Accelerating Lossy Compression on HPC datasets via Partitioning Computation for Parallel Processing

Xiangyu Zou[1,2], Tao Lu[3], Sheng Di[4], Dingwen Tao[5], Wen Xia[1,2],
Xuan Wang[1], Weizhe Zhang[1,2], and Qing Liao[1,2]

[1] Harbin Institute of Technology, Shenzhen, China
[2] Peng Cheng Laboratory, Shenzhen, China
[3] Marvell Technology Group, USA
[4] Argonne National Laboratory, IL, USA
[5] University of Alabama, AL, USA

Abstract*—Recently, increasingly attention has been paid to data reduction in high-performance computing (HPC) environments where continually produce a large volume of data every second during scientific simulations. Unlike the traditional data reduction techniques (such as deduplication or lossless compression), error-controlled lossy compression not only significantly reduces data size, but also can hold the promise to satisfy user's requirements on error control. Point-wise relative error bounds (i.e., compression errors depends on the data values) are widely used by many scientific applications in the lossy compression, since error control can adapt to the precision in the dataset automatically. SZ lossy compressor has been one of the best choices for HPC data reduction due to its high compression ratios while meeting data precision requirements. Currently, high compression rate is also strongly demanded because of fairly high data production rate of many applications. In this work, we aim to accelerate the SZ compressor significantly by developing a parallel model in terms of the point-wise relative error bound, because this type of error bound has been widely used in the community while SZ suffers relatively low compression/decompression rate in this case. It is non-trivial to parallelize SZ because of strong *data dependency* in the SZ compression. To address this issue, we develop a *pipeline-like* method and exploit a series of strategies to parallelize the logarithmic transformation and prediction + quantization stages for SZ. We develop a parallel computing framework, called ParaSZ, that efficiently exploits parallelism of SZ compression in logarithmic transformation stage as well as prediction and quantization stage. Our evaluation with real-world scientific simulation datasets suggest that ParaSZ greatly accelerates the computation tasks of SZ in most cases while without sacrificing the compression ratio.

***Index Terms*—Lossy compression, high-performance computing, scientific data, compression rate**

I. Introduction

Scientific simulations in high-performance computing (HPC) environments are producing vast volume of data, which may consume huge storage space and cause severe I/O bottlenecks during the simulation [1]–[3]. For instance, there are 260 TB of data generated across one ensemble every 16 seconds, when estimating even one ensemble member per simulated day [4]. The data volume and data movement rate are imposing unprecedented pressure on storage and interconnects [5], [6] in HPC environments.

As HPC storage infrastructure is being pushed to the scalability limits in terms of both throughput and capacity [7], the communities are striving to find new approaches to lower the storage cost. Data reduction, among others, is deemed to be a promising candidate by reducing the amount of data moved to storage systems. Data deduplication and lossless compression have been widely used in general-purpose systems to reduce data redundancy. In particular, deduplication [8] eliminates redundant data at the file or chunk level, which can result in a high reduction ratio if there are a large number of identical chunks at the granularity of tens of kilobytes. For scientific data, this rarely occurs. It was reported that deduplication typically reduces dataset size by only 20% to 30% [9], which is far from being useful in production. On the other hand, lossless compression in HPC was designed to reduce the storage footprint of applications, primarily for checkpoint/restart. Shannon entropy [10] provides a theoretical upper limit on the data compressibility. But simulation data often exhibits high entropy, which results in low lossless compression ratio (usually less than 2.0) on HPC datasets [11]. With growing disparity between compute and I/O, more aggressive data reduction schemes are needed to further reduce data by an order of magnitude or more [4], so the focus has shifted towards lossy compression recently.

Recently, lossy compression for HPC datasets has been widely used and studied [12], [13]. For example, ZFP and SZ have been widely recognized as the top two error-controlled lossy compressors with respect to both compression ratio and rate [13]–[16]. In general, SZ has higher compression ratio than ZFP with the same data distortion, while suffering from lower compression rate in many cases.

In this paper, we mainly focus on how to accelerate the compression rate for relative-error bounded lossy compression in terms of SZ [17], by developing a parallel processing model without sacrificing the compression ratio. But it is non-trivial to parallelize SZ because of strong *data dependency* during the process of SZ compression. Specifically, SZ involves four

This research was conducted completely using publicly accessible academic resources. It does not reflect viewpoints of Marvell Technology Group.

important stages: data prediction, linear-scaling quantization, Huffman encoding, and lossless compression. The stages 1 and 2 are often the performance bottleneck, while they have strong *data dependency* during the data processing. For instance, the prediction of one data value is dependent on several other precursory data points.

In our solution, we design an efficient parallel compression model in terms of a critical characteristic in the SZ compression principle. Specifically, we observe that in SZ, the multi-dimensional prediction of a data value depends only on a few neighboring data points instead of on the entire dataset. Such an observation motivates us to develop a *pipeline-like* method to accelerate the data compression. Our evaluation with real-world scientific simulation datasets suggest that ParaSZ efficiently accelerates the computation tasks of SZ in most cases while without sacrificing the compression ratio.

