

A Study on Checkpoint Compression for Adjoint Computation

Kaiyuan Hou
Department of Electrical
Engineering & Computer Science
Northwestern University
Evanston, IL, USA
khl7265@eecs.northwestern.edu

Sri Hari Krishna Narayanan
Mathematics and Computer
Science Division
Argonne National Laboratory
Lemont, IL, USA
snarayan@mcs.anl.gov

Daniel Goldberg
School of GeoSciences
The University of Edinburgh
Edinburgh, United Kingdom
dngoldberg@gmail.com

Navjot Kukreja
Department of Earth Science &
Engineering
Imperial College London
London, UK
n.kukreja@imperial.ac.uk

Bogdan Nicolae
Mathematics and Computer
Science Division
Argonne National Laboratory
Lemont, IL, USA
bnicolae@anl.gov

Paul Hovland
Mathematics and Computer
Science Division
Argonne National Laboratory
Lemont, IL, USA
hovland@mcs.anl.gov

Abstract— When we want to understand the sensitivity of a simulation model with respect to an input value or to optimize an objective function, the gradient usually provides a good hint. The adjoint state method is a widely used numerical method to compute the gradient of a function. It decomposes functions into a sequence of basic operations. It performs a forward sweep to evaluate the function, followed by a backward sweep to calculate the gradient using the chain rule iteratively. One limitation of the adjoint state method is that all intermediate values from the forward sweep are needed by the backward sweep. Usually, we keep only a portion of those values, called checkpoints, in the memory because of limited space. The remaining values are either stored on the hard disk or recomputed from the nearest checkpoint whenever needed. In this work, we seek to compress the intermediate values in order to better utilize limited space in the memory and to speed the I/O when checkpointing to the hard disk.

Keywords—Automatic Differentiation, Adjoint State Method, Checkpoint, Compression

I. INTRODUCTION

Many scientific applications involve computing the gradient of cost functions or simulation models. Gradients are usually used to estimate the sensitivity of model output with respect to input [1] or to optimize an objective function. The adjoint state method is an efficient numerical method to compute the gradient that is widely employed by automatic differentiation (AD) tools [2,3,4,5]. Every function can be decomposed into a sequence of basic operations. The gradient of a function can be computed automatically by using the chain rule without manually deriving the derivative. The adjoint state method computes the gradient by applying the chain rule iteratively on the sequence of operations in reverse order. In many cases, it involves much less computation than do methods that follow the original order of the operations, such as the tangent linear model. By recording every intermediate result when computing a function, the adjoint state method can then be used to compute the gradient of that function in a more efficient way. It is a powerful tool in many

applications such as machine learning, data analysis, and scientific simulations. In this paper, we focus on adjoint computation in scientific simulation applications where each operation can be viewed as an iteration of time steps.

Although the adjoint state method provides a computation speed advantage over methods that apply the chain rule in the forward direction, it does come with a major limitation – memory usage. In order to apply the adjoint state method, every step during the computation of the function value along with its intermediate values must be retained. When the method is applied to large-scale scientific simulations, the state of each iteration must fit in the memory. When running simulation across a long time period, there can be thousands or even millions of iterations. Even if the size of each individual checkpoints is not large, the aggregated size across iterations can still be in terabytes or even petabytes. Although memory capacity has increased significantly in recent years, it is still far from enough to record the state of the entire simulation. In some cases, even the main file system does not have enough space to store the entire history of simulation.

To perform the adjoint state method with limited memory capacity, we need to trade time for space. Instead of saving the results of all iterations, we keep only a subset of them, called checkpoints. Whenever the value of a non-checkpoint iteration is needed, we recover it by restarting the simulation model from the nearest checkpoint [6]. Such an operation is called recomputation. In addition to memory, checkpoints can also be stored on other devices such as local hard disks, global parallel file system, or even tapes. These storage media can be employed together to create multilevel checkpointing in which higher-level (faster) checkpointing is used to recompute data between two lower-level checkpoints [7,8]. In many cases, the hard disk is used alongside the memory to create a two-level checkpointing strategy.

