Improving the I/O Throughput for Data-Intensive Scientific Applications with Efficient Compression Mechanisms

Dongfang Zhao[⋆†], Jian Yin[†], Ioan Raicu[⋆‡]
[⋆]Department of Computer Science, Illinois Institute of Technology
[†]Data Intensive Scientific Computing Group, Pacific Northwest National Laboratory
[‡]Mathematics and Computer Science Division, Argonne National Laboratory

1. INTRODUCTION

Today's science is generating significantly larger volume of data than before. Data compression can potentially improve application performance. However, in many scientific applications and especially in large scale parallel scientific applications, each process often just accesses parts of the data. This can result in some data that are decompressed by a process but not used. General compression libraries (e.g. LZO [1], bzip2 [2] and zlib [3]) do not consider chunk size in the context of parallel and distributed file systems. Indexing is a widely used technique for online scientific encoding and query e.g. [4–6], even though it would not yield a high compression ratio for large chunks. Some mechanisms [7–9] were proposed by providing user level libraries, indicating the application developers need to modify the application and/or the high-level library.

We propose two techniques to leverage compression to improve the performance of large scale parallel applications. First, we enable decompression from the middle of the chunk and stop decompression after we extract the data that we need, which eliminates the overhead of decompressing the data that is not needed by the process. Second, we build compression into the parallel file system which allows caching and prefeteching to be seamlessly integrated and allows applications to transparently leverage compression. Caching decompressed data allows the data to be accessed in later points.

2. METHOD

The procedure to compress a file is described in Algorithm 1. We assume the file content could be split into a logical array. The first phase of the compression algorithm is to convert the original data entries to increments by XORing every pairs of neighbor entries in the original file, as shown in Lines 2 - 4. The number of leading zeros is stored in the first 5 bits of the encoded data (more details in [7]). The second phase is to append P reference points to the end of the compressed file, as shown in Lines 6 - 8.

The decompression is described in Algorithm 2. The nearest upper reference point is located as IDX at Line 3, and GAP indicates the distance between the user-requested starting address and the reference point. Lines 5 - 8 restores the original data points by incrementally XORing from the reference point.

We will show how many reference points to pick for to achieve the optimal end-to-end throughput. Here, by "end-to-end", we consider the overall I/O time for compression (when write) and decompression (when read) in the worst case (e.g. reading the last record).

Algorithm 1 Compress a file

Require: F^d is the original file to be compressed; F^e is the name of the compressed file; P is the number of partitions to be applied to the original file
Ensure: F^e could be used to recover the content of F^d
1: SIZE = F^d.size()
2: **for** (int i = 1; i < SIZE; i++) **do**
3: F^e[i] = F^d[i] **xor** F^d[i - 1]
4: **end for**
5: BS = SIZE / P
6: **for** (int j = SIZE; j <= SIZE + P; j++) **do**
7: F^e[j] = F^d[BS * (j - SIZE)]
8: **end for**

Algorithm 2 Decompress a (portion of) file

Require: F^e is the encoded file to be decompressed; F^{d-} is the decompressed data for the chunks to be decompressed; P is the number of partitions to be applied to the original file; $BASE$ is the starting point from where to decompress; LEN denotes the number of data entries to be decompressed
Ensure: F^{d-} is identical to same portion of the original file
1: SIZE = F^e.size() - P
2: BS = SIZE / P
3: IDX = BASE / BS
4: GAP = BASE % BS
5: F^{d-}[0] = F^e[SIZE + IDX]
6: **for** (int i = 1; i < GAP + LEN; i++) **do**
7: F^{d-}[i] = F^{d-}[i - 1] **xor** F^e[BASE - GAP + i]
8: **end for**

Table 1: Terminology

Variable	Description
B_r	Read Bandwidth
B_w	Write Bandwidth
S	File Size
N	Number of Data Entries
R	Number of Reference Points

We define the variables in Table 1. The overhead to write the extra R reference points is

$$T_{compress} = \frac{R \cdot S}{N \cdot B_w},$$

and we would save the following time in decompression:

$$T_{decompress} = \frac{S - S/R}{B_r}.$$

Now, to maximize $F(R) = T_{decompress} - T_{compress}$. By taking the derivative on R (suppose R is continuous) we have

$$\hat{R} = \sqrt{\frac{N \cdot B_w}{B_r}}.$$

Note that the second derivative of $F(R)$ is negative and R is a non-negative integer, so the optimal R is:

$$\arg\max_R F(R) = \begin{cases} \lfloor \hat{R} \rfloor & \text{if } F(\lfloor \hat{R} \rfloor) > F(\lceil \hat{R} \rceil) \\ \lceil \hat{R} \rceil & \text{otherwise} \end{cases}$$