The remainder of this paper is organized as follows. In Section II, we discuss the related work. In Section III, we discuss details about problems and challenges. In Section IV, we present the design and implement of our approach. In Section V, we evaluate our new scheme with multiple real-world scientific HPC application datasets across different domains, comparing our scheme with the latest SZ. The breakdown of time cost will also be provided to show the reduction of time in different stages. In Section VI, we conclude our work, and discuss the future work in this research domain.

II. Related Work

Lossless compression fully maintains data fidelity, but it depends on the repetition of symbols in the data sources. Even for slightly variant floating-point values, their binary representations may hardly contain identical symbols (or exactly duplicated chunks). As such, lossless compression suffers from very low compression ratio [2], [9], [13], [17], [18] on scientific data. Error-bounded lossy compression creates a new avenue to drastically improve data compression ratio, while still satisfying user-requirement on data distortion.

ZFP [18] follows the classic texture compression for image data. Working in 4^d (where d is the number of dimensions) sized blocks, ZFP first aligns the floating-point data points within each block to a common exponent and encodes exponents. Then, a reversible orthogonal block transform (e.g., discrete cosine transform) is applied to the signed integers in each block. Such a transform is carefully designed to mitigate the spatial correlation between data points, with the intent of generating near-zero coefficients that can be compressed efficiently. Finally, embedded coding [19] is used to encode the coefficients, producing a stream of bits that is roughly ordered by their impact significance on error, and the stream can be truncated to satisfy any user-specified error bound.

Motivated by the reduction potential of spline functions [20], [21], ISABELA [22] uses B-spline based curve-fitting to compress the incompressible scientific data. Intuitively fitting a monotonic curve can provide a model that is more accurate than fitting random data. Based on this, ISABELA first sorts data to convert highly irregular data to a monotonic curve. Its biggest weakness is pretty slow compression/decompression because of its expensive sorting operation.

SZ compressor has experienced multiple revolutions since the very first version 0.1 [23] was released in 2016. SZ 0.1 employed multiple curve-fitting models to compress data streams, with the goal of accurately approximating the original data. SZ 0.1 encoded the bestfit curve-fitting type for each data point or mark the data point as unpredictable data if its value is too far away from any curve-fitted value. SZ 1.4 [2], [24] significantly enhanced the compression ratios by improving the prediction accuracy with a multi-dimensional prediction method plus a linear-scaling quantization method. SZ 2.0 [25] further improved the compression quality for the high-compression cases by leveraging an adaptive method facilitated with two main candidate predictors (Lorenzo and linear regression). Note that the classic SZ framework [2] did not support point-wise relative error bound. As such, Liang et al. [17] proposed an efficient logarithmic transformation to convert the pointwise relative-error-bounded compression problem to an absolute-error-bounded compression problem. However, as we have confirmed in our performance profiling, this will significantly slow down the compression and decompression because of its costly logarithmic transformation operations. SZ 2.1 [26] developed a technology to avoid the time cost in the transform stage by building pre-computed table and combining the quantization stage with the transform stage. And now, the workflow of SZ compressor is described as follow. Firstly, using an efficient logarithmic transformation to convert the point-wise relative-error-bounded compression problem to an absolute-error-bounded floating-point data compression problem, which is further converted a lossless integer data compression problem by prediction and quantization method. And then, through huffman encoding and other lossless compressors, like ZStandard [27] (or called Zstd) and GZip [28]. Thus, SZ finally achieves the highest compression ratio from among all lossy compressors.

ZFP and SZ have been widely recognized as the top two error-controlled lossy compressors with respect to both compression ratio and rate [13]–[16]. They have been adopted on thousands of computing nodes. For example, the Adaptable IO System (ADIOS) [29] deployed on the Titan (OLCF-3) supercomputer at Oak Ridge National Laboratory has integrated both ZFP and SZ for data compression. Although being the two best compressors in class, ZFP and SZ still have their own pros and cons because of distinct design principles. Motivated by fixed-rate encoding and random access, ZFP follows the principle of classic transform-based compression for image data. SZ adopts a prediction-based compression model, which involves four key steps: data prediction, linear-scaling quantization, Huffman encoding and lossless compression. In general, SZ outperforms ZFP in compression ratio (about 2X or more), but SZ is often 20-30% slower than ZFP [13], bringing up a dilemma for users to choose an appropriate compressor in between. In fact, the compression/decompression rate is the same important as the compression ratio, in that many of applications particularly require fast compression

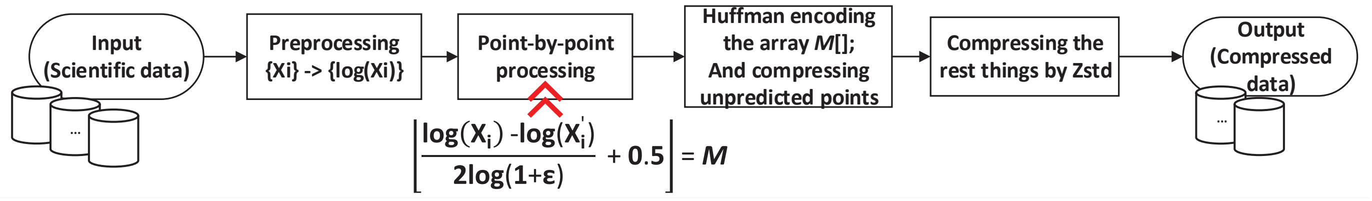


Fig. 1. Workflow of SZ Compressor with log transform.

