Importance of Compression Technique in Matrix and Graph Computations on Mapreduce Framework

Mandavamanjula[1], Ms.J.Ramadevi[2]
1 PG Student, CSE Department, PVPSIT College/JNTUK University, Vijayawada, A.P, India,
2 Sr. Asst. Professor, CSE Department, PVPSIT College/JNTUK University, Vijayawada, A.P, India
Correspondence Author: Mandavamanjula

ABSTRACT
As throughout this paper, we've got associate degree inclination to propose coming up with and usage of the N/W Overlapped Compression theme and CAS theme. CAS decreases the info load time and also hides the compression overhead and processed one after another in sequential fashion of a network Input/output demand transfer with bind format. CAS improves the dynamic allocation of blocks with different block sizes.[4] Information on scientific applications and analytic applications running in Hadoop can increase info compression has become essential to store this info at intervals an affordable storage value, although info is usually keep compressed, presently Hadoop takes forty ninth longer compressed info is comparison with uncompressed information. Compressed info decreases the task correspondence and Distributing the uneven work for every block. The MR parallel programming pattern ought to alleviate the magnitude connection. To boot, are bent to implement a MR phase that recognizes the features of bind info spice up method allotment and load balance ,in combination of intelligence officers, CAS and MR phases job execution time reduces the time by sixty six and data load time by thirty first.
KEYWORDS — information bind, MapReduce, Hadoop

Date of Submission: 29-06-2018 Date of acceptance: 14-07-2018

I. INTRODUCTION

In present days we have a dramatic increase of data from different sources such as like independent and from the connected services. The best example is Internet why because of data in the internet is updated and used by worldwide.[1] Internet needs high storage spaces and we are also not estimate how much data is added per second is also tough. Large data collection with different data types is nothing but a BIGDATA even in the bigdata applications also large data processing is not supported by the NOSQL databases and PDMS (parallel database management systems).We are using the HBase in HADOOP., [4]and Hadoop MapReduce(MR) used for knowledge extraction. Hadoop process the bind knowledge is 49% slower than the uncompressed data. However, knowledge binding increases the processing time and reduces the storage spaces on disk space. The most scientific applications of data sets are binded/compressed format.

Data warehouses are with the large scale data of Facebook is compress all data. Usually Facebook Collect more than 600TB of data with uncompressed format as of 2013,[2] this leads to adding a storage racks for every two days. As we are comparing with a new compression factor of 7, the storage data is diminishes to 100TB and huge quantity of information is processed to data ware houses it occupies 9.6GB it would be more feasible and reasonable.

Many companies use different types of compression formats like[2] LZO, snappy, and Gzip to reduce their storage spaces and increases their runtime. In Hadoop MapReduce pattern is used to reducece the processing time of binded knowledge. MapReduce performance is done by 2 factors

1. Hadoop distributed file system doesn't treated the storage space for the binded information it is treated as unbinded information.
2. MapReduce's resource allocation is binded size but the required resources after the the execution is uncompressed size.

For the issues we have 3 problems

1. For every mapping tasks are created and the compressed data is determined. For this the amount of mapped tasks and compression ratios are decreases as long as the process of every tasks is increases this leads to increases the job runtime.
2. Every map tasks runtime and the length of the map tasks is vary and it also depends upon the compression ratios. Variations among the tasks runtime then prolongs the job execution time it leads to back up the load balance.
3. Compressing data is significant in data warehouses there is a time taken to load knowledge to clusters on data centres.

II. LITERATURE SURVEY

Data/Knowledge compression is widely used in the computing applications with the utilization of parallel database management systems. Knowledge compression have many advantages for back of storage consumption. Optimized row column format is enhanced usage of [2]Facebook to improve the compressed statistics it is mostly dependent information and processed with queries. Implementation of Hadoop produces massive scale processing on scientific and analytic applications.

Large scale compressed block by block matrix-matrix multiplications are used in data analytic applications and scientific applications. The matrices can be categorized into two broad types such as sparse and dense matrices. In order to process matrix computations in parallel, partitioning the input matrix in blocks and these partitions are assigned to different processes. For the solution of a linear system $Ax = b$, efficient block by block matrix-matrix and matrix-vector operations are carried out. The efficiency of a given large scale compressed block by block matrix-matrix multiplication algorithm depends upon amount of arithmetic operations and storage required by a procedure, stride, memory access, and other data movement overheads.

