Concealing Compression-accelerated I/O for HPC Applications through In Situ Task Scheduling

Sian Jin
Temple University
Philadelphia, PA, USA
sian.jin@temple.edu

Sheng Di
Argonne National Laboratory
Lemont, IL, USA
sdi1@anl.gov

Frédéric Vivien
Inria & LIP, ENS Lyon
Lyon, France
frederic.vivien@inria.fr

Daoce Wang
Indiana University
Bloomington, IN, USA
daocwang@iu.edu

Yves Robert
Inria & LIP, ENS Lyon
Lyon, France
yves.robert@inria.fr

Dingwen Tao*
Indiana University
Bloomington, IN, USA
ditao@iu.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@mcs.anl.gov

Abstract

Lossy compression and asynchronous I/O are two of the most effective solutions for reducing storage overhead and enhancing I/O performance in large-scale high-performance computing (HPC) applications. However, current approaches have limitations that prevent them from fully leveraging lossy compression, and they may also result in task collisions, which restrict the overall performance of HPC applications. To address these issues, we propose an optimization approach for the task scheduling problem that encompasses computation, compression, and I/O. Our algorithm adaptively selects the optimal compression and I/O queue to minimize the performance degradation of the computation. We also introduce an intra-node I/O workload balancing mechanism that evenly distributes the workload across different processes. Additionally, we design a framework that incorporates fine-grained compression, a compressed data buffer, and a shared Huffman tree to fully benefit from our proposed task scheduling. Experimental results with up to 16 nodes and 64 GPUs from ORNL Summit, as well as real-world HPC applications, demonstrate that our solution reduces I/O overhead by up to 3.8× and 2.6× compared to non-compression and asynchronous I/O solutions, respectively.

CCS Concepts: • **Software and its engineering → Scheduling**; • **Theory of computation → Data compression**.

Keywords: HPC, parallel I/O, data compression, task scheduling, performance.

*Corresponding author: Dingwen Tao, Luddy School of Informatics, Computing, and Engineering, Indiana University.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629573

ACM Reference Format:
Sian Jin, Sheng Di, Frédéric Vivien, Daoce Wang, Yves Robert, Dingwen Tao, and Franck Cappello. 2024. Concealing Compression-accelerated I/O for HPC Applications through In Situ Task Scheduling. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3627703.3629573

1 Introduction

Large-scale scientific simulations on HPC systems play a crucial role in various scientific and engineering domains. These simulations often generate massive volumes of data that requires significant storage resources. For example, a single Nyx [3] cosmological simulation with a resolution of $4096 \times 4096 \times 4096$ cells generates up to 2.8 TB of data for a single snapshot; a total of 2.8 PB of disk storage is needed, assuming the simulation is run 5 times with 200 snapshots dumped per simulation. Managing such large amounts of data is a major challenge. It is impractical to save all the generated raw data to disk due to: (1) the limited storage capacity even for supercomputers, and (2) the time required to save all the produced data (about 1 hour) due to the limitation of I/O bandwidth: at best 1 TB/s (on ORNL Summit) [12, 57, 58, 60].

Lossy compression has been identified as one of the major data reduction techniques to address this issue. Specifically, a new generation of error-bounded lossy compression techniques, such as SZ [15, 37, 50], ZFP [40], MGARD [1], and their GPU versions [13, 34, 54], have been widely used in the scientific community [12, 15, 22, 28, 37, 40–42, 50, 52]. Compared to lossless compression, which provides up to a 2× compression ratio on scientific data [46], error-bounded lossy compressors offer a much higher compression ratio (16× to more than 200× for the applications considered in this paper) while maintaining controllable loss of accuracy.

Scientific applications typically use parallel I/O libraries such as the Hierarchical Data Format 5 (HDF5) [53] for reading and storing their data. Specifically, HDF5 is lauded for its high parallel I/O performance, data portability, and rich APIs for managing data on various systems. It is widely used at supercomputing facilities for storing, reading, and querying

scientific datasets [2, 20]. This is largely due to its specific design and performance optimizations for popular parallel file systems such as Lustre [9, 44]. Instead of relying on a general database in distributed storage, these datasets employ a specific data management approach based on the parallel file system [9, 44]. Moreover, HDF5 offers users dynamically loaded filters [25], including lossless and lossy compression [14], which enable the automatic storage and access of data in compressed formats. Thus, it allows HPC applications to handle scientific data in these formats. Parallel I/O in HDF5, combined with lossy compression filters, can significantly reduce data size, thereby improving the overall I/O performance by transmitting less data.

On the other hand, asynchronous I/O from parallel I/O libraries can help alleviate I/O bottlenecks by overlapping I/O operations with computations, enhancing the end-to-end performance of HPC applications. However, the current implementation of asynchronous I/O has certain limitations. It supports either (1) asynchronous compression and I/O [30], or (2) asynchronous I/O and computation [62]. In the former scenario, the data writing and computation are still executed sequentially, limiting potential performance gains. In the latter scenario, one would miss the opportunity to utilize lossy compression to reduce data size and enhance write performance. Moreover, asynchronous I/O typically occurs in a background thread to prevent interference with the main computational thread responsible for running the simulation. Nevertheless, it can still potentially result in performance degradation by introducing interference with other background tasks related to communication within the simulation, especially without proper scheduling [48, 55].