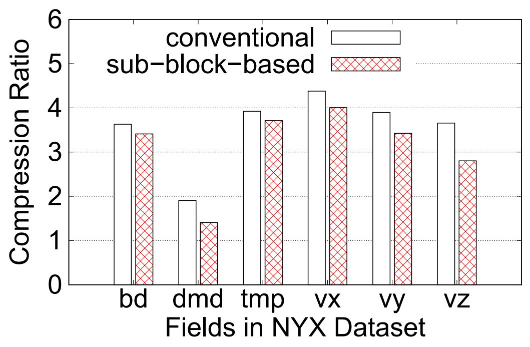


Fig. 2. Compression ratios of *sub-block-based* method ($64 \times 64 \times 64$) and *conventional SZ* methods on NYX dataset with 1E-4 point-wise relative error bound.

at runtime because of extremely fast data production rate such as the X-ray analysis data generated by APS/LCLS [30], [31]. A straight-forward question is can we significantly improve the compression and decompression rates based on SZ lossy compression framework, leading to an optimal lossy compressor for users.

There are also some existing studies working on combining different lossy compressors to obtain better compression quality. Lu et al. [13] conducted a comprehensive evaluation based on SZ and ZFP, and proposed a simple sampling method to select the best compressor with higher compression ratio in between. Tao et al. [15] proposed an efficient online, low-cost selection algorithm that can predict the compression quality accurately for SZ and ZFP in early processing stages and selects the best-fit compression based an important statistical quality metric (PSNR) for each data field. Their work, however, relies on the compression performance of SZ and ZFP, so their compression result would never go beyond the best choice from between SZ and ZFP.

III. Background and Motivation

Compared with other lossy compression schemes, SZ usually achieves higher compression ratio [17], [32]. In general, given a user-set point-wise relative error bound, SZ performs the following five steps (also shown in Figure 1):

- Transforming all data points $\{X_i\}_{i=1}^{n}$ to $\{log(X_i)\}_{i=1}^{n}$ (denoted by $\{Y_i\}_{i=1}^{n}$: converting a point-wise relative error bound problem to absolute error bound problem.
- Applying best-fit prediction to the given dataset based on user-set error bound: each floating-point data value is predicted by its neighbor data points, and then mapped to a quantization factor (integer value).
- Encoding the quantization factors by Huffman coding. And compressing the unpredictable data points (i.e., the data points that cannot be predicted in the second step) by binary-representation analysis.
- Further applying a lossless compressor (such as GZip [28] or Zstandard [27]) onto the compressed bytes generated in the above steps.

We identify that it is feasible to perform parallelism optimization in the first step, because the logarithmic transformation on one data point does not depend on any other data points. However, the second stage (i.e., data prediction and error quantization) is hard to be parallelized because of *data dependency*. As shown in Figure 1, for each data point $\{Y_i\}$ (after log transformation), we compute its predicted value $\{Y_i'\}$ and the quantization factor *M*.

More specifically, data is predicted with Lorenzo predictor [33]. For a 2D dataset, the data point $Y_{i,j}$ is predicted as $Y_{i,j}'$ = $Y_{i,j-1}'$ + $Y_{i-1,j}'$ - $Y_{i-1,j-1}'$; for a 3D dataset, the data point $Y_{i,j,k}$ is predicted as $Y_{i,j,k}'$ = $Y_{i,j,k-1}'$+ $Y_{i,j-1,k}'$ - $Y_{i,j-1,k-1}'$ + $Y_{i-1,j,k}'$ - $Y_{i-1,j,k-1}'$ + $Y_{i-1,j-1,k}'$ - $Y_{i-1,j-1,k-1}'$. In other words, the data points will be processed in the row-column order in 2D and layer-row-column order in 3D. In fact, a quantization factor depends on a predicted value and further depends on its previous predicted values. which is the challenge of *data dependency* for parallelizing SZ compression at the point-wise processing stage.

In order to realize parallel compression, a straight-forward idea is dividing data into relatively small sub-blocks (such as $64 \times 64 \times 64$) and then compressing these blocks separately. However, compressing sub-blocks separately may cause significantly degraded compression ratios. The key reason is that many data points located on the sides of each sub-block cannot be predicted very accurately because of fewer neighboring data points, leading to a more disordered quantization factor array for the following Huffman encoding algorithm to compress. Figure 2 demonstrates that the compression ratios are degraded up to 30% under the independent-sub-block based compression for cosmological NYX simulation dataset [17]. Thus, in this paper, we focus on exploring the feasibility of parallelizing SZ compression in the logarithmic transformation and the prediction & quantization stages without sacrificing any compression ratio.