An Overview of Compression techniques

The naturally Hadoop supported compression format is Gzip. [5]Gzip is based on DEFLATE algorithm, which combination of Huffman coding and LZ77 algorithms. LZO compression is composed many smaller blocks (256K) of compressed data and split along block boundaries. LZO is designed with speed in mind: it decompresses twice as fast as Gzip, but it does not compress quite as fast as Gzip. Snappy is a compression and decompression algorithm. The main aim is very high speed and reasonable compression. An other efficient mode zlib, Snappy is very fast for more inputs, but the resulted compressed files are from 20% to 100% larger. Snappy compress at 250 MB/sec and decompress at 500 MB/sec on the single core with the i7 process with 64-bit mode. Snappy is widely used in Google, Big Table and MapReduce to our internal RPC system.

Graph Computations

Graph computations are used in numerous scientific applications. The large graph computations involve building of sparse matrix based on adjacency graph which contain edge weights as per applications. The existing algorithms have restrictions to load the complete graph data and process in the memory of the system. Typical graph computational algorithms assume that the graph fits in the memory of a system, and in case of BIG DATA, the data of graph is spanned across the multiple Giga to Tera bytes of data. The graph partitioning methods are employed to distribute the sub-graphs on multiple core processors which show orders of magnitude performance greater than traditional sequential algorithms. Also, Out-of-Core algorithms for graph computations have been established.

The graph data can be stored on the disk the format of storage as like compressed spare row and compressed column .The graph values can be processed in sequential fashion.

Graph Computations – on a sparse Graph

Several authors developed algorithms for parallelization of storage of a sparse graph $G(V,E)$ having v vertices and e edges. These algorithms involve unstructured communications and load balancing is the significant issue in implementation. The first step is to use decomposition techniques which recognize the concurrency that is available and decompose it into tasks that can be executed in parallel. The next step in the process of designing a parallel algorithm is to take these tasks and assign (i.e. map) them to the available processors. Achieving load balancing is another main issue in the coloring of a graph. Static load balancing methods distributes the work among processors prior to execution of the algorithm. Dynamic load balancing methods distribute the work among processors during the execution of the algorithm. Algorithms such as coloring a sparse graph, parallel graph partitioning algorithms, and parallel search algorithms require dynamic load balancing.

In graph coloring no two adjacent vertices have the same color is called the proper coloring of a graph. A set of vertices in a graph is said to be an independent set of vertices or simply an independent set if no two vertices in the set are adjacent.[6] A maximal independent set is an independent set to which no other vertex can be added without destroying its independence property.

Explain the coloring of sparse graph using three colors. Several algorithms exist in parallel and serial graph coloring. Luby"s Algorithm for coloring a graph is quite popular. Luby's algorithm chooses each maximal independent set by using a randomized algorithm in an incremental fashion. Graph partitioning with Lucy's algorithm for a graph coloring has been used in many applications. Large-scale graph partitioning tools are available, e.g. METIS, SCOTCH, and Chaco are used in graph colourings algorithms.

III. EXISTING SYSTEM

The no.of mapping tasks decreases the compression ratio also decreases and the[3] quantity of the data for each process increase. Individual process resulting more data increased job runtime and decreased parallelism. The variation between each map task's runtime increases. Runtime variation results long running tasks which prolong job execution time and reduce load balance time. Compression and data transfer is done sequentially which further increases the time till users can process their data.

IV. PROPOSED SYSTEM

The proposed algorithm and procedures identifying and solving the above problems to avoid the n/w traffic overhead and I/O map task skews in the proposed work.

Develop Network Overlapped Compression (NOC), reduces[3] data load time and hides compression overhead by interleaving network I/O. Compression Aware Storage (CAS), increases parallelism by changing file's dynamically block size with compression ratio. Map Reduce module, which recognizes the features of compressed data to improve resource allocation and load balance.

V. IMPLEMENTATION

MapReduce:

- MapReduce is a programming pattern/model.
- MapReduce is implemented for large data sets for processing and generating large data with parallel distributed algorithms that data is stored in HDFS.
- It contains Compressed Task Divider and Compressed Task Reader.

New Compression Algorithm (BZIP2):

The compression algorithm to processing the compressed data using Map Reduce framework.

- The problem is aimed to improve performance of compressed data processing using Map Reduce (MR) framework.
- The new compression algorithm is split table the input data, to reduce the secondary storage space and increase the I/O transfer rate.
- Develop the Map Reduce module to recognize the features of compressed data to improve the allocation of resources and load balance.[8]

Bzip2 Compressed Matrix Computation using MR

Matrix multiplication of 2 dimensional dense matrices or sparse matrices based on Hadoop MapReduce can be developed in different ways. For simplicity we assume that both of the input matrices either square matrices or rectangular matrices. Several different strategies are possible for partitioning the input matrix A and input matrix B and then compresses the block matrices using Bzip2 compression in Hadoop MapReduce implementation. It is expected that the Hadoop cluster is size "p" and each node "m" in the cluster has 16 core processor joined with fast Ethernet network.