In this paper, we present a design that enables concurrent execution of compression, I/O, and computation. As scientific applications increasingly adopt GPUs for computation [2, 7, 19], it is crucial to ensure that the GPUs are utilized continuously with minimal interruptions. However, the main and background threads experience periods of idle time while processing computational tasks. Our goal is to harness these idle periods to perform data compression and I/O tasks. To ensure that compression and I/O tasks do not impact overall execution, we carefully schedule compression tasks during the idle times of the corresponding computation thread. It is essential to avoid any delays in existing processing tasks, as this would hinder GPU work and potentially result in a slowdown of the entire execution. To facilitate this scheduling, we propose several system designs: we introduce a framework that integrates fine-grained compression with parallel I/O libraries for scientific applications. Our framework includes three key components: fine-grained compression, a compressed data buffer, and a shared tree for Huffman coding. The fine-grained compression algorithm allows for independent compression of data blocks, thereby enhancing task scheduling efficiency. The compressed data buffer enables the overlapping of compression and I/O tasks, while the shared Huffman tree minimizes the overhead associated with building the Huffman tree for compressing small data blocks. To the best of our knowledge, unlike previous works that have only addressed the overlap between computation and I/O operations through asynchronous I/O [30, 62], this is the *first* attempt to simultaneously conceal both the compression and I/O operations from computation, thereby significantly improving the performance of HPC applications. While this work primarily focuses on HPC systems and applications, the methodology of overlapping compression, I/O, and computation can be extended to other systems, such as Cloud computing and datacenter systems. By predicting/assessing the compression performance and I/O bandwidth on these platforms, our solution could be relevant to a wider range of data-intensive, iterative applications.

The main contributions of this paper include:

- We propose to overlap both compression and I/O with computation, including task scheduling algorithms designed to efficiently integrate compression and I/O into computation with minimal interference to the system.
- We propose workflows that include three components aimed at improving end-to-end performance and task scheduling efficiency: fine-grained compression, a compressed data buffer, and a shared tree for Huffman coding.
- We discuss the implementation of our proposed framework in both simulation and real-world scientific applications to provide detailed insights.
- Evaluation of our solution on two real-world HPC applications, utilizing up to 16 nodes and 64 GPUs from the Summit supercomputer, demonstrates a reduction in I/O overhead by up to 3.8× over the baseline solution without asynchronous I/O or compression, and 2.6× over the previous asynchronous I/O only solution.

The remainder of this paper is organized as follows. In Section 2, we introduce the research background. In Section 3, we formulate the task scheduling problem considering compression, I/O, and computation, and present the solution algorithms. In Section 4, we present our proposed framework that integrates lossy compression with parallel I/O libraries. In Section 5, we present our evaluation results. In Section 6, we conclude the paper and discuss future work.

2 Background

In this section, we introduce background knowledge about parallel I/O libraries, such as HDF5, error-bounded lossy compression, and their use in scientific applications.

2.1 Parallel I/O Libraries for HPC Applications

Scientific applications generate and analyze massive amounts of data. These applications critically require the ability to efficiently access and manage this data on HPC systems. Given the complex storage hierarchy, including node-local persistent memory, burst buffers, and disk-based storage,

parallel I/O becomes the key technology that enables efficient data movement between compute nodes and storage. For instance, HDF5 [53], netCDF [36], and the Adaptable IO System (ADIOS) [21] are among the most widely used high-performance I/O libraries for HPC applications. However, these I/O libraries often struggle to handle extremely large files (e.g., with aggregate scale of petabytes and beyond) due to the inevitably limited I/O bandwidth. Consequently, compression techniques are frequently adopted to reduce the data size [47]. For example, H5Z-SZ [14] is a data filter that integrates SZ compression into HDF5.

Given HDF5's wide acceptance in the scientific community as a system supporting parallel I/O, we primarily focus our performance evaluation on HDF5, without loss of generality. Moreover, to improve performance and productivity, a recent release of HDF5 [9] implements a Virtual Object Layer (VOL), which can redirect I/O operations to the VOL connector and enable asynchronous I/O [49]. This feature allows an application to overlap I/O with other operations, such as compression. Therefore, we can leverage this capability to deeply integrate and overlap predictive lossy compression with parallel write operations, thereby improving parallel write performance. Furthermore, we focus on parallel writing to a large shared file due to three main factors: (1) It is a common usage of HDF5 because it reduces the workload for scientists in managing multiple files for storage, post-hoc analysis, and visualization. (2) It minimizes the performance overhead of opening/closing multiple files and the storage overhead of metadata for numerous small files. (3) Partial processes (e.g., up to 4,096 processes in [10]) of an HPC application with subfiling still write to a shared file.

2.2 Error-Bounded Lossy Compression

Compression is a widely utilized technique in various systems and frameworks for reducing data sizes and enhancing performance [17, 26, 31, 59]. Compared to lossless compression, lossy compression can compress data with extremely high compression ratios by losing non-critical information in the reconstructed data. The two most important metric types to evaluate the performance of lossy compression are: (1) compression ratio, i.e., the ratio between original data size and compressed data size, or bit-rate, i.e., the number of bits on average for each data point (e.g., 32/64 for single/double-precision floating-point data) before compression; and (2) data distortion metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) to measure the reconstructed data quality compared to the original data. In recent years, a new generation of high-accuracy lossy compressors for scientific data has been proposed and developed for scientific floating-point data, such as SZ [15, 37, 50] and ZFP [40]. These lossy compressors provide parameters that allow users to control the loss of information due to lossy compression precisely. Unlike traditional lossy compressors such as JPEG [56] which are designed for images (in integers), SZ and ZFP are designed to compress floating-point data and can provide a strict error-controlling scheme based on user's requirements. Generally, lossy compressors provide multiple compression modes, such as the error-bounding mode. The error-bounding mode requires users to set an error type, such as point-wise absolute error bound, and a bound value (e.g., 10^{-3}). The compressor ensures that differences between original and reconstructed data do not exceed the error bound.

Specifically, SZ is a prediction-based, error-bounded lossy compressor designed for scientific data. It involves three main steps: (1) Each data point's value is predicted based on its neighboring points, using an adaptive, best-fit prediction method. (2) The difference between the actual value and the predicted value is quantized, based on the user-set compression mode and error bound. (3) Customized Huffman coding and additional lossless compression are applied to achieve a high compression ratio.

Prior work has studied the impact of lossy compression on the quality of reconstructed data and post-hoc analysis, providing guidelines on how to set compression configurations for specific applications [16, 28, 29, 37, 38, 51]. For instance, a comprehensive framework was developed to conduct a systematic analysis of compression configurations with a given dataset, providing the best-fit solution that satisfies post-hoc analysis requirements [22]. Moreover, Jin et al. [27] proposed a theoretical ratio-quality model to efficiently maximize the compression ratio given the quality constraints of post-hoc analysis. Note that, as in previous work [63] on improving communication efficiency via lossy compression, this study assumes that the compression configuration is set up by users based on their data quality requirements. Therefore, the aforementioned compression configuration methods are orthogonal to our solution.

