Compression-Aware I/O Performance Analysis for Big Data Clustering

Zhenghua Xue
Computer Network Information Center
Chinese Academy of Sciences
zhxue@cnic.cn

Jianhui Li *
Computer Network Information Center
Chinese Academy of Sciences
lijh@cnic.cn

Yang Zhang
Computer Network Information Center
Chinese Academy of Sciences
zhangyang@cnic.cn

Geng Shen
Computer Network Information Center
Chinese Academy of Sciences
shengeng@cnic.cn

Qian Xu
Baidu.com Inc., China
xuqian01.baidu.com

Jing Shao
Computer Network Information Center
Chinese Academy of Sciences
jingshao@cnic.cn

ABSTRACT
As the data volume increases, I/O bottleneck has become a great challenge for data analysis. Data compression can alleviate the bottleneck effectively. Taking K-means algorithm as an example, this paper proposes a compression-aware performance improvement model for big-data clustering. The model quantitatively analyzes the effect of a variety of factors related to compression during the entire computational process. We perform clustering experiments on 10 dimensional data with up to 1.114 TB in size on a cluster computer with hundreds of computing cores. The measurement validates that using compression contributes significantly to improving the I/O performance, and confirms our theoretical analysis empirically. Furthermore, the proposed model can effectively determine when and how to use compression to improve I/O performance for big-data analysis.

Keywords
Big data clustering, Compression contribution model, I/O bottleneck

1. INTRODUCTION
Recently, the amount of data in our world has been exploding, and thus analyzing large data sets has become a key direction of innovation. Big data is valuable for both business and scientific research. Discovering patterns behind big data is a challenging but interesting as well as valuable work. Therefore, big data analysis has become a hot topic in scientific research. Clustering plays an important role in data analysis and understanding. Clustering is the organization of a collection of patterns, usually represented as a vector of measurements, or a point in a multidimensional space, into clusters based on similarity [14]. Application examples with Terabytes data clustering or even larger data volume are: massive satellite images partition to identify the fringes of rivers, mountains, deserts, etc; weather monitoring and climate change analysis with years of multidimensional records on wind speed, temperature, humidity, pollutants, and so forth. Other examples include community detection within large groups of people in social networks according to their behavior and locations, interests, purchasing records, etc. In bioinformatics, big data analysis include tasks such as identifying potentially meaningful relationships among many billions of microbial genes.

Clustering analysis encompasses a number of different algorithms and methods for grouping objects of similar kind into respective categories. However, existing algorithms are impractical for datasets spanning Terabytes and Petabytes [11]. Clustering researchers face a general dilemma: how to guarantee the algorithm efficiency when confronted with big data while guaranteeing quality. Parallelizing existing serial clustering algorithms and strategies is by far the most popular choices [11]. Server cluster is currently the most popular system architecture to run parallel programs, in which computation is partitioned into multiple processes that are concurrently running on many servers. This can significantly improve the performance of problem solving. However, computer-system performance is seriously hampered by I/O bottleneck. Big data processing today is still very inefficient in many areas even when running on a cluster system.

Due to the fact that the K-means algorithm is well known for its efficiency in clustering data sets, we adopted K-means clustering algorithm as an example to measure the clustering algorithm performance with 1TB sized data on a 14-server cluster machine, where each server has 24 computing cores. Three different strategies are measured: 1) uniform distribution of data on the servers; 2) high performance storage (Panasas) with a 10Gb/s network interface, which connects each server with 1Gb/s network; and 3) running a K-means algorithm of Mahout [13], which is based on MapReduce. The measurement results of each iteration are shown in the
K centroids and the sum of the mean-squared error. Finally, all processors perform a reduce operation to update the and the accumulated value of all possessed samples. Thirdly, all processors asynchronously execute in parallel. Secondly, it assigns all the data to each processor uniformly and initializes K global initial centroids. Secondly, all processors asynchronously execute in parallel. Each processor assigns samples to the closest centroid in the K centroids and calculates the sum of mean-squared error and the accumulated value of all possessed samples. Thirdly, all processors perform a reduce operation to update the K centroids and the sum of the mean-squared error. Finally, algorithm compares the sum of the mean-squared error and a user given threshold, and iterates if above the threshold.