STRATEGY 1:

In MapReduce job framework, each map task processes[9] a compressed block of the input file. In the case of compressed block matrix multiplication, the matrix A is one input file and the block contain a piece of block data from the file i.e. matrix A. The two matrices are necessary as input matrices for matrix computation. Each map task can access one input matrix only but not both matrices at a time if both matrices are stores separate HDFS files.

Different implementations such as (i) map task reads the decompressed data from the both matrices, (ii) distributes the input matrix data to reduce tasks, (iii) reduce task completes the compressed block matrix multiplication and (iv) reduce task use temporary HDFS file to write the intermediate result.

STRATEGY 2:

We divide the each of the input matrix A and input matrix B into X * X small square blocks of same size and compresses the input matrix block using Bzip2 compression. The size of the each compressed input block X * Q and Q * X and output matrix consists of X* X blocks of data and each resulting from addition of B compressed block matrix multiplications. In our implementation, each task handles one compressed block matrix multiplication. We map each task to each mapper and X3 map tasks are necessary to complete the computation.

STRATEGY 3:
We divide the each of the input matrix A into x row wise strips and compress the block [9]and input matrix B into x column wise strips and compress the block using Bzip2 compression. The mapper tasks are given to one row wise block strip and one column wise block strip. Each map task yields m/x * m/x blocks in the output matrix. The total number of map tasks in the partition would be x2.

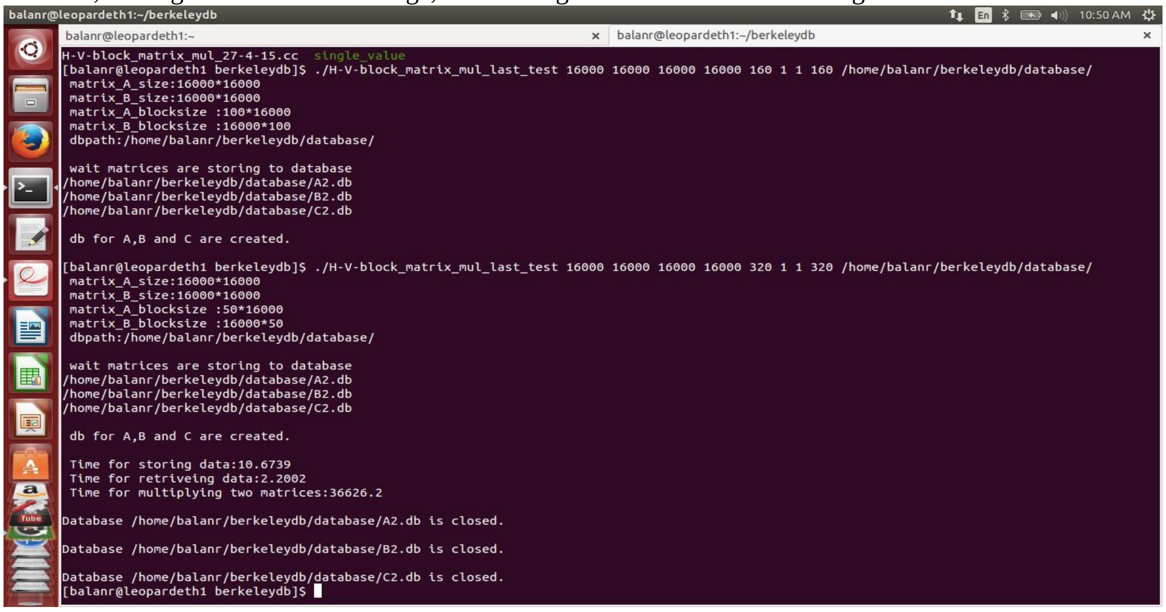
STRATEGY 4:
There are two techniques to implement bzip2 compressed block matrix multiplication. (i) The block partitioning of matrix A and row wise partitioning of the input matrix B and perform the matrix computations and (ii) row wise partitioning of the input matrix A and the block partitioning of the input matrix B. These two strategies require m2 map tasks. We have implemented the first strategy for bzip2 compressed block matrix multiplication in this this.
This implementation performs bzip2 compressed block by block matrix multiplication using HMR
STEP 1: Push the matrix A and matrix B into HDFS
STEP 2: Set the compressed input and compressed output format classes, Mapper and Reducer classes, file paths and submit the job.
STEP 3: The mapper1 generates key, values pair's compresses the mapper1 output and pass to reducer1 so that each reducer gets one block of matrix.
STEP 4: The reducer1 performs block matrix multiplication.
STEP 5: The mapper1 generates key, values pair's compresses the mapper1 output and pass to reducer1 so that each reducer gets respected partial sum.
STEP 6: The reducer2 performs summation of all partial sums then compress the output and writes into HDFS.