2.3 I/O-intensive Scientific Applications

In this paper, without loss of generality, we primarily focus on two I/O-intensive scientific applications—Nyx [2] and WarpX [19], which have been used in numerous previous I/O studies [4–6, 11, 24, 33]. These scientific simulations generally follow an iterative process, with variable durations. Notably, neighboring iterations frequently exhibit high similarity in their durations, enabling us to predict the resource utilization pattern of the current simulation iteration based on past iterations. We introduce them in detail as follows.

Nyx is an adaptive mesh, hydrodynamics code designed to model astrophysical reacting flows on HPC systems [2, 3]. This code models dark matter as discrete particles moving under the influence of gravity. The fluid in gas-dynamics is modeled using a finite-volume methodology on an adaptive set of 3-D Eulerian grids/meshes. The mesh structure is used to evolve both the fluid quantities and the particles via a particle-mesh method. For parallelization, Nyx uses MPI for the long-range force calculation and architecture-specific programming language for the short-range force algorithms,

such as OpenMP and CUDA. Nyx uses multiple 3-D arrays to represent field information in grid structure. According to prior studies [2, 23, 45], it can run up to tens of thousands of GPUs on leadership supercomputers such as Summit [18].

WarpX is a highly-parallel and highly-optimized code that utilizes AMReX [61], runs on GPUs and multi-core CPUs, and features load-balancing capabilities. WarpX can scale up to the world's largest supercomputer and was the recipient of the 2022 ACM Gordon Bell Prize [43].

3 Task Scheduling with Lossy Compression

Unlike previous works that can only overlap two of the three components [30, 62] among compression, I/O, and computation, our solution leverages task scheduling techniques and runtime system designs to efficiently conceal both the compression and I/O operations from computation in HPC applications. In this section, we present a comprehensive approach to task scheduling that considers all compression, I/O, and computation. First, we define the problem and highlight the challenges encountered when optimizing task scheduling under the constraints that arise with data compression. Next, we recall the known complexity results. Then we introduce our proposed scheduling algorithms to determine the most efficient execution sequence for compression and I/O tasks, while maintaining high computational performance. Lastly, we discuss a technique to balance the I/O workload between processes, thereby enhancing overall system efficiency.

3.1 Problem Formulation

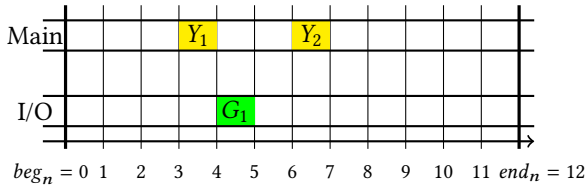
Data compression is one of the most effective approaches for reducing data size and mitigating I/O bottlenecks in HPC applications. Likewise, the use of asynchronous I/O from parallel I/O libraries can significantly enhance end-to-end performance by overlapping I/O operations with computations. Regrettably, these two techniques are not yet well-integrated, meaning we can only benefit from one method at a time. Although compression can be considered part of the computation, and its execution can be overlapped with I/O, no previous studies have proposed a comprehensive task scheduling approach that incorporates compression, I/O, and computation together. Therefore, before designing our task scheduling algorithm and proposing our framework design, we first formulate the problem that asynchronously executes data compression and I/O with respect to computation.

In this work, we target parallel scientific applications on HPC systems. Inter-process optimizations will be addressed in Section 3.4. Moreover, we focus on iterative HPC applications. Typically, either an initial solution is refined iteratively or a dynamic simulation is performed where each iteration computes the state of the system at a different date.

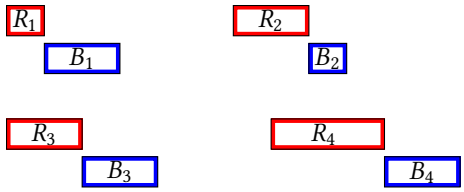
Let us consider iteration n in such a process. Let $I_n = [beg_n, end_n]$ denote the time interval during which this iteration is executed. The length of this iteration is then $T_n = end_n - beg_n$. During each iteration, the HPC application is executing a series of tasks on both CPU and GPU. Usually, each node operates multiple processes, with each process assigned to a dedicated GPU along with a subset of CPU cores. (In well-optimized HPC applications, these processes can utilize the majority of CPU cores). Within each process, there are multiple threads: one associated with the GPU (for computation) and others with the CPU cores (for computation/compression/IO). Parallel I/O operations are collaboratively executed by the background thread in each process. It is also common for the application to perform a number of I/O operations before or after some computation. We ignore the GPU and concentrate on the computing, compression, and I/O tasks performed on the main thread, and on the background thread by the CPU. We assume that for each iteration, there are k computing tasks, $Y_{n,1}, \ldots, Y_{n,k}$, executed by the main thread. Interference with these computing tasks would directly delay the execution of the application. The i-th task, $Y_{n,i}$, begins its execution at time $beg_n + a_{n,i}$ and ends at time $beg_n + b_{n,i}$, for $1 \leq i \leq k$.

Periodically, either at each iteration, or every l iterations for some fixed value l, the application generates a large amount of data that needs to be written to the storage system. To reduce I/O overhead and storage footprint, we apply lossy compression to these data. A process is responsible for managing a certain number of data fields, and each of these fields undergoes our proposed fine-grained compression. This decomposition divides the compression into a total of m independent compression tasks for the given process, as detailed in Section 4. All iterations that execute compression tasks comprise m compression tasks that need to be executed on the main-thread, namely tasks $R_{n,1}, \ldots, R_{n,m}$ for iteration n. Compression task $R_{n,j}$ has a duration $c_{n,j}$, for $1 \leq j \leq m$. Task scheduling must ensure that the m compression tasks are inserted within the time period $[beg_n, end_n]$ without interfering with any of the computing tasks $Y_{n,1}, \ldots, Y_{n,k}$. In particular, this means that the execution of each task $Y_{n,i}$ should start and end at the same times whether we perform lossy compression (execution of tasks $R_{n,1}, \ldots, R_{n,m}$) or not. Otherwise, the original computing tasks would be interrupted by compression, which would delay the execution of the application. Moreover, any compression task $R_{n,j}$ that was started before a task Y_i must be completed before Y_i starts (in other words, a compression task cannot be preempted). The scheduling of the compression tasks is therefore quite constrained. However, we have absolute freedom on the order in which the m compression tasks are executed. The scheduler will therefore have to choose an execution order of compression tasks that optimizes the overall performance.