IV. Design and Implementation

In this section, we provide the design details about our *pipeline-like* method that exploits parallelism in the point-by-point processing stage on 2D datasets for SZ. For simplicity,

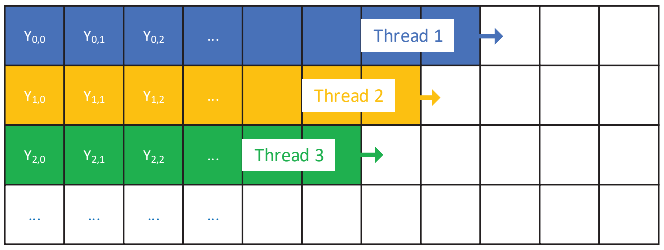


Fig. 3. A general description about parallelizing SZ.

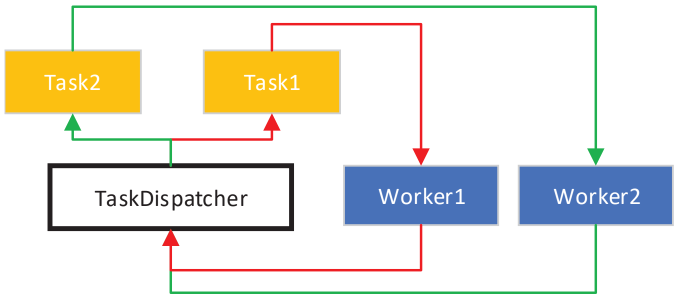


Fig. 4. A general description about *TaskDispatcher*.

we mainly discuss pallelizing SZ with 2D datasets as an example. The methodologies for 3D and 4D datasets are similar to the 2D case.

Observations: We find that boundary data points such as $Y_{1,0}$, $Y_{2,0}$, which locate at the boundary of data fields, are predicted in a different way from the other data points in the prediction and quantization stage. For example, the data point $Y_{i,0}$ is predicted as $Y'_{i-1,0}$. This kind of data value prediction can be exploited for parallel processing. For example, if SZ is predicting the value of $Y_{i,j}$, then the value of $Y_{i,0}$~$Y_{i,j-1}$ must be predicted in advance, and $Y_{i+1,0}$ can be predicted as $Y'_{i,0}$, thus the condition for predicting $Y_{i+1,1}$ is met (i.e., $Y'_{i+1,1}$ = $Y'_{i+1,0}$ + $Y'_{i,1}$ - $Y'_{i,0}$). After predicting $Y_{i+1,1}$, the condition for predicting $Y_{i+1,2}$ is also met (i.e., $Y'_{i+1,2}$ = $Y'_{i+1,1}$ + $Y'_{i,2}$ - $Y'_{i,1}$). Similarly, all of $Y_{i+1,0}$~$Y_{i+1,j-1}$ can be predicted in order.

From the above discussion we know that if in the i^{th} row, $Y_{i,j}$ has been predicted, then in the $(i+1)^{th}$ row, we can predict $Y_{i+1,0}$~$Y_{i+1,j-1}$ concurrently by using a *pipeline-like* method. Figure 3 gives an example on this method on 2D data. We use a thread *T1* to process the first row. We use another thread *T2* to process the second row while controlling the progress of the second row, ensuring that the progress of the 1^{th} row leads (i.e., runs ahead of) the 2^{nd} row. We use a similar way to control the third and subsequent rows. Therefore, we can concurrently process different rows while ensuring the prediction condition is always met.

Implementation: As shown in Figure 4, we design a *TaskDispatcher* - *Worker* model to implement our idea. We split the work in the point-by-point processing stage into several tasks row by row (in 2D and layer by layer in 3D). The *TaskDispatcher* is designed to distribute and schedule the parallel tasks, as well as manage computing and memory resources in our implementation.

Specifically, when a task is constructed, *TaskDispatcher* provides all fields in the data structure *Task* (as shown below).

```
struct Task {
  float* oriData; /* the input data for compression */
  int* factorArray; /* quantization factor array */
  float* prevPredicted; /* predicted values array for
                              the previous row */
  float* curPredicted; /* predict values array for
                              this row */
  int staIndex; /*started index of the task in oriData*/
  int endIndex; /*ended index of the task in oriData*/
  int taskIndex; /* task index */
  int checkWidth; /* reduce lock conflict by checking
            the working progress of the previous row */
  ConcurrentController* prevConcurrentController;
  ConcurrentController* curConcurrentController;
}
```

We control the progress of neighboring rows with conditional variables. So we need *n+1* conditional variables for *n* threads. The data structure *ConcurrentController* is designed for the progress control, which is shown below.

```
Struct ConcurrentController{
  pthread_mutex_t          mutex;
  pthread_cond_t           cond;
  uint32_t        pos; // progress in a row (layer)
}
```

For each *Worker*, it gets a *Task* from *TaskDispatcher*, and then processes a row (indicated as the *Task*) with the progress control, which will be discussed in the next paragraph. Then we conduct the point-by-point processing just like the conventional SZ does.