Implementation of Graph Computations based on Bzip2 Compression
The graph nodes are numbered from zero to N − one per the penning order. [6]We tend to let A(x) denotes successors set of node x that is the set . y such there's Associate in nursing arc from x to y and a pair. We tend to represent the graph mistreatment with different words in the contiguity list and coded because the sequence of contiguity lists of nodes zeros, one, etc., every Precedence of the out degree of the corresponding node, to form it self-delimiting. It also suggests that we always represent successors list with the inventory gaps. If A(x) = (a1, . . . ,ak), we'll represent it as (a1 − x,a2 − a1 − one,a3 − a2 − one, . . . ,ak − ak−1 − 1). To improve the compression quantitative relation is to take advantage of similarity rather than representing the contiguity list A(x) directly we are able to code it as a modified.
Differential compression:
Differential compressions are using in the web graph with variations of unit area A(y) is recorded in sequence of copy blocks. As, associate copy list is processing with in a sequence of 1 and 0 blocks it also specify the blocks length for every block. We always leave the last block from the blocks list and the block count is from out [6]degree with a node reference. The block 1 is always refers a primary copy block.
The list of additional nodes is compressed as follows:

- An inventory of whole number intervals every interval is diagrammatic by its extremes left e area unit compressed mistreatment with variations between every left extreme and therefore the previous right extreme minus a pair of interval lengths area unit decremented by the brink minimum.
- An inventory of residuals compressed variations.
- Total input non-zero elements =N=N
- Sub matrix splits (in each dimension) =s=s
- Map-1 output records = Reduce-1 input records =Ns=Ns
- Reduce-1 number of keys =s3=s3
- Reduce-1 records per key =Ns2=Ns2
- Reduce-1 output data volume = Map-2 input data volume ∝Ns∝Ns
- Reduce-2 output data volume ∝N∝N