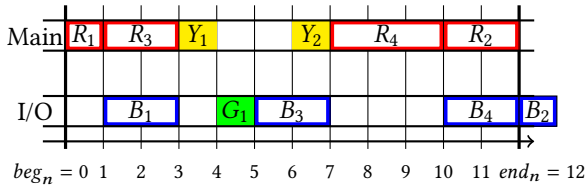
Each compression task $R_{n,j}$ is followed by a corresponding I/O task $B_{n,j}$ of length $c'_{n,j}$ that writes the compressed data to the storage system. We call a *job* the pair formed by a compression task $R_{n.j}$ and the corresponding I/O task $B_{n,j}$. We



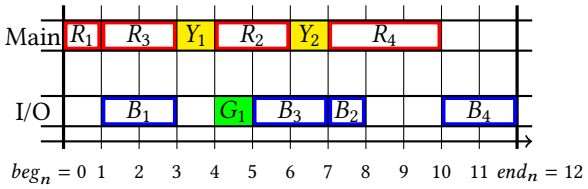
(a) The set of unavailability intervals. On the main-thread there are two unavailability intervals, from $a_1 = 3$ to $b_1 = 4$ and from $a_2 = 6$ to $b_2 = 7$; and on the background thread one from $a'_1 = 4$ to $b'_1 = 5$.



(b) The 4 jobs : $c_1 = 1$, $c'_1 = 2$, $c_2 = 2$, $c'_2 = 1$, $c_3 = 2$, $c'_3 = 2$, $c_4 = 3$, and $c'_4 = 2$.



(c) Extended Johnson's algorithm. Jobs 1 and 3 belong to $\mathcal{M}_1$ ($c_i < c'_i$) while jobs 2 and 4 belong to $\mathcal{M}_2$. The algorithm schedules the jobs in the order 1, 3, 4, 2.



(d) Extended Johnson's algorithm with backfilling. Job 2 is scheduled before job 4 (both for the compression task and the actual I/O) because this is possible without modifying the start times of tasks R_4 and B_4.

Figure 1. An example of the task scheduling problem: (a) the set of unavailability intervals; (b) the tasks to schedule; (c) the schedule built by the ExtJohnson algorithm on this instance; (d) the solution built by ExtJohnson+BF. Main stands for main-thread.

assume that the I/O operations and communications of the application all occur on the background thread. For iteration n, these m I/O tasks must then be scheduled within the time interval $[beg_n, end_n]$ on a background thread. By construction, these tasks are separated from, and cannot interfere with neither the computing tasks (the $Y_{n,i}$'s) nor the compression tasks (the $R_{n,j}$'s). However, there are o core tasks $G_{n,1}, \ldots, G_{n,o}$ associated with computing which are executed on the background thread. Note that these tasks can be either I/O operations or communications. Like the computing tasks $Y_{n,1}, \ldots, Y_{n,k}$, these tasks have known execution intervals that must be respected when scheduling the I/O tasks. Any interference or re-allocationof these tasks would directly cause delays in the simulation. Namely, task $G_{n,i}$ must be executed during the time interval $[beg_n + a'_{n,i}, end_n + b'_{n,i}]$. Similarly to the compression tasks $R_{n,1}, \ldots, R_{n,m}$, the m I/O tasks $B_{n,1}, \ldots, B_{n,m}$ must be scheduled within the interval $[beg_n, end_n]$ without interfering with any core task $G_{n,i}$. Specifically, the execution of an I/O task cannot be preempted and should not overlap with any interval $[beg_n + a'_{n,j}, beg_n + b'_{n,j}]$, for $1 \leq j \leq o$. However, I/O tasks have an additional constraint. An I/O task $B_{n,i}$ cannot start before the corresponding compression task $R_{n,i}$ has finished. This is because the I/O operation on compressed data must occur after the data has been compressed. The execution order of the I/O tasks $B_{n,1}, \ldots, B_{n,m}$ can also be freely adjusted to optimize overall performance, as long as all constraints are satisfied.

In essence, we can summarize all these constraints as follows (refer to Figure 1 for an example): On the compute thread, the compute (yellow) tasks $Y_{n,j}$ are immovable obstacles that the compression (red) tasks must avoid. Similarly, on the background thread, the core (green) tasks $G_{n,j}$ are immovable obstacles that the I/O (blue) tasks must avoid. The two threads are interdependent, as an I/O (blue) task cannot start until the corresponding compression (red) task completes. This is the only constraint on the order of compression and I/O tasks, which can otherwise be scheduled in any order compatible with these dependence constraints.

The goal of the task scheduling algorithm is to find an ordering for compression tasks $R_{n,1}, \ldots, R_{n,m}$ and for I/O tasks $B_{n,1}, \ldots, B_{n,m}$ to minimize the overall execution time of the current iteration. This execution time is defined as:

$$T_n^{overall} = \min \begin{cases} T_n \\ \max_{1 \leq i \leq m} t_{end}(R_{n,i}) - beg_n \\ \max_{1 \leq i \leq m} t_{end}(B_{n,i}) - beg_n \end{cases}$$

where $t_{end(X)}$ denotes the completion time of task X. Since an I/O task must always occur after the corresponding compression task, $t_{end}(B_{n,i})$ is always larger than $t_{end}(R_{n,i})$. This simplifies $T_n^{overall}$ to $\min\left\{T_n, \max_{1 \leq i \leq m} t_{end}(B_{n,i}) - beg_n\right\}$.