Figure 5 provides a general workflow of the *pipeline-like* method in SZ with the progress control. Assuming a thread *T1* is processing the i^{th} row and the data point $Y_{i,j}$ has been processed, then thread *T1* needs to check the progress of both the i^{th} and $(i-1)^{th}$ rows from the *ConcurrentController*. In *ConcurrentController*, thread *T1* can check the progress of thread *T2* in the $(i-1)^{th}$ row. If *ConcurrentController.Pos* is not larger than *j+1*, it means thread *T2* has not processed the data $Y_{i-1,j+1}$, thus the data $Y_{i,j+1}$ can not be predicted, thus thread *T1* is required to wait thread *T2* through *ConcurrentController.ConditionVariable* until it is waken up by the thread *T2*. When the thread *T2* processes the data $Y_{i-1,j+1}$, thread *T2* should update its progress in *ConcurrentController.Pos*, and wake up thread *T1* through *ConcurrentController.ConditionVariable* too. Finally, thread *T1* can proceed to process the data $Y_{i,j+1}$. In this way, multiple threads can concurrently process different rows in a pipeline way.

Because the frequent progress-checking (when each data point is processed) will cause serious lock conflicts, batch checking is introduced to improve performance of our parallelizing SZ approach. In our implementation, we use a user-specified parameter *checkWidth* to define the size of a batch. As the method describe above, frequently progress checking will cause serious lock conflicts. Therefore, a thread should check progress less, and we let it check a batch at one time. It means if a thread found that it is allowed to process the next batch, it will continue to process the next **checkWidth** data, otherwise to wait condition variable signal which shows the next **checkWidth** data could be processed. In this way, we

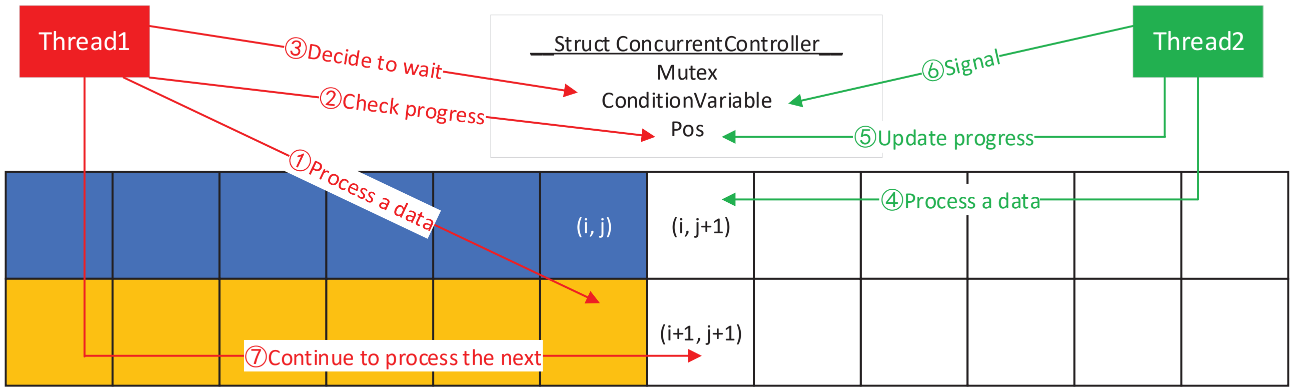


Fig. 5. Workflow of the *pipeline-like* method in SZ.

decrease the times of checking progress, and reduce the lock conflicts when running our parallel SZ appraoach.

V. Evaluation

In this section, we compare our approach (denoted by SZ_M and SZ_M2 means 2 threads, SZ_M4 means 4 threads, etc.) with original SZ (denoted by SZ_T). Compression rate, and the break-down of compression time are provided and compared in quantitative ways. In addition, SZ_M needs a new configuration item named 'check_width', which is the **checkWidth** as defined and discussed in Section IV, and should be set in the configuration file. A bigger **checkWidth** will lead to less lock conflict, but maybe reduce the advantage bringing from parallelism. Firstly we will do a serial of examination to demonstrate the influence of **checkWidth**, and then according to our observation, we will choose a recommended value for **checkWidth**. Finally we will use the recommended value in our final experiments for a better performance.

A. Experimental Setup

We conduct our tests on a server with four 2.4GHz Intel Xeon E5-2640 v4 processors and totally 256GB memory. The HPC datasets are from multiple domains, like NYX cosmology simulation (3D), Hurricane ISABEL simulation (3D) and CESM-ATM climate simulation (2D). The sizes of these four datasets are 3.1GB, 1.9GB, and 2.0GB per snapshot for each application, respectively. We respect data dimensions during compression. For example, NYX cosmology simulation dataset is 3D, so we conduct 3D compression on it.

Because our new parallelism framework will generate exactly the same quantization code in the prediction and quantization stage, thus the compression ratio and data fidelity of our new approach will be totally the same as the original SZ method and we no longer need to concern these two things. Hence we mainly focus on compression rate in the evaluation: here compression rate indicates the throughput of compression. For the metric of compression rate, we run each test five times to get the average values. Since each application involves many fields, each in a data file, we use the aggregated file size to divide by the total compression time cost to calculate the rate.