### 2. I/O optimization

Recently, more and more research works have been published on improving the I/O performance. Gibson [12] provided a historical overview and discussed the potential technology trends. Ousterhout [19] discussed some typical techniques to improve I/O performance and perform a profound study of the Log-structured File System. Anderson [5] proposed a message system to improve the I/O performance of network storage system and performed test for the sequential access of files. Del Rosario [8] proposed a two-staged access policy of I/O system, which splits the data in computing nodes in accordance with storage distribution and achieved I/O performance improvement of data access for most modes. Chiang [7] proposed an I/O optimal technique for extraction of the iso-surfaces from volumetric data, which in advance establishes a search structure for the data on disk. Then, each extraction only issues the requests to the search structure. Vivek [20] proposed a unified buffering and caching system IO-Lite, and the test results on Web server show that the performance improvement ranges over 40%-80%. Ousterhout [18] proposed an idea of migration of the main location of the online data from disk to memory and disks only act as roles of the backup or archive.

Some researchers started to apply data compression to improve I/O performance. Chen [6] developed a decision-making algorithm that helps MapReduce users identify when and where to use the compression in an attempt to improve energy efficiency. Chen’s algorithm only considered the compression ratio and the frequency of reading. Abadi [1] extended C-Store(a column-oriented DBMS) designed a compression sub-system and evaluated a set of compression algorithms proposed with compression techniques used in commercial databases and showed that they can significantly alleviate I/O bottlenecks. Lee [15] proposed three methods for parallel compression of scientific data to reduce the data-migration cost and analyzed eight scientific data sets. Welton [22] haressed idle CPU resources to compress network data, reducing the amount of data transferred over the network and increasing effective network bandwidth.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Computing Duration(s)</th>
<th>Reading Duration(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Disk</td>
<td>23</td>
<td>1049</td>
</tr>
<tr>
<td>Panasas</td>
<td>24</td>
<td>1365</td>
</tr>
<tr>
<td>HDFS</td>
<td>52</td>
<td>1670</td>
</tr>
</tbody>
</table>

![Table 1: Comparison for each iteration](image)

We make two observations from the above discussion: 1) Data locality is an important consideration, which may be used to result in higher efficiency than others; 2) all three methods mentioned above perform very poorly in I/O performance, as most of the computational time is spent on reading data, which makes the CPUs keep idle for a long time. Therefore, it is desirable to design efficient methods to relieve the dramatic I/O latency. This is exactly what this paper aims to solve.

The remainder of this paper is organized as follows. In Section 2, we survey the work on data clustering and I/O performance improvement. Section 3 establishes a mathematical model to analyze I/O performance contribution using data compression in big data clustering. Section 4 measures the proposed compressing model empirically.

## 2. RELATED WORK

### 2.1 K-means algorithm

A great deal of research on K-means clustering has been conducted in data mining research. PCA-based CURLER [21] algorithm took advantages of density-based K-means algorithm, and has the advantage of being free from constraints of the correlation of attributes. PROCLUS [3] was proposed for K-medoid clustering. On the basis of PROCLUS algorithm, ORCLUS [4] included a merge clustering mechanism. The projective cluster algorithm [2] used an objective function, allowing a trade-off between the intrinsic subspace dimension and clustering error.

However, when clustering large-scale data with more than Terabytes in size, serial K-means algorithms often fail due to the inability to scale. In contrast, parallel schemes show their advantages. For big data clustering, the research community has published several parallel K-means algorithms. Dhillon [10] proposed a parallel implementation of K-means algorithm based on message passing model and analyzed the algorithm’s scalability and speedup. Li [17] proposed a bi-secing parallel K-means algorithm, which balanced the load among multiprocessors with a prediction measure. Based on the most popular parallel programming framework MapReduce[16], Zhao [23] proposed a parallel K-means algorithm and assessed the performance assessments by speed-up, scale-up and size-up.