The scheduling problem is now formally defined. The optimization problem depends on many task lengths and dates, namely the values of all variables $a_{n,i}, b_{n,i}, c_{n,i}, a'_{n,i}, b'_{n,i}, c'_{n,i}$'s. The objective is to minimize the iteration length $T_n^{overall}$. Based on our observations, we assume that the execution pattern is highly similar between consecutive iterations for iterative scientific applications. Therefore, for scheduling the n-th iteration we will use the recorded characteristics of the $(n-1)$-th iteration. Namely, we will assume that, for $1 \leq i \leq k$, $a_{n,i} = a_{n-1,i}$ and $b_{n,i} = b_{n-1,i}$, that for $1 \leq i \leq o$,

$a'_{n,i} = a'_{n-1,i}$ and $b'_{n,i} = b'_{n-1,i}$, and that $T_n = T_{n-1}$. The duration $c_{n,j}$ of compression task $R_{n,j}$ can be accurately predicted based on the data to be compressed, for $1 \leq j \leq m$. Hence, this information is available before scheduling tasks and there is no need to reuse past data for the $c_{n,j}$'s.

It is important to note that slight variations in the required task lengths and dates between neighboring iterations may result in some performance degradation, as the proposed task scheduling approach takes imperfect data as its input. However, as we will demonstrate in the evaluation section, these variations do not significantly impact the effectiveness of the proposed solution.

3.2 Problem Analysis

Following scheduling theory nomenclature, the problem of scheduling the compression and I/O tasks is a flow-shop problem with two machines. The first machine corresponds to the computation thread and the second machine correspond to the background thread. Each job of the flow-shop formulation comprises first a compression task that must be scheduled on the first machine and then an I/O task that must be executed on the second machine and can start only after the completion of the first task. More precisely, our problem is exactly a flow-shop problem with two machines with *deterministic unavailability intervals* on both machines and with *non resumable* jobs. The unavailability intervals are said to be deterministic because their existence and dates are known in advance. In practice, however, the predictions related to the dates and durations of the unavailability intervals, and on the durations of compression and I/O tasks, will be unperfect. Therefore, the proposed solutions should tolerate some variability with minimal performance degradation. A job is said to be resumable [35] if its execution can continue after an unavailability interval whenever it was started but not completed before the start of this unavailability interval.

The flow-shop problem with two machines, in the absence of unavailability intervals, is optimally solved by Johnson's greedy algorithm [32]. Unavailability intervals change the complexity. The problem with non resumable jobs becomes NP-complete as soon as there is at least one unavailability interval on one machine [35]; it becomes non-approximable by any constant factor (unless P=NP) as soon as there are at least two unavailability intervals on one machine [8].

Lastly, we note that although the optimization problem is complex, it can be formulated as an Integer Linear Program (ILP). We provide the ILP formulation in the Appendix for completeness. However, it is important to mention that the ILP was unable to find a solution for any of the experiments we conducted due to the large number of variables involved in the ILP formulation.

3.3 Scheduling Algorithms

We present in this section six different scheduling algorithms for the problem. We start by proposing two extensions of Johnson's algorithm. Then we propose two algorithms based on list-scheduling. Finally, we propose two greedy algorithms which are more computationally demanding, since they explore a wider spectrum of solutions. Each algorithm produces the ordered list used to schedule the tasks, and the rule of the game (defined below) : (i) without backfilling; (ii) with backfilling; or (iii) with exhaustive insertion. The complete schedule, with the starting time of each task, is fully determined by the ordered list and the rule of the game, as detailed below.

3.3.1 Extensions of Johnson's Algorithm.

As recalled in Section 3.2, Johnson's algorithm builds an optimal solution in the absence of unavailability intervals. Johnson's algorithm works as follows. The jobs are partitioned in two sets $\mathcal{M}_1$ and $\mathcal{M}_2$ where jobs in $\mathcal{M}_1$ are exactly the jobs which have an execution time on the first machine smaller than or equal to their execution time on the second machine. In our context, this means that $\mathcal{M}_1$ contains exactly the jobs whose compression task is not longer than its I/O task. Now jobs in $\mathcal{M}_1$ are sorted by non-decreasing execution time on the first machine (i.e., by non-decreasing duration of their compression task). While jobs in $\mathcal{M}_2$ are sorted by non-increasing execution time on the second machine (i.e., by non-increasing duration of their I/O task). Then Johnson's algorithm executes first all jobs in $\mathcal{M}_1$ followed by all jobs in $\mathcal{M}_2$, starting each task as soon as possible.

We extend Johnson's algorithm to take unavailability intervals into account. The first version is straightforward: the ExtJohnson algorithm executes tasks in the same order as Johnson's algorithm would have in the absence of unavailability intervals, but executes each task as soon as possible after already scheduled tasks and while respecting the unavailability intervals. ExtJohnson is illustrated on Figure 1c where $\mathcal{M}_1 = \{(R_1, B_1), (R_3, B_3)\}$.

On Figure 1c, there is an unused interval on the first machine between times 4 and 6, during which task R_2 could have been executed. Backfilling [39] is a scheduling technique which enables to take advantage of such intervals of idleness. Under backfilling, a new task, rather than being mandatorily scheduled *after* the completion of all already scheduled tasks, can be scheduled in an idleness interval if doing so does not delay the start of any already scheduled task. If several idleness intervals can accomodate the new task, the task is scheduled in the earliest one. ExtJohnson+BF is a variant of ExtJohnson with backfilling. ExtJohnson+BF considers tasks for scheduling decisions in the same order as ExtJohnson but starts each task as soon as possible, provided that this never postpones the start of an already considered task. The behavior of ExtJohnson+BF is illustrated on Figure 1d. Note that, because tasks in $\mathcal{M}_1$ are ordered by non-decreasing compression times, the execution dates of compression tasks of $\mathcal{M}_1$ are always the same under ExtJohnson and ExtJohnson+BF.

3.3.2 List Scheduling Algorithms. In parallel scheduling, a list scheduling algorithm considers tasks one by one following a predefined order, and it schedules each task as soon as possible after the already scheduled tasks. Hence, ExtJohnson is a list scheduling algorithm following Johnson's order without unavailabity intervals. Our third algorithm GenerationListSchedule uses list-scheduling on the tasks, according to their original order when we generate them by fine-grained compression. The fourth algorithm GenerationListSchedule+BF is the counterpart of GenerationListSchedule when adding a backfilling mechanism: when scheduling a task, GenerationListSchedule+BF allows it to start earlier than some already scheduled task, as long as it does not cause any delay to scheduled tasks.