TABLE I
Compression ratios on three tested datasets with given point relative error bound.

Datasets	1E-01	1E-02	1E-03	1E-04
NYX (3.1 GB)	14.20	9.00	5.25	3.40
Hurricane (1.9 GB)	18.71	11.01	7.07	4.67
CESM (2.0 GB)	82.89	34.01	15.63	8.13

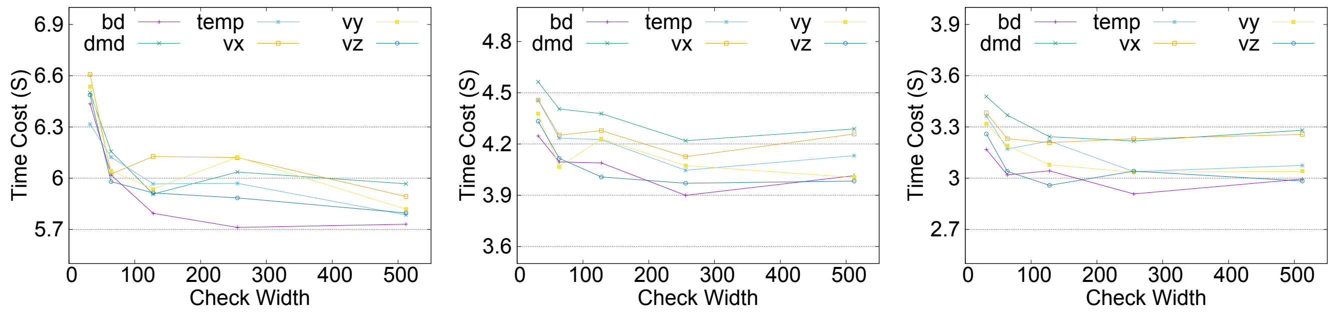
In addition, SZ_P needs a new configuration item named *checkWidth*, which is the batch size, as discussed in Section IV, and should be set in the configuration file. According to our observation, using the very large or very small *checkWidth* both will lead to low compression rate, and we will carefully study and discuss the value of *checkWidth* firstly and use a reasonable *checkWidth* in following experiments.

B. Influence of textbfcheckWidth

In this subsection, we compare different *checkWidth* on 6 fields in NYX dataset and try to demonstrate the influence of *checkWidth*. Several settings of threads is involved and the result is provided in Figure 6.

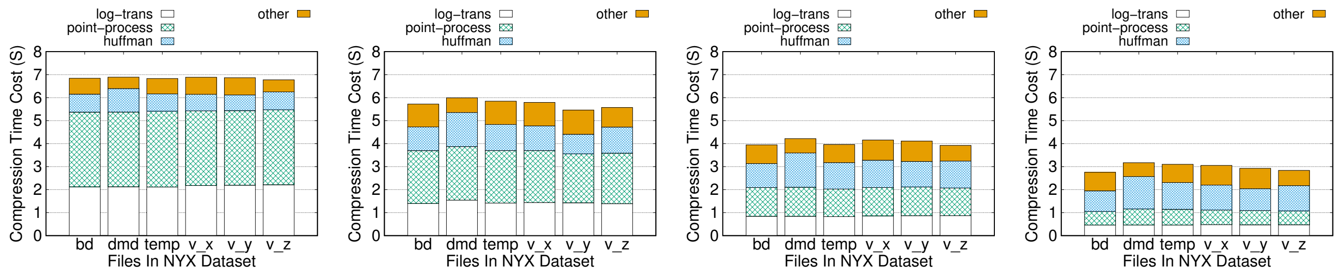
We can find that a smaller *checkWidth* will lead to larger time cost, because different threads unlikely have exactly the same processing speed, therefore as shown in Figure 5, small *checkWidth* will cause frequent lock conflicts. And when a lock conflict happens, there will be several system calls produced, which is always a performance killer (i.e., performance degradation). To mitigate this effect, we can use a larger *checkWidth*, which will give every thread a larger buffer area to avoid the situation that the progress of one thread is caught up by its next thread. The result of experiments also supports this idea, and from Figure 6 we can find that increasing the size of *checkWidth* really reduce the time cost of the total task accordingly.

On the other hand, a larger *checkWidth* is not always better for decreasing the time cost. We also can find that the large *checkWidth* brings negative influence. Specifically, a larger *checkWidth* will cause a larger difference of progress between threads, which leads to more insignificance lock waiting and could not make full use of computing source. Because of the pipeline-like design, the large *checkWidth* will cause a quite