Different factors must be considered when doing checkpointing on the hard disk. Unlike doing checkpointing in memory where storing and retrieving checkpoints are virtually

instantaneous, hard disk I/O involves a significant latency and a limited bandwidth. Usually, disk-based checkpointing is bandwidth bound rather than capacity bound; in other words, the number of checkpoints we can store (called the budget) is limited not by disk or file system capacity but by the amount of data we can transmit from and to the storage device. The higher the bandwidth, the more budget we have on checkpoints. Improving I/O speed gives us a larger budget for checkpoints on the second level, which in turn allow the first-level budget to be applied to a shorter period, reducing recomputation and increasing overall performance. When disk-based checkpointing is applied alone as a single-layer strategy, reducing the I/O time leads to direct improvement of overall execution time.

An intuitive way to speed the checkpoint I/O time is to reduce the size of the checkpoint. A straightforward approach is to compress the checkpoints. However, it also introduces the overhead of compression and decompression. The compression algorithm must be efficient so that the saving from I/O time can cover its overhead, yet it also must be effective in reducing the size of the checkpoints. When a lossy compression algorithm is used, the error introduced may also become an issue.

Since I/O speed is known to be far slower than computation, the idea of applying compression on checkpoints used for resilience purposes has been thoroughly explored [9,10,11,12]. In such application, most of the checkpoints will just be discarded after execution unless there are anomalies that interrupts the execution. Unlike the checkpoints created for resilience purpose where decompression time does not matter, every checkpoint created by adjoint state method will be read back later, making decompression performance as important as compression performance. From the other aspect, the checkpoints in adjoint state method are not used to compute the simulation output but only used to compute the gradient. As a result, the presence of error is more acceptable than checkpoints used for resilience purpose, giving us more opportunity to employ lossy compression methods.

In this paper, we study the concept of compressing the checkpoint data for a disk-based checkpointing strategy used by adjoint computation. We run experiment using the MITgcm [13,14] simulation framework with OpenAD [2] for gradient computation. We used Zlib [15] for lossless compression. For lossy compression, we tried ZFP [16,17] and SZ [18,19]. We also tried to do checkpointing with reduced precision. We ran experiments on a small cluster and stored checkpoints to the local hard disk. In the experiment, we were able to reduce the checkpoint size by up to 90% and reduce the checkpointing time by 87%. When using lossy compression, the error did not exceed 1% even after tens of iterations. Overall, the experiment results suggest that compression is a promising approach to improve disk-based checkpointing performance.

II. RELATED WORK

Given a fixed checkpoint budget, the strategy to select iterations as checkpoints is called checkpoint scheduling. Scheduling policy is a hot topic in checkpointing research. Griewank and Walther introduced an algorithm called Revolve that is guaranteed to minimize recomputation on single-level

memory-based checkpointing architecture [6]. Wang et al. proposed a single-level dynamic checkpointing algorithm that does not require prior knowledge about the number of time steps [20]. Stumm and Walther proposed an optimal strategy in which a certain amount of time is needed to store and retrieve checkpoints, such as disk-based checkpointing [21]. They also proposed a near-optimal online strategy where the number of iterations is not known a priori. They showed that it took no more than two additional recomputations compared with the optimal solution [22]. Aupy et al. extended the work of [21] into an optimal strategy for two-layer checkpointing using memory and disk [7]. Schanen et al. introduced an asynchronous two-level adjoint checkpointing scheme for numerical time discretization targeted at large-scale numerical simulations; the scheme combines bandwidth-limited disk checkpointing and space-limited memory checkpointing [8]. Datta et al. presented a model to design and implement asynchronous two-level checkpointing methods with parameterizable values to fit current and future system configurations [23]. Chen et al. demonstrated an application of checkpointing to train large-scale deep neural networks in which the memory used is proportional only to the square root of network depth [24].

Data compression is a well-studied field in computer science. Many compression libraries cover different types of data. Zlib is a well-known general-purpose lossless compression library developed by Gailly and Adler [15]. It widely used by other applications and libraries including the gzip utility. ZFP [16] is a lossy compression library for floating-point data, developed at Lawrence Livermore National Laboratory, based on the algorithm described in [17]. It is optimized for 2D and 3D floating-point arrays. SZ is another lossy floating-point compression library, developed at Argonne National Laboratory, that specializes in compressing large-scale high-performance computing (HPC) datasets [18,19].