3.3.3 More Costly Greedy Algorithms. The fifth and sixth algorithms explore more execution orders. The idea is to try and insert the new task under consideration at any position in the task list that captures the current solution. Both algorithms start with the original order of the tasks when these are generated by fine-grained compression. OneListGreedy always keep the same order for compression and I/O tasks, while TwoListsGreedy enables different orderings. For OneListGreedy, assume that we have a partial ordered list of r compression tasks . For the $r+1$-th task, we try and insert it at any possible position (first, second, ..., last) in the list, which means $r+1$ attempts. Each attempt consists of greedily scheduling the compression tasks and the I/O tasks as soon as possible, while ensuring that no I/O task can start before the completion of the corresponding compression task. OneListGreedy will retain the attempt that leads to the smallest total execution time and updates the partial list accordingly. This insertion technique is more aggressive than backfilling, because it recomputes and possibly delays the starting times of previously scheduled tasks.

OneListGreedy restricts to constructing and updating the same ordered list for both compression and I/O tasks. On the contrary, TwoListsGreedy allows for different orderings of the compression tasks and the I/O tasks. Therefore it maintains two partial lists, one for each task type. When inserting the $r+1$-th tasks, we now have $(r+1)^2$ attempts and keep the best of them to update both lists. Note that each attempt can be executed in time linear in the number of tasks and unavailability intervals. Altogether, the complexity of OneListGreedy is $O(K^2)$ and that of TwoListsGreedy is $O(K^3)$, where $K = \max(m, k, o)$ is the maximum number of parameters, either tasks (m compression and m I/O tasks) or unavailability intervals (k on the compression thread and o on the background thread).

3.4 I/O Workload Balancing

The proposed scheduling algorithms lead to significant reductions in the overall execution time of simulations. However, it is important to note that the size of compressed data can vary significantly across processes, depending on the compressibility of the data assigned to each process. Some processes may have partitions with less information that can be effectively compressed, while others may have more challenging partitions that are less compressible. While compression time is not highly dependent on the compression ratio, I/O time is greatly influenced by the size of the compressed data. Consequently, some processes may experience longer I/O times compared to others, potentially becoming a bottleneck for the entire system. To address this issue, we propose an intra-node load balancing mechanism to mitigate the imbalance of the I/O workload for compressed data. We do not extend this mechanism to inter-node scenarios due to the significantly higher inter-node communication overhead.

For most HPC applications, we notice that the compression ratio of the data being dumped remains relatively stable from one iteration to another for a given partition. For instance, in a Nyx simulation, we have observed that the compression ratio differences between two consecutive data dumping operations has an average of 1.45% with a standard deviation of 0.64% from six samples. This stability is due to the fact that the simulation does not undergo significant changes in the data characteristics over a short period of time. Furthermore, our evaluation, as depicted in Figure 6, also corroborates this point by demonstrating our ability to utilize the same Huffman tree for encoding quantization codes across multiple iterations. Based on this observation, we propose utilizing the compression ratio from the previous iteration as a guide for load-balancing the I/O operations in the current iteration. By doing so, we can effectively address any imbalances and ensure a more equitable distribution of the I/O workload.

Within a given node, we determine the workload of each process by considering its total length of I/O tasks from the previous iteration. If a process has a high workload, we redistribute a proportion of its I/O tasks to processes with lower workloads. Specifically, we implement a load-balancing mechanism by assigning the first I/O task of the process with the largest I/O workload to be the last I/O task for the process with the least I/O workload. We continue this assignment until the workload of the process with the largest workload is smaller than twice the workload of the process with the smallest workload. This helps to redistribute the I/O tasks more evenly across the processes and alleviate any workload imbalances. However, we have observed that the total length of compression tasks for different processes remains relatively stable. This is because the compression throughput is not significantly affected by the compressibility of the data, and the size of the raw data is the same across all processes. Therefore, we apply the task load-balancing technique only to the I/O tasks, as it effectively addresses the workload imbalance in that aspect.

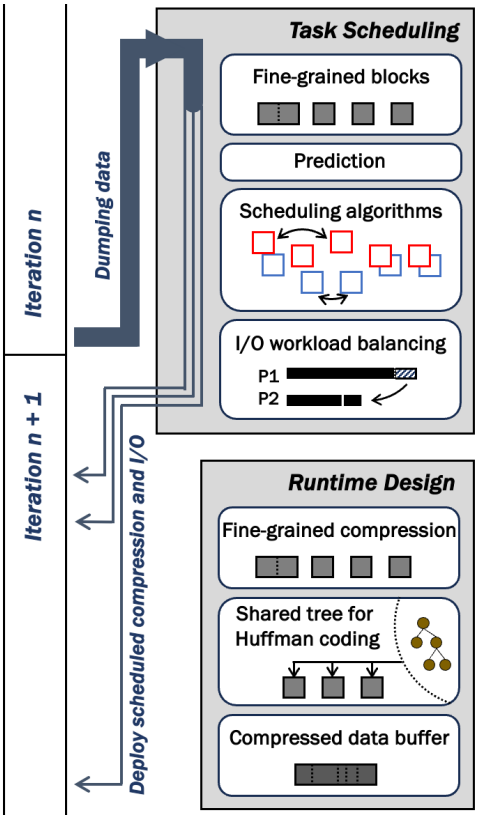


Figure 2. Overview of our proposed framework. We propose three runtime designs to facilitate task scheduling to handle the data dumping process from iteration n and execute asynchronous compression, I/O operations, and computation tasks during iteration n+1.

4 Design of Our Proposed Framework

In this section, we present our proposed framework that deeply integrates lossy compression with parallel I/O libraries, allowing for the overlap of compression and I/O with computation using our task scheduling algorithms. Figure 2 shows the overview of our proposed framework. The goal is to dump the data generated from iteration n during the next iteration $n+1$. First, we detail the runtime design that consists of three main components: fine-grained compression, compressed data buffer, and shared tree for Huffman coding. Then, we offer implementation details for HPC applications.