In this paper, we extend the parallel K-means algorithm by Dhillon [9]. Firstly, it assigns all the data to each processor uniformly and initializes K global initial centroids. Secondly, all processors asynchronously execute in parallel. Each processor assigns samples to the closest centroid in the K centroids and calculates the sum of mean-squared error and the accumulated value of all possessed samples. Thirdly, all processors perform a reduce operation to update the K centroids and the sum of the mean-squared error. Finally,
the trade-off. Table 2 lists parameters used:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a-d-dimensional dataset</td>
</tr>
<tr>
<td>d</td>
<td>Dimensionality of dataset a, 10 default</td>
</tr>
<tr>
<td>η</td>
<td>Number of samples for a, 15 billion default</td>
</tr>
<tr>
<td>μ</td>
<td>Size of uncompressed data/size of compressed data</td>
</tr>
<tr>
<td>K GB</td>
<td>Size of a raw data block, 0.1123 default</td>
</tr>
<tr>
<td>K GB</td>
<td>Size of a compressed data block</td>
</tr>
<tr>
<td>D GB</td>
<td>Size of dataset a, 1114.875 default</td>
</tr>
<tr>
<td>N</td>
<td>Number of total processes, 336 default</td>
</tr>
<tr>
<td>M</td>
<td>Number of compression processes</td>
</tr>
<tr>
<td>avai_band</td>
<td>Disk reading rate in GB/Sec, 0.08969 default</td>
</tr>
</tbody>
</table>

### 3.1 Task decomposition

In the process of K-means computation, we assign two types of processes: compression processes and computation processes. Compression process is responsible for reading the raw data from local disks, then compressing the raw data and writing compressed data back to disks. Computation process is responsible for reading the compressed data from local disks, and for decompressing and calculating it. All the processing of the data is performed in a block by block manner, as shown in Figure 1.

Under parallel conditions, we assume that each compression process takes $T_{comp}$ seconds to deal with a raw data block with the size of $K$. $T_{comp}$ consists of three parts:

1. $T_{comp\_read}$: duration of reading a data block with size $K$
2. $T_{comp\_comp}$: duration of compressing the data block from the size of $K$ to the size of $K/μ$
3. $T_{comp\_write}$: duration of writing compressed data with the size of $K/μ$ back to disks.

Therefore, we can conclude $T_{comp} = T_{comp\_read} + T_{comp\_comp} + T_{comp\_write}$

Under parallel conditions, each computation process takes $T_{calc}$ to deal with a compressed data block with the size of $K/μ$. $T_{calc}$ consists of three parts:

1. $T_{calc\_read}$: duration of reading a compressed data block with the size of $K/μ$
2. $T_{calc\_uncom}$: duration of decompressing the compressed data block from the size of $K/μ$ to the size of $K$
3. $T_{calc\_calc}$: duration of calculating the uncompressed data with the size of $K$

Therefore, it concludes $T_{calc} = T_{calc\_read} + T_{calc\_uncom} + T_{calc\_calc}$

In general, the K-means algorithm needs to iterate many times before convergence. Therefore, only in the first iteration is the data compression part necessary. In the subsequent iterations, it just uses the compressed data generated in the first iteration. We will analyze the first iteration and subsequent iterations respectively.

### 3.2 Analysis for the first iteration

In order to facilitate our analysis, we draw a timing diagram for compression processes and computation processes according to three different situations, as shown in Figure 2.

In the Figure 2, Comp, Calc and Idle respectively denotes the compression, computation and idle. The green, blue and red box respectively indicates the time taken by each compression process to deal with a raw data block, the time taken by each computation process to deal with a compressed data block and the idle time of compression/computation processes.

We assume that there exists a constant $C$, such that $T_{comp} = C \cdot T_{calc}$. If and only if this constant satisfies $C = \frac{M}{N-M}$, namely $\frac{T_{comp}}{T_{calc}} = \frac{M}{N-M}$, $N - M$ computational processes need $M$ compression processes to supply the compressed data. At this point, the processing rates of the compression process and computation process are consistent. Only in this situation, the compression process resources and computation process resources can be fully utilized.