3. DESIGN AND IMPLEMENTATION

We designed a user-level filesystem prototype called Multi-Reference Compression FileSystem (MRC-FS) with a transparent compression/decompression layer. It leveraged the FUSE [10] framework, and mounted each local compute node to the remote GPFS [11] filesystem. The system was implemented in about 3,000 lines of C/C++ code and Shell script. The compression algorithm was implemented in the *mrc_write()* interface, that is the handler for catching the write system calls in MRC-FS. *mrc_write()* compressed the raw data, cached it in the memory if possible, and wrote the compressed data into the filesystem. The decompression algorithm was implemented in the *mrc_read()* interface, similarly. When a read request came in, this function loaded the compressed data (either from the cache or the disk) into memory, applied the decompression algorithm to the compressed data and passed the result to the end users.

4. EVALUATION

We evaluated the system on the IBM BlueGene/P supercomputer. We used up to 256 compute nodes (1024-core), each of which has a FUSE mount point to the remote GPFS file system of 128 storage nodes. The dataset is 244.25GB of climate data from the high-resolution Global Cloud-Resolving Model (GCRM).

The **throughput** when enabling the compression layer is about 1.37X faster than the original data, when tested with 1 – 2000 reference points. The overhead introduced by the compression layer is smaller than 10%. We measured the total I/O time from compressing the entire data set to retrieving the latest (i.e. the last one to be compressed) temperature. This is supposed to be the **worst case I/O time**. We observed that a single reference point improved the I/O time from 748 seconds to 501 seconds, and multiple reference points (200 – 2000) further improved the I/O time to below 348 seconds. We finally ran a **real application** $MMAT$ that calculated the minimal, maximal and average temperatures on the GCRM dataset, when compressing the data with $R = 800$ and the raw data. The former case resulted in a $\frac{180.97}{146.45} = 1.24X$ speedup on the overall execution time.

5. CONCLUSION AND FUTURE WORK

We proposed a new compression mechanism particularly for scientific data with high compression ratio and low computation overhead. We implemented the compression layer at the filesystem level that significantly improved the end-to-end I/O throughput.

We are planning to apply MRC to HyCache [12, 13] to further improve the caching effect for distributed file systems in general. Particularly, we will integrate MRC into FusionFS [14, 15], which is a new distributed file system targeting exascale [16] with unique features such as distributed metadata [17], high reliability through erasure coding [18], and efficient data provenance [19, 20].

Acknowledgement

This work was supported in part by the Office of Biological and Environmental Research, Office of Science, U.S. Department of Energy, under contract DE-ACO2-O6CH11357.

References

[1] LZO real-time data compressoin library. http://www.oberhumer.com/opensource/lzo.

[2] bzip2. http://www.bzip2.org.

[3] zlib. http://www.zlib.net.

[4] S. Lakshminarasimhan et al. Scalable in situ scientific data encoding for analytical query processing. IEEE HPDC '13.

[5] Z. Gong et al. Multi-level layout optimization for efficient spatio-temporal queries on isabela-compressed data. IEEE IPDPS '12.

[6] J. Jenkins et al. Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *DEXA (2)*, pages 16–30, 2012.

[7] T. Bicer et al. Integrating online compression to accelerate large-scale data analytics applications. IEEE IPDPS '13.

[8] E. R. Schendel et al. Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization. HPDC '12.

[9] J. Jenkins et al. Byte-precision level of detail processing for variable precision analytics. ACM/IEEE SC '12.

[10] FUSE Project. http://fuse.sourceforge.net.

[11] F. Schmuck et al. GPFS: A shared-disk file system for large computing clusters. FAST '02.

[12] D. Zhao et al. HyCache: a user-level caching middleware for distributed file systems. IEEE IPDPSW '13, 2013.

[13] D. Zhao et al. HyCache: A hybrid user-level file system with ssd caching. GCASR '12, 2012.

[14] D. Zhao et al. FusionFS: a distributed file system for large scale data-intensive computing. GCASR '13, 2013.

[15] D. Zhao et al. Distributed file systems for exascale computing. ACM/IEEE SC'12 Doctoral Showcase.

[16] D. Zhao et al. Exploring reliability of exascale systems through simulations. ACM/SCS HPC '13.

[17] T. Li et al. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. IEEE IPDPS '13.

[18] D. Zhao et al. Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms. IEEE CLUSTER '13.

[19] D. Zhao et al. Distributed data provenance for large-scale data-intensive computing. IEEE CLUSTER '13, 2013.

[20] C. Shou et al. Towards a provenance-aware distributed filesystem. 5th USENIX TaPP Workshop, 2013.