4.1 Fine-grained Compression

The large volume of data generated by HPC applications often consists of multiple data fields. For instance, data from a Nyx cosmological simulation may include density, temperature, and velocity information, resulting in multiple data fields. While it may seem intuitive to compress each data field separately to achieve granularity in compression tasks for our task scheduling algorithms, most scientific applications have only a few data fields (e.g., 6~12 data fields). This limits the number of compression tasks and I/O tasks, leading to low task scheduling efficiency. To address this challenge, we propose slicing each data field into smaller data blocks and independently compressing each data block. This significantly increases the number of tasks, benefiting our task scheduling problem with obstacles. Specifically, we use a block size of 8~16 MB for compressing the data, while ensuring an even division of each data field. As an example, consider a Nyx cosmology simulation with a scale of $1024 \times 1024 \times 1024$ distributed across 64 processes, resulting in 64 MB of data per process for each data field. In this case, we divide each data field into eight blocks per process using a block size of 8 MB. Fine-grained compression may introduce two potential issues: (1) a degradation in compression ratio compared to compressing the data together, and (2) a decrease in compression and I/O throughput. However, the evaluation presented in Section 5.3 demonstrates that using a minimum data block size of 8 MB results in minimal compression ratio degradation. In practice, we use offline profiling to evaluate compression and I/O performance on a given system to identify the point at which compression and I/O throughput start to deteriorate with small data block sizes. This analysis informs our choice to select the smallest available block size ($\geq$ 8 MB). To mitigate the impact on I/O throughput, we propose the use of a compressed data buffer. Moreover, to minimize overhead caused by degradation in compression throughput, we propose the use of a shared Huffman tree.

4.2 Compressed Data Buffer

Based on our observations, the size of compressed data blocks can be relatively small, with some blocks even smaller than 1 MB, depending on the achieved compression ratio. However, writing data smaller than 1 MB can significantly reduce I/O throughput. To optimize I/O efficiency, we introduce an additional compressed data buffer that allows us to initiate I/O tasks as soon as possible while maintaining high throughput. Based on our evaluation presented in Section 5.3, we set the maximum size of the compressed data buffer to 10 MB, which typically accommodates 12 compressed data blocks when the data block size is 8 MB and the average compression ratio is 10×. We have observed that further increasing the maximum compressed data buffer size does not result in significant performance improvement. The policy of writing the compressed data to the buffer and subsequently writing the buffer to the storage system is determined after the task scheduling. During the execution phase, once the execution orders for compression and I/O tasks are determined, we start placing the compressed data into the buffer when the background thread is engaged with I/O tasks or core tasks.

4.3 Huffman Coding with Shared Tree

During the prediction-based lossy compression process, one of the crucial steps is Huffman encoding of the quantization codes after prediction and quantization steps. When compressing small data blocks with high compression ratios,

building the Huffman tree can become a bottleneck for compression throughput. This is because building the Huffman tree takes nearly constant time regardless of the size of the input data, as the number of quantization codes after the predictor and quantizer of lossy compression is a fixed number in most cases. Additionally, based on our observation, data of similar types often result in similar Huffman trees. For example, the Huffman tree built from the data of one iteration is highly similar to the tree built from the data of the next iteration.

To improve compression throughput for small data blocks, we propose using shared Huffman trees across different timesteps and data blocks on the same process. Prediction-based lossy compression accommodates outliers, which allow us to modify to include values that defy coding by this shared Huffman tree. Building and using a shared Huffman tree for each process based on the data of the current iteration is impractical, as it would require synchronization of all compression tasks before any I/O taskfor compressed data can proceed. Instead, we build the shared Huffman tree based on the quantization code from the previous one or few iterations and utilize it for the data of the current iteration. This is a trade-off between compression ratio and shared tree reuse frequency. This tree is stored in memory and loaded into the compressor when compressing each data block. Based on our evaluation in Section 5.3, the shared Huffman tree can be reused for over 10 iterations without significant compression ratio degradation.

4.4 Implementation Details

We implement our solution using the HDF5 parallel I/O library [53], but the principles can be extended to other parallel I/O libraries as well. We utilize the VOL connector from HDF5 [49] to control the compression queue of the compression tasks and launch asynchronous I/O tasks in the background thread. In addition, we predict the compression ratio before the actual compression process to compute the offset for each data block using the algorithms described in [27]. We also predict the compression time and I/O time for the compressed data based on the compression throughput and I/O throughput prediction approach proposed by Jin et al. [30]. To handle the rare occurrence of data overflow caused by a lower-than-predicted compression ratio, we have implemented additional space at the end of the shared HDF5 file to store the overflowed data. This extra I/O task for the overflowed data is not predictable and cannot be scheduled in advance. Thus, we queue this extra I/O task at the end of the last I/O task for the compressed data.

All the proposed task scheduling algorithms are based on the observation that the total compression time is theoretically fixed regardless of the compression order. Our optimization focuses on the dependencies and timing of launching write operations for each compressed data to minimize timeouts compared to compression. The time complexity of the proposed algorithm is $O(n \log n)$, whereas the time complexity of our compression is $O(N)$. Considering that N (i.e., the number of values) in one data partition is significantly larger than n (i.e., the number of data fields), the optimization overhead is almost negligible compared to the actual compression and write time. On a typical HPC application run, n can range from 6 to 12, while N can range from 2 million to 128 million.

Based on our algorithm design, we can expect the optimization to bring benefits when there is a relatively stable balance between compression time and I/O time. Additionally, we note that our optimization can provide greater benefits when the size of data fields is relatively large. This is because the overall performance is dependent on the process with the longest time among all the processes, due to independent asynchronous writes.

5 Performance Evaluation

In this section, we present the evaluation results of our proposed framework for accelerating HPC applications. We first provide details about the experimental setup and the HPC applications used in our evaluation. Next, we assess the performance of our proposed task scheduling algorithms and identify the most efficient one for further evaluations. We then evaluate the performance of each individual component of our compression design. Finally, we conduct a comprehensive performance evaluation using both simulation and real-world HPC applications, comparing the results to the baseline solution without compression and the previous solution using asynchronous I/O without lossy compression.