When $C > \frac{M}{N-M}$, as shown in the Figure 2(a), the computation process is faster than the compression process. Computation waits for compression, and the value of $M$ is smaller than that in the optimal situation. When $C < \frac{M}{N-M}$, as shown in the Figure 2(b), the compression process is faster than the computation process. Compression waits for computation, and $M$ is larger than that in the optimal situation.

When firstly $C = N : 0(M = N)$, as shown in the Figure 2(c), there are no computation processes and all the $N$ processes work as compression processes to deal with all the raw data. Then $C = 0 : N(M = 0)$, there are no compression processes and all the $N$ processes work as computation processes.

Based on the above analysis, we can get the formula of
time $T_1(M)$ for the first iteration.

$$
T_1(M) = \begin{cases} 
\left[ \frac{D}{N M} \right] * T_{\text{comp}} + f(D/K(M-N)) * T_{\text{calc}} & 1 \leq M \leq \frac{C}{C+1} * N \\
\left( \frac{D}{N M} \right) * (T_{\text{comp}} + T_{\text{calc}}) & M = N 
\end{cases}
$$

where the function $f(x)$ is:

$$
f(x) = \left\{ \begin{array}{ll} 
\frac{x}{N-M+1} & x \neq 0 \\
1 & x = 0 
\end{array} \right.
$$

$T_1(M)$ corresponds to three different situations in Figure 2(a), Figure 2(b) and Figure 2(c). $T_{\text{comp}}$ and $T_{\text{calc}}$ are related to the value of $M$, and they follow a non-linear relationship. To facilitate the analysis, we perform a polynomial fitting $T_{\text{comp}}$ and $T_{\text{calc}}$ versus $M$. In order to avoid overfitting, the order of the fitted function is set to three. Therefore, we can draw the curve of function $T1(M)$ as shown in the Figure 3:

![Figure 3: Time $T_1(M)$ for 1st Iteration](image)

In Figure 3, the parameter $D$ is set as 1141.875, $T_{\text{comp}}$ and $T_{\text{calc}}$ are substituted by the fitted functions $T_{\text{comp}}(M)$ and $T_{\text{calc}}(M)$; the parameter $C$ is set as $\frac{T_{\text{comp}}}{T_{\text{calc}}}$; the number of processes $N$ is set as 336. The figure illustrates that:

1) When the computation process is faster than the compression, the trend of the red curve is relatively flat. The number of compression processes $M$ has little effect on the time $T_1(M)$;

2) When the compression process is faster than the computation process, the green curve shows a slower increasing trend with the increasing number of compression processes. When $M$ is close to 280, the curve shows a sharp increasing trend. The number of compression processes $M$ has a positive effect on the time $T_1(M)$.

3) When the compression is conducted before the computation, $T_1(M)$ will be a constant and is a little larger than the other two cases.

Based on this analysis, we conclude that the appropriate choice of $M$ can effectively improve the performance of parallel K-means application.

### 3.3 Analysis for all iterations

1) Analysis with compression

In the second and subsequent iterations, the execution processes are similar, and it is not necessary to compress the raw data. All $N$ processes work as the computation processes.

Under parallel condition, each computation process takes $T_{\text{calc}}$, seconds to deal with a compressed data block with the size of $K/\mu$, $T_{\text{calc}}$ consists of three parts

1) $T_{\text{calc,read}}$: duration of reading a compressed data block with the size of $K/\mu$.

2) $T_{\text{calc,uncom}}$: duration of decompressing the compressed data block from $K/\mu$ to $K$.

3) $T_{\text{calc,calc}}$: duration of calculating the uncompressed data block with the size of $K$.