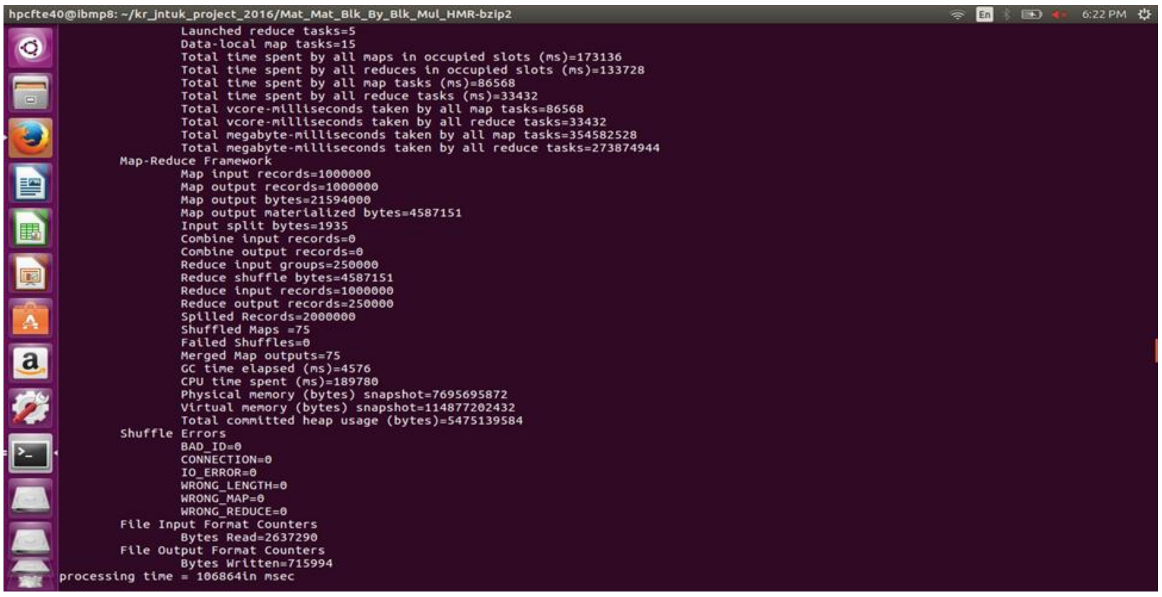
VI. RESULTS AND DISCUSSIONS

Here, the input is sparse matrix with different size such as like n*n or 2000*2000 etc. the input values are tested on MR cluster and by using with Bzip2 compression internally Hadoop can map and reduce the input value and produces the processing time for every one and it also The components of software such as matrix data generation, writing the matrices to storage, and reading matrix data as shown in figure 1.



```
balanr@leopardeth1:~/berkeleydb                                    10:50 AM
balanr@leopardeth1:~                        balanr@leopardeth1:~/berkeleydb
H-V-block_matrix_mul_27-4-15.cc  single_value
[balanr@leopardeth1 berkeleydb]$ ./H-V-block_matrix_mul_last_test 16000 16000 16000 16000 160 1 1 160 /home/balanr/berkeleydb/database/
 matrix_A_size:16000*16000
 matrix_B_size:16000*16000
 matrix_A_blocksize :100*16000
 matrix_B_blocksize :16000*100
 dbpath:/home/balanr/berkeleydb/database/

 wait matrices are storing to database
/home/balanr/berkeleydb/database/A2.db
/home/balanr/berkeleydb/database/B2.db
/home/balanr/berkeleydb/database/C2.db

 db for A,B and C are created.

[balanr@leopardeth1 berkeleydb]$ ./H-V-block_matrix_mul_last_test 16000 16000 16000 16000 320 1 1 320 /home/balanr/berkeleydb/database/
 matrix_A_size:16000*16000
 matrix_B_size:16000*16000
 matrix_A_blocksize :50*16000
 matrix_B_blocksize :16000*50
 dbpath:/home/balanr/berkeleydb/database/

 wait matrices are storing to database
/home/balanr/berkeleydb/database/A2.db
/home/balanr/berkeleydb/database/B2.db
/home/balanr/berkeleydb/database/C2.db

 db for A,B and C are created.

 Time for storing data:10.6739
 Time for retriveing data:2.2002
 Time for multiplying two matrices:36626.2

Database /home/balanr/berkeleydb/database/A2.db is closed.

Database /home/balanr/berkeleydb/database/B2.db is closed.

Database /home/balanr/berkeleydb/database/C2.db is closed.
[balanr@leopardeth1 berkeleydb]$
```

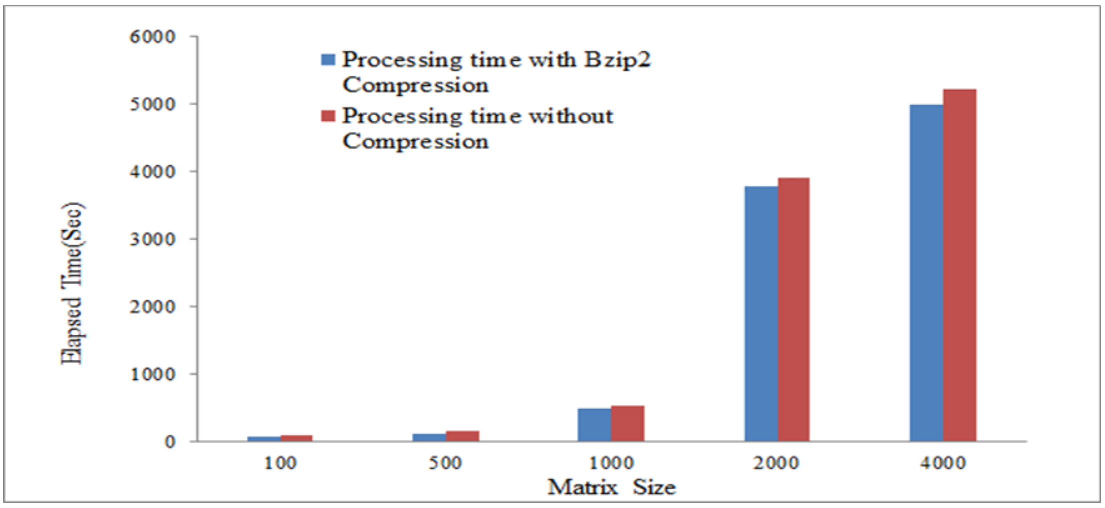
“Figure 1: Elapsed time for compressed block by block matrix multiplication of Size 8000 using Bzip2 on a single nodecluster



```
hpcfte40@ibmp8: ~/kr_jntuk_project_2016/Mat_Mat_Blk_By_Blk_Mul_HMR-bzip2        6:22 PM
                Launched reduce tasks=5
                Data-local map tasks=15
                Total time spent by all maps in occupied slots (ms)=173136
                Total time spent by all reduces in occupied slots (ms)=133728
                Total time spent by all map tasks (ms)=86568
                Total time spent by all reduce tasks (ms)=33432
                Total vcore-milliseconds taken by all map tasks=86568
                Total vcore-milliseconds taken by all reduce tasks=33432
                Total megabyte-milliseconds taken by all map tasks=354582528
                Total megabyte-milliseconds taken by all reduce tasks=273874944
        Map-Reduce Framework
                Map input records=1000000
                Map output records=1000000
                Map output bytes=21594000
                Map output materialized bytes=4587151
                Input split bytes=1935
                Combine input records=0
                Combine output records=0
                Reduce input groups=250000
                Reduce shuffle bytes=4587151
                Reduce input records=1000000
                Reduce output records=250000
                Spilled Records=2000000
                Shuffled Maps =75
                Failed Shuffles=0
                Merged Map outputs=75
                GC time elapsed (ms)=4576
                CPU time spent (ms)=189780
                Physical memory (bytes) snapshot=7695695872
                Virtual memory (bytes) snapshot=114877202432
                Total committed heap usage (bytes)=5475139584
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2637290
        File Output Format Counters
                Bytes Written=715994
processing time = 106864in msec
```