Compression has been widely used to reduce checkpoint size as well as I/O time when writing checkpoints to an I/O device for resilience purpose. Ibtesham et al. studied the viability and scalability of applying compression to checkpoints in Checkpoint/Restart protocol. They suggested that checkpoint data compression is a feasible mechanism for reducing checkpoint commit latencies as well as storage overheads [9,11]. Ibtesham et al. also suggested that checkpoint data compression can be part of a scalable checkpoint/restart solution. They discussed additional scenarios and improvements that make it more viable [10]. Sasaki et al. propose a lossy compression technique based on wavelet transformation for checkpoints. They achieved significant reduction in checkpointing time while introducing small error on NICAM climate application [12].

III. BACKGROUND

A. About adjoint state method and automatic differentiation

An iterative function $f: R^p \rightarrow R^q$ can be viewed as a composite function of its iterations.

$$f(x) := f_{n-1} \circ f_{n-2} \circ \dots \circ f_0(x)$$

The function can be evaluated by iterate through the iterations.

$$x_0 = x, \quad x_i = f_{i-1}(x_{i-1}), \quad f(x) = x_n$$

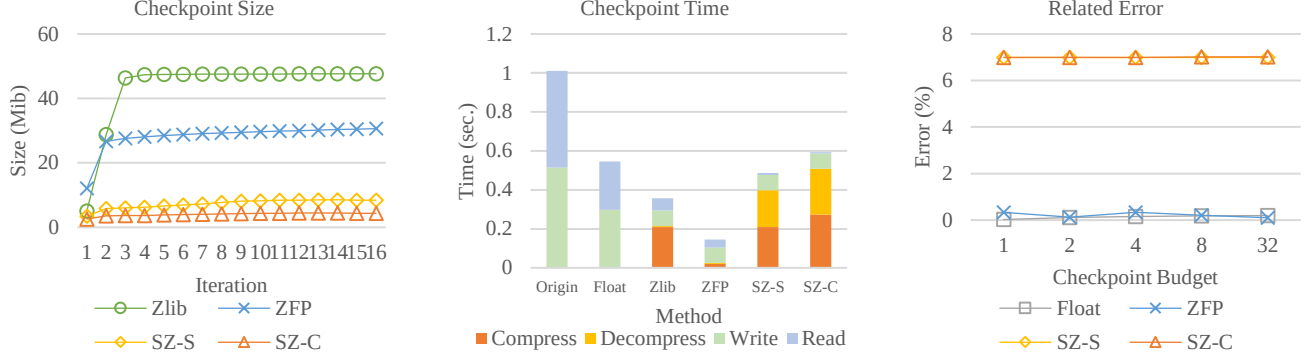


Fig. 1. Results of hs94 experiment. SZ-C and SZ-S refers to SZ library in best compression and best speed mode, respectively. Original checkpoint size is 436 MiB. Checkpointing time is averaged across 32 iterations. Errors are the maximum among the four sampled points.

Applying the chain rule, one can compute the gradient of f as follows.

$$\frac{\partial x_i}{\partial x} = f'_{i-1}(x_{i-1}) \frac{\partial x_{i-1}}{\partial x} = \bar{x}_{i-1} \frac{\partial x_{i-1}}{\partial x} \text{ where } \bar{x}_i = \frac{\partial x_i}{\partial x}$$

$$f'(x) = \frac{\partial x_n}{\partial x} = f'_{n-1}(x_{n-1}) f'_{n-2}(x_{n-2}) \dots f'_0(x_0)$$

$$= \bar{x}_{n-1} \bar{x}_{n-2} \dots \bar{x}_0$$

The multiplication (accumulation) of $\bar{x}_i$ s can be done in either the forward (starting from $\bar{x}_0$) or backward (starting from $\bar{x}_{n-1}$) direction. Usually, the number of output variables is significantly smaller than the number of input variables (i.e., $p \gg q$) in scientific simulation applications since the input variables are the initial state of the entire simulation grid whereas the output variables are just a few features we are interested in. In such cases, backward accumulation used by the adjoint state method involves significantly less computation. However, every x_i s must be available in order to compute $\bar{x}_i$ s, leading to the problem we are studying in this paper.

The adjoint state method consists of a forward sweep and an adjoint (backward) sweep. During the forward sweep, the intermediate steps to compute a function's value along with all intermediate values are recorded. The backward sweep follows recorded steps in reverse order, applying the chain rule recursively along the way to calculate the gradient of the function at the same point.