Therefore, we conclude $T_{\text{calc}} = T_{\text{calc,read}} + T_{\text{calc,uncom}} + T_{\text{calc,calc}}$ and the time taken by the $i^{th}$ iteration is:

$$
T_i = \left[ \frac{D/\mu}{(K/\mu) * N} \right] * T_{\text{calc}}
$$

where, $S$ indicates the total number of iterations of parallel K-means application. Based on the above analysis, the total time $T_{\text{total,com}}(S)$ taken by the application throughout the whole execution process can be formulated as:

$$
T_{\text{total,com}}(S) = T_{1}^* + \sum_{i=2}^{S} \left[ \frac{D/\mu}{(K/\mu) * N} \right] * T_{\text{calc}}
$$

(2) Analysis without compression

When the compression is not adopted, each of iterations of the application has the similar execution process, and all the $N$ processes work as computation processes.

Under parallel condition, each computation process takes $T_{\text{calc}}$, seconds to deal with a raw data block with the size of $K$. To avoid ambiguity between formula with compression and without compression, we use $T_{i}^*$ here instead of $T_{\text{calc}}$. $T_{\text{calc}}$ consists of two parts

1) $T_{\text{calc,read}}$: duration of reading a raw data block with the size of $K$.

2) $T_{\text{calc,calc}}$: duration of calculating the raw data block with the size of $K$.

Therefore, it concludes $T_{\text{calc}} = T_{\text{calc,read}} + T_{\text{calc,calc}}$.

Based on the above analysis for various iterations, the total time $T_{\text{total}}(S)$, without compression, taken by the program throughout the whole execution process can be formulated as:

$$
T_{\text{total}}(S) = \sum_{i=1}^{S} \left[ \frac{D}{K * N} \right] * T_{\text{calc}}
$$

### 3.4 Compression contribution model

To determine the circumstances, under which the compression can improve the performance of the application, we perform subtraction between $T_{\text{total}}(S)$ and $T_{\text{total,com}}(S)$ as below:

$$
T_{\text{total}}(S) - T_{\text{total,com}}(S) = S * \left( \left[ \frac{D}{K * N} \right] * (T_{i}^* - T_{\text{calc}}) \right)
$$

$$
+ \left[ \frac{D}{K * N} \right] * T_{\text{calc}} - T_{i}^*
$$

(7)
If the compression is able to improve the performance of the program, the difference between \( T_{\text{total}}(S) \) and \( T_{\text{total,com}}(S) \) must satisfies:

\[
T_{\text{total}}(S) - T_{\text{total,com}}(S) > 0 \quad (8)
\]

Therefore, the number of iterations \( S \) must satisfies:

\[
S > \frac{T_{\text{total}} - \left( \frac{D}{K*N} * T_{\text{calc}} \right)}{\frac{D}{K*N} * (T_{\text{calc}} - T_{\text{calc}})} \quad (9)
\]

In order to facilitate the comparison, we draw the curve of \( T_{\text{total}}(S) \), as well as the curve of \( T_{\text{total,com}}(S) \) versus the number of iterations, \( S \): It can be seen from the Figure 4, the application with compression takes more time for the first iteration. With the increasing number of iterations, the application with the compression starts to takes less time than that without compression, and the gap between two curves will be larger. We can conclude that the more the number of iterations, the higher the performance improvement the compression will yield. Especially for those applications with high real-time requirements, using the compression can effectively improve the performance when dealing with large-scale data and many rounds of iterations.

4. MEASUREMENT AND ANALYSIS

4.1 Analysis for the first iteration

In our experiments, we use a computing cluster with 336 cores to run a parallel program for the K-means clustering algorithm that is implemented by a message-passing interface programming model. Table 3 lists the initial value of environmental parameters. LZO, LZMA, Gzip and Bzip in the Table 3 indicates four different compression algorithms respectively, used by the following measurements and analysis. Initial values of parameters are measured on our computing clusters and may not be the same as those of others.

