the ACM, Vol. **51**, 2008, No 1, pp. 107-113.
6. White, T. H. The Definitive Guide. O'Reilly Media, USA, 2009.
7. Buluc, A., J. R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. – Siam J. Sci. Comput., Vol. **34**, 2011, No 4, pp. C170-C191.
8. Ballard, G., A. Bulluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, S. Toledo. Communication Optimal Parallel Multiplication of Sparse Random Matrices. – In: Proc. of 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures, New York, July 2013, pp. 222-231.
9. Smith, S., N. Ravindran, N. D. Sidiropoulos, G. Karypis. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. – In: Proc. of 29th IEEE International Parallel & Distributed Processing Symposium, May 2015.
10. Sun, Z., T. Li, N. Rishe. Large-Scale Matrix Factorization using MapReduce. – In: Proc. of International Conference on Data Mining Workshops, December 2010, pp. 1242-1248.
11. Seo, S., E. J. Yoon, J. Kim, S. Jin, J. S. Kim, S. Maeng. HAMA: An Efficient Matrix Computation with the MapReduce Framework. – In: Proc. of IEEE 2nd International Conference on Cloud Computing Technology and Science (CloudCom), November 2010, pp. 721-726.

12. S e o, S., I. J a n g, K. W o o, I. K i m, J. S. K i m, S. M a e n g. HPMR: Prefetching and Pre-Shuffling in Shared MapReduce Computation Environment. – In: Proc. of 11th IEEE International Conference on Cluster Computing, New Orleans, August 2009, pp. 1-8.
13. M y u n g, J., S. L e e. Matrix Chain Multiplication via Multi Way Join Algorithms in Mapreduce. – In: Proc. of 6th International Conference on Ubiquitous Information Management and Communication, Article No 53, February 2012.
14. Z h e n g, J. H., L. J. Z h a n g, R. Z h u, K. N i n g, D. L i u . Parallel Matrix Multiplication Algorithm Based on Vector Linear Combination Using MapReduce. – In: Proc. of IEEE 11th World Congress on Services, July 2013, pp.193-200.
15. C e c c a r e l l o, M., F. S i l v e s t r i. Experimental Evaluation of Multi-Round Matrix Multiplication on MapReduce. – In: Proc. of ALENEX'15 Meeting on Algorithm Engineering & Experiments, Society for Industrial and Applied Mathematics Philadelphia, January 2015, pp. 119-132.
16. C h o i, J., J. J. D o n g a r r a, R. P o z o, D. W. W a l k e r. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. – In Proc. of 4th Sysmposium on the Frontiers of Massively Parallel Computation, IEEE, October 1992, pp. 120-127.
17. B o s i l c a, G., A. B o u t e i l l e r, A. D a n a l i s, T. H e r a u l t, P. L e m a r i n i e r, J. D o n g a r r a. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. – J. Parallel Computing, Vol. **38**, 2010, No 1-2, pp. 37-51.
18. Q i a n , Z., X. C h e n, N. K a n g, M. C h e n, Y. Y u, T. M o s c i b r o d a, Z. Z h a n g. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. – In Proc. of 7th ACM European Conference on Computer Systems EuroSys'2012, Berne, Switzerland, April 2012, pp. 197-210.