B. About MITgcm

MITgcm is an atmospheric and oceanic simulation framework written mainly in Fortran. It supports gradient computation using either the tangent linear or adjoint state method for sensitivity study or parameter optimization in some of the applications. OpenAD is used for adjoint computation. MITgcm employs a single-level disk-based checkpointing that uses a revolving scheduling strategy where the checkpoint budget is configurable. It uses native Fortran I/O to read and write checkpoints. A file is created per process per checkpoint. During each checkpoint, every variable in the simulation model is written to the file one after another in its native representation. In order to retrieve the checkpoints, data is read to the variables following the same order.

IV. IMPLEMENTATION

To perform data reduction, we reimplemented the checkpointing routine of MITgcm in C. We performed compression on each variable before writing it to the file. We decompressed the data read from the file before returning the results to the simulation routines. We studied three types of data reduction method: reduced precision, lossless compression, and lossy compression.

In the first study, we reduced the precision of floating-point values among the checkpoint data. This approach can be viewed as a simple lossy compression method. MITgcm by default stores all floating-point variables in double precision. We modified it to store single-precision checkpoints by performing type casting between the variable and the I/O buffer. This method applies only to floating-point data. We leave variables of other data types unchanged.

To study the effect of lossless compression, we used the compression library Zlib [15]. We deflate variables before writing them to the file. When we needed to restore the checkpoints, we inflated the data before putting them into variables. The size of compressed data was appended to the front of the compressed stream. Since small variables, such as scalars, are likely to be expanded instead of compressed by Zlib, we set a threshold of 1 KiB on variable size to prevent such an issue.

For lossy compression, we used the ZFP library [16] developed at Lawrence Livermore as well as the SZ library developed at Argonne [19]. ZFP takes a structure called the field from the caller that contains the dimensionality and shape of the array to decide the best compression strategy. It accepts fields only up to three dimensions. For higher-dimensional variables, we merged its higher dimensions into the third (third fastest changing) dimension. We used ZFP in fixed-accuracy mode in which the error is bounded to within 10^{-3} . Similar to the lossless case, we compressed only those floating-point variables with a size larger than 1 KiB. SZ also use the dimensionality and shape of the array to help improve the compression results. It supports arrays with up to four dimensions. We performed a transformation similar to that used in the ZFP case for arrays with higher dimensionality. We set the error bound to 10^{-3} in absolute error to match the setting used in ZFP. SZ supports two modes: best compression mode for better compression ratio

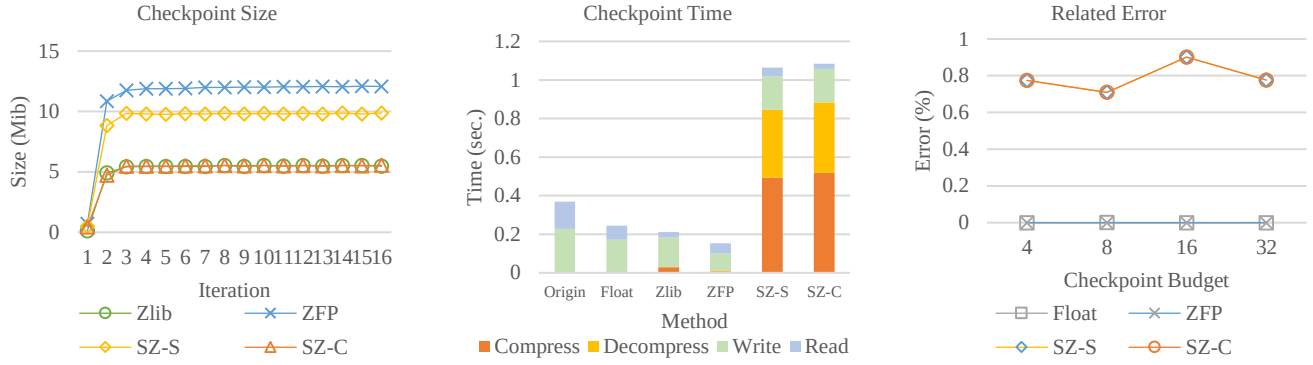


Fig. 2. Results of halfpipe experiment. SZ-C and SZ-S refers to SZ library in best compression and best speed mode, respectively. Original checkpoint size is 31 MiB. Checkpointing time is averaged across 32 iterations. Errors are the maximum among the four sampled points.