(1) Analysis on the first compression then computation

(Use LZO only)

In this situation see Figure 2(c), there is no coexistence for compression processes and computation processes and no direct interaction between them. Therefore, the time \( T_{\text{comp}} \) taken by compression processes to deal with a raw data block and the time \( T_{\text{calc}} \) taken by computation processes to deal with a compressed data block can be viewed as fixed values. Then the total execution time of the application can be formulated as:

\[
T_{\text{total,com}}(S) = \left( \frac{D}{K*N} \right) * T_{\text{comp}} + \sum_{i=1}^{S} \left( \frac{D}{(K*N)} \right) * T_{\text{calc}}
\]

(10)

In order to facilitate the comparison, we draw three curves:

1) Total execution time without the compression predicted by Formula 6, \( T_{\text{total}}(S) \)
2) Total execution time with the compression predicted by Formula 10, \( T_{\text{total,com}}(S) \)
3) Total execution time with the compression by real measured data.

It can be shown in Figure 5, the application with the comp-

Table 3: Initial Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K )</td>
<td>0.1123GB</td>
</tr>
<tr>
<td>( D )</td>
<td>1141.875GB</td>
</tr>
<tr>
<td>( N )</td>
<td>336</td>
</tr>
<tr>
<td>( M )</td>
<td>14 336</td>
</tr>
<tr>
<td>avai_band</td>
<td>0.08969 GB/Sec</td>
</tr>
<tr>
<td>uncom_speed</td>
<td>LZMA: 0.0064 GB/Sec</td>
</tr>
<tr>
<td></td>
<td>Gzip: 0.0053 GB/Sec</td>
</tr>
<tr>
<td></td>
<td>Bzip: 0.0038 GB/Sec</td>
</tr>
<tr>
<td>( \mu )</td>
<td>LZO: 2.313</td>
</tr>
<tr>
<td></td>
<td>LZMA: 5.004</td>
</tr>
<tr>
<td></td>
<td>Gzip: 4.107</td>
</tr>
<tr>
<td></td>
<td>Bzip: 5.646</td>
</tr>
</tbody>
</table>

Figure 4: Time with/without compression

Figure 5: Time with/without compression
versus the number of compression processes. Figure 6 shows

that, when the number of compression processes is less than 280, the pink curve is more consistent with the red curve than the green one. When the number of compression processes is greater than 280, the pink curve is more consistent with the green curve than the red one. After processing the measured data, we found that when the number of compression processes is less than 280, the measured data for $T_{\text{comp}}$ and $T_{\text{calc}}$ satisfy the following inequality, which means that the computation process is faster than the compression.

$$\frac{T_{\text{comp}}}{T_{\text{calc}}} > \frac{M}{N - M}$$

When the number of compression processes is greater than 280, the measured data for $T_{\text{comp}}$ and $T_{\text{calc}}$ satisfy the following inequality, which means that the compression is faster than the computation.

$$\frac{T_{\text{comp}}}{T_{\text{calc}}} < \frac{M}{N - M}$$

Based on the above discussion, we can conclude that the $T_1(M)$ function is consistent with the measured data and provides an accurate description on the relationship between the execution time of the first iteration and the number of compression processes. $T_1(M)$ can also provide a strong criterion to set an appropriate number of compression processes to make the execution time of the first iteration minimum.

### 4.2 Analysis for all iterations

1. Analysis on the number of iterations

   (Use LZO only)

   In order to facilitate the analysis of multiple iterations, we draw predicted curves in the Figure 7. As Figure 7 shows, for the first iteration, curves without compression are higher than that with compression. For the second and subsequent iterations, application does not need to compress data. Then curves with compression become flatten and are lower than those without compression. As the number of iteration increases, the gap between curves with compression and those without compression become larger and larger. The performance improvement will be more significantly as the number of iterations increases.