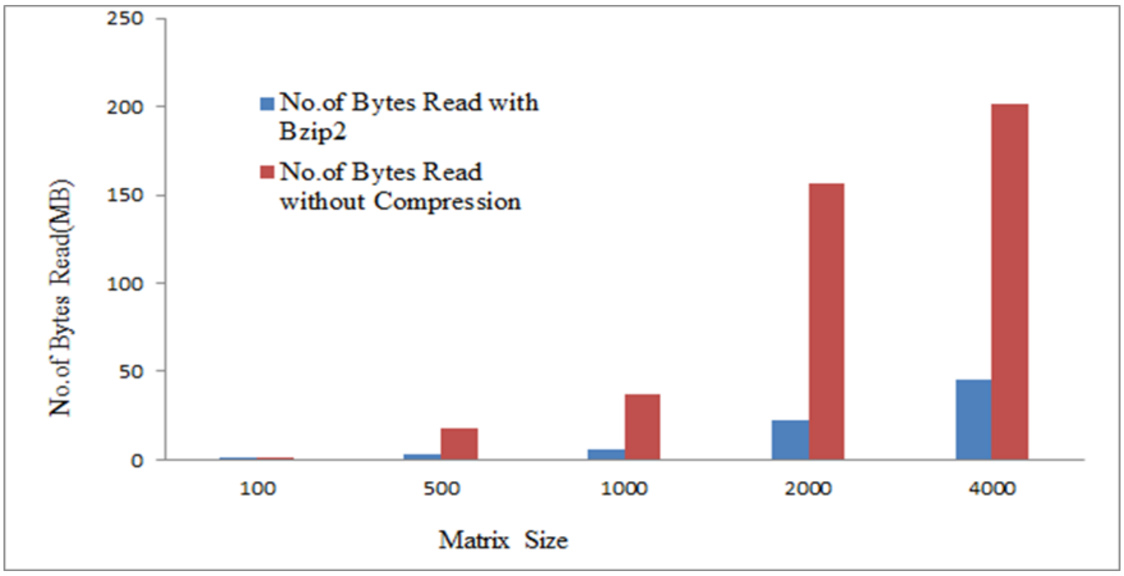
“Figure 2 Elapsed time for compressed block by block matrix multiplication of a matrix size 8000 using compression on a single node of a Hadoop Cluster”

The processing time is always less for compressed block by block matrix multiplication then without compressed block by block matrix multiplication. Block by block matrix multiplication based on Bzip2 and MR