(original size / compressed size) and best speed mode for faster compression/decompression time.

V. EXPERIMENT

We ran the experiment on a small cluster connected by Intel Omni-Path. Each node is equipped with two Intel Xeon Platinum 8180M processors (56 cores in total) paired with 384 GiB of memory. Each node has access to a GPFS, and two local SSDs. We use GPFS in the experiment.

We evaluated the concept of apply compression on the hs94.1x64x5 (hs94) and halfpipe stream-ice (halfpipe) example programs. Since checkpointing in MITgcm is done in a file-per-process basis, evaluating on a single node should be sufficient to indicate the compression ratio and its impact on performance. However, to measure I/O time more accurately, we need to scale up the simulation to generate larger checkpoint files. More nodes are then added in order to speed the computation. We used direct and synchronized I/O to eliminate the interference of caches. We aggregated checkpointing data in the memory so that there is only one I/O request per checkpoint. We used the latest release of OpenAD (2014-03-15).

For the hs94 experiment, we used 256 processes on 4 nodes. We set tile size per process to 256 x 1 and the number of processes along each dimension to 256 x 1. The resulting grid is 65,536 cells evenly distributed across 180 degrees in the spherical polar grid. The corresponding checkpoint size per iteration is about 437 MiB.

In the halfpipe experiment, we adapted the same scale as in [25]. We fixed the global grid to 160 x 80 with 500 m spacing. Because of the smaller problem size, we used only 64 processes on a single node in this experiment. There are 8 processes on both dimensions, each of them holding a 20 x 10 tile. The resulting data size per checkpoint is around 31 MiB. We used the same input files as in [25].

A. Performance Study

To study the effectiveness of compression on increasing checkpoint performance, we measured the time spent on writing checkpointing files as well as time spent on doing compression. We also measured the checkpoint size after each iteration.

In the hs94 experiment, lossless compression achieves a 90% compression ratio and 65% reduction in checkpointing time. Performing checkpointing in reduced precision also results in 46% reduction in checkpointing time with negligible computational overhead. ZFP achieves a 94% compression ratio and 86% reduction in checkpointing time. SZ achieves a 98% and 99% compression ratio in best speed and best compression mode, respectively. Because of its longer computational time, however, it reduces the checkpointing time only by 51% and 41%. The best compression mode does not improve performance because the reduction in size is not large enough to compensate additional time on computation. We believe it can outperform other reduction methods if the checkpoint size is several magnitudes larger. Overall, ZFP performs the best among all methods tested.

Although the time spent on performing compression may exceed the time writing our compressed data, it is still a fraction of the time we saved by reducing data size. We can see in Table 1 that applying compression significantly reduces the overall execution time of the simulation. Since even lossless compression can achieve a decent compression ratio, we believe that checkpointing data in the hs94 example program is loosely packed or contains many redundancies. Also, we can see that the compression ratio during the first few iterations is significantly higher than in later iterations. In the first iteration, it is nearly 99% regardless of the method used. This result suggests that the initial state of the model may have low variance.

In the halfpipe experiment, lossless compression achieves an 84% compression ratio and 43% reduction in checkpointing time. Performing checkpointing in reduced precision results in a 34% reduction in checkpointing time. ZFP achieves a 65% compression ratio and 59% reduction in checkpointing time. SZ achieves a 71% and 84% compression ratio in best speed and best compression mode, respectively. SZ fails to improve overall performance because it has a Huffman coding step which has a constant cost that slow it down on small sized data. Contrary to intuition, lossless compression achieves the best compression ratio. We think the reason is that the tile size is too small (only 20 x 10) for lossy compression to utilize information from adjacent cells or to do meaningful block transformation. In terms of overall performance, ZFP again performs the best because of its faster computational time. Unlike in hs94, the end to end execution time does not reduce in halfpipe. We believe it

TABLE I. END TO END RUN TIME

Program	Origin	Float	Zlib	ZFP	SZ-S	SZ-C
hs94	104.2	78.8	59.4	54.7	71.9	72.0
halfpipe	807.2	831.4	804.7	832.5	897.2	894.0

SZ-C and SZ-S refers to SZ library in best compression and best speed mode respectively. Time is shown in seconds.

is caused by the computation-intensive characteristic of the halfpipe example. Only a fraction time is spent on checkpointing compared to doing computation. In such case, the extremely small share of checkpointing time in the entire simulation makes compression non-profitable. The small I/O time improvement may even not be measurable due to measuring noise. In addition, running compression algorithm can also compete cache resource with the computation, leading to slower overall execution time.