2. Analysis on the compression ratio

   Empirically, the higher the compression ratio, the lower the compression speed. Assuming that the compression ratio $\mu$ and the decompression speed $\text{uncom\_speed}$ satisfy the formula $\mu = \frac{\text{comp\_coeff}}{\text{uncom\_speed}}$, where $\text{comp\_coeff}$ is a coefficient related to the specific compression algorithm. The formula of the total execution time for the application can be transformed to:

$$T_{\text{total\_com}}(S) = T_1^* + \sum_{i=2}^{S} \left[ \frac{D}{K \ast N} \right]$$

When the application iterates multiple times, to facilitate the analysis, the performance improvement of the first iteration by the compression can be negligible and $T_1^*$ can be viewed as a fixed value. Based on the above assumptions and the formula, we can see that compression only has influence on $T_{\text{calc\_read}}$ and $T_{\text{calc\_uncom}}$. To intuitively analyze the influence on the total execution time by the compression ratio, we draw the surface for the sum of $T_{\text{calc\_read}}$ and $T_{\text{calc\_uncom}}$ versus the compression ratio and compression coefficient $\text{comp\_coeff}$ as shown in the Figure 8.
algorithm, LZMA algorithm, Gzip algorithm, and Bzip algorithm, respectively. Among these compression algorithms, the LZMA algorithm outperforms others.

(3) Analysis on the number of processes. (Use LZO only) In Figure 9, we can see that the total execution time shows a steady decrease as the number of processes increases. Especially, when the number of processes is small, the performance improvement is significant. As the number of processes exceeds a certain number, the performance improvement is no longer very clear. In addition, compression greatly reduces the execution time taken by the application. When the number of processes is small, compression does not improve the performance of the program. As the number of processes increases, the gap between curves with the compression and those without the compression becomes larger and larger.

(4) Analysis on the data size. (Use LZO only) In the Figure 10, it is shown that two predicted time curves is approximately proportional to the data size. As the amount of data increases, the gap between the curves with the compression and those without compression becomes larger and larger. When the amount of data approaches to 1,000GB, the time taken by MPI application with compression is approximately only half of that without the compression. However, the measured time taken by MapReduce application with the compression is not reduced significantly.

When the amount of data is less than 400GB, without compression, the measured curve by MPI application shows different performance when compared with the predicted result, due to caching. When the cache can hold all the data, the second and subsequent iterations no longer need the disk I/O operations. Therefore, the measured curve by MPI is significantly lower than the predicted one. As the amount of data increases, until the cache is not able to hold all the data, the cache will not contribute to the performance improvement between two neighboring iterations.

Therefore, an effective use of caching can greatly improve the performance of the application. Actually, we can adopt a trick of the cache in the parallel K-means algorithm to improve its performance. Suppose that the order of data blocks accessed in the $i^{th}$ iteration is: $1, 2, \cdots , P$, then the order of data blocks accessed in the $(i+1)^{th}$ iteration can be set to: $P, \cdots , 2, 1$. In this way, the $(i+1)^{th}$ iteration can use the left data of the $i^{th}$ iteration in memory, and thereby reduce the I/O overhead to enhance the overall performance of the program. Experiments show that this trick can achieve a 30% reduction in disk I/O overhead.

5. CONCLUSIONS

Taking parallel K-means clustering as an example, this paper proposed a compression based clustering model. The experimental results show that the compression can significantly alleviate the increasingly prominent I/O bottleneck occurred in the big data analysis. We investigated several factors related to compression, including the data size, the compression ratio, the compression/decompression speed, the computing resource performance and when to use the compression. Our experiments were run on terabytes of data via computing clusters of hundreds of cores. The measurement result shows that integrating compression into big data analysis yields significant performance improvement. This model is capable of providing effective decision support for how to use the compression to improve I/O performance.

The parallel K-means clustering task only needs a small amount of network transmission for data. Therefore, we only analyzed the impact on the local disk I/O of using the compression. In fact, for many applications, network transmission of data takes up a large proportion in I/O overhead and plays an important role. One of our future research directions is to extend our compression contribution model to tackle with the network transmission intensive applications to reduce network I/O. In addition, we plan to apply the proposed model to other big data analysis algorithms to further validate it.

6. ACKNOWLEDGMENTS

This work is supported by the Natural Science Foundation of China(NSFC) under Grant No. 61003338, the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06010202 and XDA05150401, the National R&D Infrastructure and Facility Development Program of China under Grant No. BSDN2009-17 and BSDN2009-18, the Knowledge Innovation Program of Chinese Academy of Sciences under Grant No. CNIC_QN_11005.

7. REFERENCES