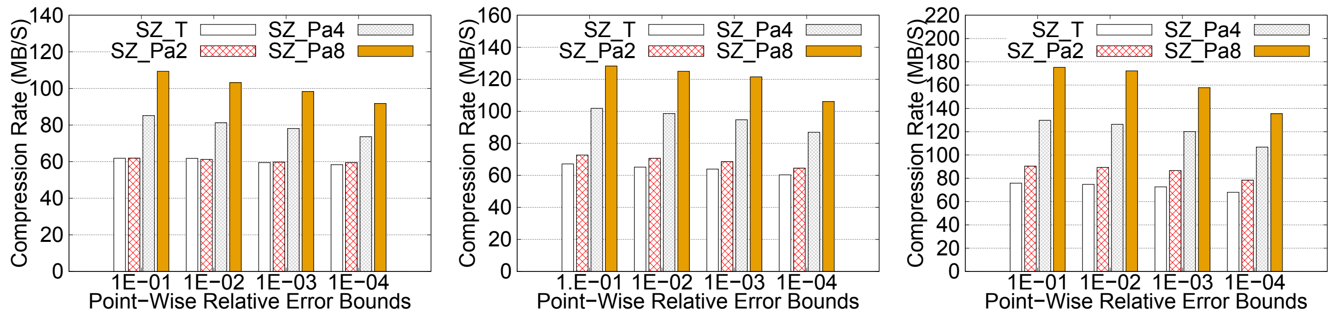
(a) Paralel SZ_Pa w/ 2 threads (b) Paralel SZ_Pa w/ 4 threads (c) Paralel SZ_Pa w/ 8 threads

Fig. 6. Time cost of running Parallel SZ as a function of the *checkWidth* on 6 fields in NYX dataset.



(a) Oringinal SZ (SZ_T) (b) Paralel SZ_Pa w/ 2 threads (c) Paralel SZ_Pa w/ 4 threads (d) Paralel SZ_Pa w/ 8 threads

Fig. 7. Break-down of the compression time for Parallel SZ (using relative error bound on NYX dataset).



(a) CESM Dataset (b) Hurricane Dataset (c) NYX Dataset

Fig. 8. Compression rate of Parallel SZ on given point relative error bound on three datasets.

large difference in progress between threads, which also can lead to lock waiting. And the processing on first several *Task*s and the last several *Task*s could not make full use of the advantages of our parallelism design, and the time cost of this situation can not be ignored.

Considering above factors (and also the results shown in Figure 5) and balancing the effect on two sides, we can configure *checkWidth* = 384, which also have some empirical reason, for high compression rate. And next we compare our approach (denoted by SZ_Pa#, where # means the number of threads) with original SZ (denoted by SZ_T) [17] to finish other experiments.

C. Time Cost in Each Stage

In this subsection, we compare break-down of compression time cost of SZ_Pa series and SZ_T with an 1e-1 point-wise relative error bound and show the time cost difference of every part in different method. The situations of 2 threads, 4 threads and 8 threads are listed and changes caused by threads number is demonstrated.

Figure 7 presents the time cost of logarithmic transformation and point-by-point processing stage significant decreases as the number of threads increases. The time cost of logarithmic transformation and point-by-point processing stage are the main parts of the total time cost for original SZ, but consist less than 50 percent of the total time cost for paralel SZ with 8 threads. As the number of threads increases, the time cost these two parts accounts for a less and less proportion, and the time cost of huffman encoding accounts for a bigger and bigger proportion. We could find that for the point-by-point processing stage, the time cost with 4 threads is about half of the time cost with 2 threads, and the time cost with 8 threads is about half of the time cost with 4 threads. And the same things happen to the logarithmic transformation stage. Therefore we can say that our method is able to linearly accelerate the two parallelized stages in SZ.

D. Compression Rate

In this section we compare the compression rate between the original SZ and our approach. Figure 8 shows that the final compression rate of our approaches with different threads. From Figure 7 we also can find that comparing with original SZ and Parallel SZ with 2 threads, parallel SZ with 2 threads do not have a very obvious improvement. The reasons are two fold: ① There are other compute-intensive tasks such as Huffman encoding in SZ, which is not parallelized in our current implementation. ② As it is shown in Figure 7, time cost of Huffman encoding becomes longer compared with SZ_T due to more cache misses in multi-thread environments. For SZ_Pa4 and SZ_Pa8, the acceleration is about 2.0× in most cases, because the time consumed on the two parallelized stages is significantly reduced. But when we further increase the number of threads, the bottlenecks are shifted to the other stages. Those stages have not been parallelized in our current implementation yet. Actually, parallelizing Huffman tree construction is very challenging, we are still investigating related techniques.

VI. Conclusion and Future Work

In this work, we explore the feasibility of parallelizing SZ lossy compressor on HPC datasets. Our optimization strategy originates from an important observation that the logarithmic transformation, Huffman decoding are the performance bottlenecks in SZ. Specifically, we use a *pipeline-like* method and exploit a series of strategies to parallelize SZ in its logarithmic transformation and the prediction & quantization stages, which keep the compression ratio the same as the original method. The key findings about our performance evaluation with three well-known application datasets are listed as follows:

- Compression Rate: Our parallel mechanism linearly accelerates the compression rate in involved stages (i.e., the stages of log transformation and point-by-point processing) in most cases.
- Compression ratio: Our parallel mechanism achieving the same compression ratio as the conventional SZ.