In all of our experiments, compression significantly reduces the checkpoint size. Even when the improvement on performance is too small to be seen, it still reduces the load on shared I/O resource for other applications in need. With neglectable computation overhead, we can significantly speed up the checkpointing procedure while saving a large amount of space and bandwidth. Our results suggest that compression is an intuitive but effective way to improve I/O-based checkpointing performance or resource utilization for adjoint computation.

B. Lossy Compression Error

A major concern about lossy compression is the error introduced. Even if we can bound the error on the checkpoint itself, the error can be magnified when propagating across the model via recomputation. To utilize lossy compression for better compression ratio, we want to make sure that the error is within a reasonable bound. MITgcm comes with a gradient check module that reports the gradient of four points sampled from the simulation grid. We compared the sampled gradient computed with compressed checkpoints to that computed with the uncompressed one. We measured the relative error after 32 iterations. By controlling the number of checkpoint budgets, we can study how fast the error propagate across the iterations via recomputation.

In the hs94 experiment, checkpointing with reduced precision introduces only negligible error. ZFP also manages to keep the error lower than 1% even when given only one checkpoint budget. SZ, on the other hand, introduces more error. Both best speed mode and best compression mode result in around 7% deviation from the uncompressed version on the gradient. Considering its extremely high compression ratio, such error is understandable. In this experiment, the checkpointing budget, or the number of recomputations, does not affect the error significantly. This result suggests that errors are not magnified when they propagate across iterations.

Since the halfpipe experiment is computationally intensive, performing many recomputations will be too slow. As a result, we started from a larger amount of checkpoint budgets in this experiment. In the halfpipe_stream-ice experiment, all the methods tested are able to keep the error lower than 1%. Using ZFP or checkpointing in reduced precision introduces almost no error in the result. SZ again brings the most error because of its high compression ratio.

Our experiment results suggest that the error will usually not do much harm when applying lossy compression on checkpoints. For most applications, such as sensitivity study and parameter tuning, we believe that such an error rate is acceptable. Unless precision is absolutely needed, lossy compression can be used to further increase the checkpointing performance.

VI. CONCLUSION

In this work, we studied the concept of applying data compression to reduce checkpoint size. We implemented compressed checkpointing in the MITgcm framework and evaluated the performance using different compression methods. We also investigated the error propagation caused by lossy compression. Our results show that compression can effectively reduce checkpoint size and increase overall performance when doing checkpointing to a slow device. We believe it is a good answer to the bandwidth limitation of I/O-based checkpointing architectures.

VII. FUTURE WORK

We briefly describe two possible research direction related to applying compression in checkpointing.

1) *Extension to memory-based CP.* Although we focus on I/O-based checkpointing in this paper, the concept can also be applied to memory-based checkpointing. Since memory access can be considered as instantaneous, compression cannot reduce checkpointing time; instead, it reduces space usage. Less space usage means a larger checkpoint budget and less recomputation. Unlike the case of I/O-based checkpointing where I/O time is linearly related to checkpoint size, applying compression on memory-based checkpointing is more complicated. The effect of checkpoint compression on overall performance depends on the checkpoint scheduling strategy and the complexity of the model. As a result, the effectiveness needs to be studied in a case-by-case fashion.

2) *Scheduling algorithm.* Many checkpoint scheduling strategies are being proposed that optimize performance on many scenarios. Many of these strategies assume that checkpoint size remains the same across all iterations; however, such an assumption may not hold true when compression is applied. It is not the case after checkpoints are compressed. In addition, many strategies consider that memory-based checkpointing does not take time to complete. This assumption no longer holds when we need time to do compression and decompression. In order to achieve the best performance using compression, a tailored scheduling strategy is needed.

ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCE

- [1] R. M. Errico, "What is an adjoint model?," Bulletin of the American Meteorological Society, vol. 78, no. 11, pp. 2577-2592, 1997.
- [2] J. Utke, et al., "OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes.," ACM Transactions on Mathematical Software (TOMS), vol. 34, no. 4, p. 18, 1008.
- [3] D. Maclaurin, D. Duvenaud and R. P. Adams, "Autograd: Effortless gradients in numpy.," in ICML 2015 AutoML Workshop, 2015.
- [4] A. Paszke, et al. , Automatic differentiation in pytorch., 2017.
- [5] A. Griewank and A. Walther, Evaluating derivatives: principles and techniques of algorithmic differentiation., vol. 105, Siam, 2008.
- [6] A. Griewank and A. Walther, "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation.," ACM Transactions on Mathematical Software (TOMS), vol. 25, no. 1, pp. 19-45, 2000.
- [7] G. Aupy, J. Herrmann, P. Hovland and Y. Robert, "Optimal multistage algorithm for adjoint computation.," SIAM Journal on Scientific Computing, vol. 18, no. 3, pp. C232-C255, 2016.
- [8] M. Schanen, M. Oana , Z. Hong and A. Mihai , "Asynchronous Two-level Checkpointing Scheme for Large-scale Adjoints in the Spectral-element Solver Nek5000," in ICCS, 2016.
- [9] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance.," in Parallel Processing (ICPP), 2012 41st International Conference on, 2012.
- [10] D. Ibtesham, D. Arnold, K. B. Ferreira and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance.," in European Conference on Parallel Processing, Berlin, Heidelberg, 2011.
- [11] D. Ibtesham, K. B. Ferreira and D. Arnold, "A checkpoint compression study for high-performance computing systems.," The International Journal of High Performance Computing Applications, vol. 29, no. 4, pp. 387-402, 2015.
- [12] N. Sasaki, K. Sato, T. Endo and S. Matsuoka, "Exploration of lossy compression for application-level checkpoint/restart.," in Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, 2015.
- [13] MITgcm. [Online]. Available: <http://mitgcm.org/>.
- [14] J. Marshall, A. Adcroft, C. Hill, L. Perelman and C. Heisey, "A finite - volume, incompressible Navier Stokes model for studies of the ocean on parallel computers.," Journal of Geophysical Research: Oceans, vol. 102, no. C3, pp. 5753-5766, 1997.
- [15] J.-l. Gailly and M. Adler, Zlib compression library., 2004.
- [16] "ZFP," [Online]. Available: <https://github.com/LLNL/zfp>.
- [17] P. Lindstrom, "Fixed-rate compressed floating-point arrays.," IEEE transactions on visualization and computer graphics, vol. 20, no. 4, pp. 2674-2683, 2014.
- [18] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ.," in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)., 2016.
- [19] 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 Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International., 2017.
- [20] Q. Wang, P. Moin and G. Iaccarino, "Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation.," SIAM Journal on Scientific Computing, vol. 31, no. 4, pp. 2549-2567, 2009.
- [21] P. Stumm and A. Walther, "Multistage approaches for optimal offline checkpointing.," SIAM Journal on Scientific Computing, vol. 13, no. 3, pp. 1946-1967, 2009.
- [22] P. Stumm and A. Walther, "New algorithms for optimal online checkpointing.," SIAM Journal on Scientific Computing, vol. 32, no. 2, pp. 836-854, 2010.
- [23] D. Datta, D. Appelhans, C. Evangelinos and K. Jordan, "An Asynchronous Two-Level Checkpointing Method to Solve Adjoint Problems on Hierarchical Memory Spaces," Computing in Science & Engineering, vol. 20, no. 4, 2018.
- [24] T. Chen, B. Xu, C. Zhang and C. Guestrin, "Training deep nets with sublinear memory cost.," arXiv preprint arXiv, p. 1604.06174, 2016.
- [25] D. N. Goldberg, S. H. K. Narayanan, L. Hascoet and J. Utke, "An optimized treatment for algorithmic differentiation of an important glaciological fixed-point problem.," Geoscientific Model Development, vol. 9, no. 5, pp. 1892-1904, 2016.
- [26] [Online]. Available: <https://www.alcf.anl.gov/user-guides/xc40-file-systems>.

Government License

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

<http://energy.gov/downloads/doe-public-access-plan>.