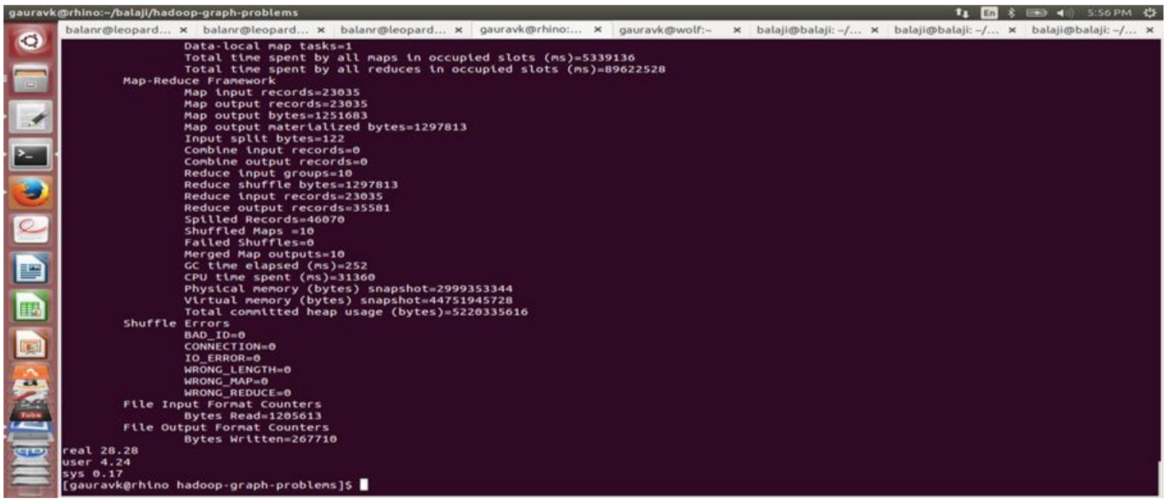
"Figure3. PROCESSING TIME COMPARISIONS ON HMR and BZIP2"

The elapsed time depicted as histogram on performance of the block by block matrix multiplication using Bzip2 compression and MapReduce of various matrix sizes. From the figure-3 we observed that for the matrix size 4000, the block by block matrix multiplication using Bzip2 compression takes 11% less elapsed time compare to block by block matrix multiplication using Hadoop MapReduce as shown above figure 3.
The figure-4 Illustrates storage of number of bytes read for block by block matrix multiplication using Bzip2 and MapReduce. In general MapReduce takes 201MB space to read the data for matrix of size 4000, whereas by using Bzip2 to store the same size matrix data that takes 45.5Mb of space. Therefore we are saving nearly 40% of storage by using Bzip2 compression compared general MapReduce process.



"Figure 4. No.of Bytes read withBZIP2and HMR CLUSTER"

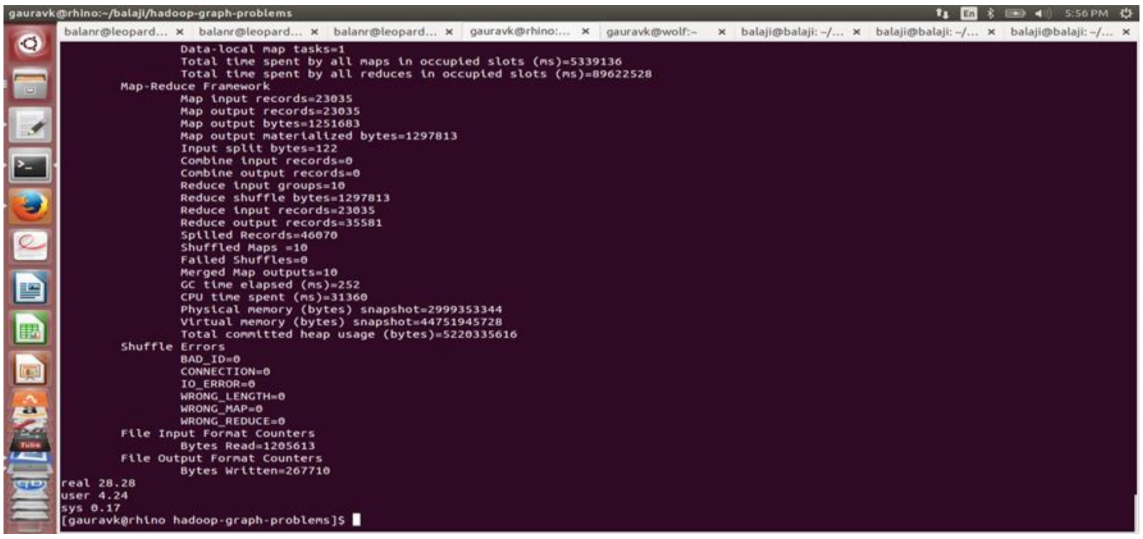
The results for sparse matrix vector multiplication computations for different problem sizes are shown in the table. The components of software such as matrix data generation, sparse graph data are shown in the figure-5.



```
gauravk@rhino:~/balaji/hadoop-graph-problems
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=5339136
                Total time spent by all reduces in occupied slots (ms)=89622528
        Map-Reduce Framework
                Map input records=23035
                Map output records=23035
                Map output bytes=1251683
                Map output materialized bytes=1297813
                Input split bytes=122
                Combine input records=0
                Combine output records=0
                Reduce input groups=10
                Reduce shuffle bytes=1297813
                Reduce input records=23035
                Reduce output records=35581
                Spilled Records=46070
                Shuffled Maps =10
                Failed Shuffles=0
                Merged Map outputs=10
                GC time elapsed (ms)=252
                CPU time spent (ms)=31360
                Physical memory (bytes) snapshot=2999353344
                Virtual memory (bytes) snapshot=44751945728
                Total committed heap usage (bytes)=5220335616
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1205613
        File Output Format Counters
                Bytes Written=267710
real 28.28
user 4.24
sys 0.17
[gauravk@rhino hadoop-graph-problems]$ 
```

"Figure 5. Elapsed time for sparse matrix vector multiplication a sparse graph with 23035 nodes (Graph-1) nodes in a two nodes of a Hadoop cluster with Bzip2 compression".