We have released our code at https://github.com/Borelset/SZ/tree/Parallelism to be shared with the HPC compressor research community. As future work, we also plan to explore the feasibility of parallelizing lossy compression schemes on other stages such as Huffman tree construction and Huffman encoding, which will further improve the speedup of our method. In addition, we also focus on combination of random access features and parallel mechanism.

References

[1] T. Lu and et al., “Canopus: A paradigm shift towards elastic extreme-scale data analytics on hpc storage,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 58–69.

[2] D. Tao, S. Di, Z. Chen, and F. Cappello, “Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization,” in *IEEE International Parallel and Distributed Processing Symposium*, 2017.

[3] W. Austin, G. Ballard, and T. G. Kolda, “Parallel tensor compression for large-scale scientific data,” in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 912–922.

[4] I. Foster and et al., “Computing just what you need: Online data analysis and reduction at extreme scales,” in *European Conference on Parallel Processing (Euro-Par’17)*, Santiago de Compostela, Spain, 2017.

[5] D. Ghoshal and L. Ramakrishnan, “Madats: Managing data on tiered storage for scientific workflows,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 41–52.

[6] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu, “Arrayudf: User-defined scientific data analysis on arrays,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017.

[7] ASCAC Subcommittee, “Top ten exascale research challenges,” 2014. [Online]. Available: https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf

[8] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, “A comprehensive study of the past, present, and future of data deduplication,” *Proceedings of the IEEE*, 2016.

[9] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, “A study on data deduplication in hpc storage systems,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[10] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948.

[11] M. Burtscher and P. Ratanaworabhan, “Fpc: A high-speed compressor for double-precision floating-point data,” *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2009.

[12] N. Sasaki and et al., “Exploration of lossy compression for application-level checkpoint/restart,” in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 914–922.

[13] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu *et al.*, “Understanding and modeling lossy compression schemes on hpc scientific data,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 348–357.

[14] J. Kunkel, A. Novikova, E. Betke, and A. Schaare, “Toward decoupling the selection of compression algorithms from quality constraints,” in *International Conference on High Performance Computing*. Springer, 2017, pp. 3–14.

[15] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, “Optimizing lossy compression rate-distortion from automatic online selection between sz and zfp,” *arXiv preprint arXiv:1806.08901*, 2018.

[16] N. J. F. N. B. A. Poppick, A. and D. Hammerling, “A statistical analysis of compressed climate model data,” in *The 4th International Workshop on Data Reduction for Big Scientific Data*, 2018.

[17] X. Liang, S. Di, D. Tao, Z. Chen, and F. Cappello, “An efficient transformation scheme for lossy data compression with point-wise relative error bound,” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 179–189.

[18] P. Lindstrom, “Fixed-rate compressed floating-point arrays,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[19] J. M. Shapiro, “Embedded image coding using zerotrees of wavelet coefficients,” *IEEE Transactions on signal processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[20] S. Wold, “Spline functions in data analysis,” *Technometrics*, vol. 16, no. 1, pp. 1–11, 1974.

[21] X. He and P. Shi, “Monotone b-spline smoothing,” *Journal of the American statistical Association*, vol. 93, no. 442, pp. 643–650, 1998.

[22] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. Samatova, “Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data,” *Euro-Par 2011 Parallel Processing*, pp. 366–379, 2011.

[23] S. Di and F. Cappello, “Fast error-bounded lossy hpc data compression with sz,” in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 730–739.

[24] ——, “Optimization of error-bounded lossy compression for hard-to-compress hpc data,” in *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[25] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, “Error-controlled lossy compression optimized for high compression ratios of scientific datasets,” 2018.

[26] X. Zou, T. Lu, W. Xia, X. Wang, W. Zhang, S. Di, D. Tao, and F. Cappello, “Accelerating relative-error bounded lossy compression for hpc datasets with precomputation-based mechanisms,” in *International Conference on Massive Storage Systems and Technology (MSST)*, 2019.

[27] Z. F. real-time compression algorithm. [Online]. Available: https://github.com/facebook/zstd

[28] J.-l. Gailly, “gzip: The data compression program,” 2016. [Online]. Available: https://www.gnu.org/software/gzip/manual/gzip.pdf

[29] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, “Hello adios: the challenges and lessons of developing leadership class i/o frameworks,” *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[30] T. E. Fornek, “Advanced photon source upgrade project preliminary design report,” 9 2017.

[31] G. Marcus, Y. Ding, P. Emma, Z. Huang, J. Qiang, T. Raubenheimer, M. Venturini, and L. Wang, “High Fidelity Start-to-end Numerical Particle Simulations and Performance Studies for LCLS-II,” in *Proceedings, 37th International Free Electron Laser Conference (FEL 2015): Daejeon, Korea, August 23-28, 2015*, 2015.

[32] S. Di, D. Tao, X. Liang, and F. Cappello, “Efficient lossy compression for scientific data based on pointwise relative error bound,” *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[33] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, “Out-of-core compression and decompression of large n-dimensional scalar fields,” in *Computer Graphics Forum*, vol. 22, no. 3. Wiley Online Library, 2003, pp. 343–348.