The sparse graph and the size of the sparse matrix, sparsity of the sparse graph and elapsed time for each one are indicated. We observed that for sparse matrix sizes of 2048 and 10240, the Sparse Matrix Vector multiplication in two node cluster takes 55.72% as shown in figure-6.



```
gauravk@rhino:~/balaji/hadoop-graph-problems
                Data-local map tasks=1
                Total time spent by all maps in occupied slots (ms)=5339136
                Total time spent by all reduces in occupied slots (ms)=89622528
        Map-Reduce Framework
                Map input records=23035
                Map output records=23035
                Map output bytes=1251683
                Map output materialized bytes=1297813
                Input split bytes=122
                Combine input records=0
                Combine output records=0
                Reduce input groups=10
                Reduce shuffle bytes=1297813
                Reduce input records=23035
                Reduce output records=35581
                Spilled Records=46070
                Shuffled Maps =10
                Failed Shuffles=0
                Merged Map outputs=10
                GC time elapsed (ms)=252
                CPU time spent (ms)=31360
                Physical memory (bytes) snapshot=2999353344
                Virtual memory (bytes) snapshot=44751945728
                Total committed heap usage (bytes)=5220335616
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=1205613
        File Output Format Counters
                Bytes Written=267710
real 28.28
user 4.24
sys 0.17
[gauravk@rhino hadoop-graph-problems]$ 
```

"Figure 6. Elapsed time for sparse matrix vector multiplication a sparse graph with 23035 nodes (Graph-1) nodes in a two nodes of a Hadoop cluster without compression"

VII. CONCLUSION

The compressed block by block matrix into matrix multiplication algorithms and sparse graph computations based MPI Cluster and Hadoop Cluster framework. We described implementation approach using Compression techniques software which handles Input/output (I/O) communication is fast in internal memory and slower in the external memory for typical matrix computation algorithms. Block matrix partitioning techniques are employed in our computations and the concurrent access to BDB is discussed. The results shown in the thesis indicates that the compression algorithms on a single multi-core system gives 25% to 35% improvement in reduction of elapsed time and 40% to 47% improvement in reduction of storage for large scale matrix and sparse graph computations in a BIG DATA frame work.

REFERENCES

[1]. J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in Proceedings of USENIX Symposium on Operating Systems Design and Implementation, 2004

[2]. "Cisco*globalcloudindex: Forecast and methodology, 2013-2018*." http://cisco.com/c/en/us/solutions/collateral/serviceprovider/global-cloud- indexgci/Cloud Index White Paper.html, [Last accessed November 2014].

[3]. M. Gaggero, S.Leo, S.Manca, F. Santoni, O. Schiaratura, and Zanetti, “Parallelizing bioinformatics applications with mapreduce,” in CCA-08: Cloud Computing and its Applications, vol.19, pp.15-23, 2008.Apache Software Foundation. (2014 Mar.) Apache Hadoop Project. Available: http://hadoop.apache.org.
[4]. Adnan Haider, Xi Yang, Ning Liu, Xian-He Sun, Shuibing He. "IC-Data: Improving Compressed Data Processing in Hadoop", 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), 2015 Publication
[5]. Apache Software Foundation. (2014 Mar.) Apache Hadoop Project. Available: http://hadoop.apache.org.
[6]. The webgraph framework I", Proceedings of the 13th conference on World Wide Web-WWW04WWW04,2004Publication
[7]. M. Schatz, “Cloudburst: highly sensitive read mapping with mapreduce,” in Bioinformatics, vol. 25, pp.1363-1369, 2009.
[8]. A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt,
[9]. S. Madden, and M. Stonebraker, “A comparison of approaches to large-scale data analysis,” in SIGMOD’09, vol. 4, pp. iv-873- iv-876, 2009.

Mandavamanjula " Importance of Compression Technique in Matrix and Graph Computations on Mapreduce Framework, International Journal of Computational Engineering Research (IJCER), vol. 8, no. 7, 2018, pp. 71-78.