5.1 Experimental Setup

System configuration. We rigorously implement our approach using HDF5 [9] and SZ3 [38], a modularized prediction-based lossy compressor. Our experiments are conducted on the Summit supercomputer [18] at Oak Ridge National Laboratory with 16 nodes and 64 GPUs, where each node is equipped with two IBM POWER9 processors featuring 42 physical cores and 512 GB DDR4 memory.

Compression configuration. We evaluate our approach using different scales of Nyx and WarpX applications. In all our evaluations, we utilize both GPUs and CPUs. While GPUs serve as the primary compute unit, CPUs handle compression and I/O tasks. Based on previous work [28, 29], we use absolute error bounds of $(0.2, 0.4, 1e{+}3, 2e{+}5, 2e{+}5, 2e{+}5)$ to compress the six Nyx data fields (baryon density, dark matter density, temperature, velocity x, velocity y, velocity z), respectively, to achieve an average PSNR (peak signal-to-noise ratio) of 78.6 dB, resulting in a compression ratio of approximately 16×. The problem size of the Nyx application used in our evaluation is $4096 \times 4096 \times 4096$ with three additional fields: particle_vx, particle_vy, and particle_vz. We compress these additional fields with a compression ratio of

Algorithm	Iteration Duration (s)
ExtJohnson	4.363
ExtJohnson+BF	4.058
GenerationListSchedule	4.665
GenerationListSchedule+BF	4.470
OneListGreedy	4.541
TwoListsGreedy	4.274

Table 1. Iteration duration (in seconds) achieved by different scheduling algorithms with Nyx cosmological simulation.

16x to ensure the post-hoc analysis quality. For the WarpX application, we compress the data fields with a compression ratio of 273.9×, as suggested by the application developers based on their post-hoc analysis.

5.2 Evaluation of Task Scheduling Algorithms

First, we conduct an evaluation of the six task scheduling algorithms proposed in Section 3.3, focusing on their overhead and optimized iteration time, based on the problem statement outlined in Section 3.1. To gather data for our evaluation, we sample three stages of a Nyx run, which is performed at a scale of $1024 \times 1024 \times 1024$ with 16 GPUs from 4 nodes. We collect data from the beginning of the run when the data distribution is mostly even, the middle of the run when the data is structured, and towards the end of the run when the data becomes highly centralized and the compressibility of the data varies across different partitions.

In this case, each process holds a partition size of $256 \times 512 \times 512$. Following our framework design described in Section 4, we use a fine-grained compression block size of 8.39 MB, resulting in 32 data blocks per process. It is important to note that we deliberately employ a non-integer block size to ensure an evenly divided distribution of data blocks without significant size discrepancies.

Next, we measure the time required to perform lossy compression and write operations for each data block. It is worth mentioning that in this section, our focus is on evaluating the performance of the proposed task scheduling algorithms. To determine the most suitable algorithm, we utilize actual values of compression time, I/O time, and computation intervals instead of relying on predicted values.

We observe that the overall performance of our proposed framework, as presented in this section, is slightly better than that in subsequent sections that employ predicted values. This discrepancy can primarily be attributed to the inherent uncertainty associated with predicting compression time, I/O time, and the intervals between neighboring simulation iterations for each data block.

Table 1 presents the average scheduled iteration time for each algorithm. In addition, we also evaluate the Integer Linear Program (ILP) as one of the task scheduling algorithms, described in the appendix. While it can sometimes offer the fastest iteration time, its computation time is significantly longer than the other algorithms. In general, Johnson's algorithms with backfilling demonstrate the best overall performance, considering both execution time and overhead. Thus, we adopt Johnson's algorithms with backfilling for our subsequent evaluations.

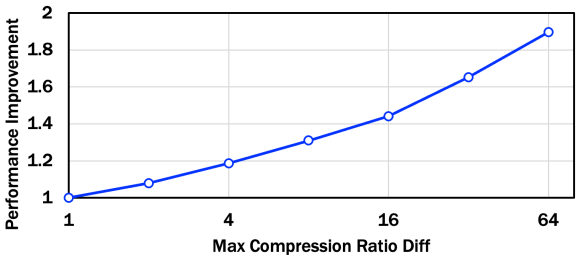


Figure 3. Relative Performance improvements using our proposed intra-node I/O workload balancing technique. The max compression ratio difference represents the variance in compression ratios between partitions with the highest and lowest compression ratios within a single node.

It is worth noting that in cases where the CPU and I/O idle time during the simulation significantly exceeds the time required for compression and compressed data I/O operations (e.g., when data is dumped at relatively longer step sizes), our selected task scheduling algorithm has the potential to effectively mitigate the impact of compression and I/O on the simulation. In other words, the scheduling algorithm can minimize the perceived delays caused by these operations, enabling the simulation to proceed seamlessly.

Next, we evaluate the effectiveness of our proposed I/O workload balancing techniques in combination with the task scheduling algorithm. We assess their performance across various data distribution scenarios by assuming a maximum compression ratio difference among processes within a given node with 4 to 8 processes (i.e., GPUs). We further assume that the compression ratios for each process follow a normal distribution based on this maximum difference.

Figure 3 shows the relative performance improvement in execution time achieved by the I/O workload balancing technique, compared to the original execution time. We observe that as the compression ratio differences between processes increase, the I/O workload balancing technique provides a higher performance improvement. It is important to note that the specific value of the maximum compression ratio difference is highly dependent on the characteristics of the HPC application being performed. For instance, in the case of the Nyx Application, this number can reach as high as 20 with an average compression ratio of 16×. In worst-case scenarios where the maximum compression ratio difference is extremely low, the I/O workload balancing technique does not introduce additional overhead to the system.

5.3 Evaluation of Proposed Compression Design

In this section, we evaluate the efficiency of the three designs we proposed in Section 4: fine-grained compression, compressed data buffer, and shared Huffman tree. We begin

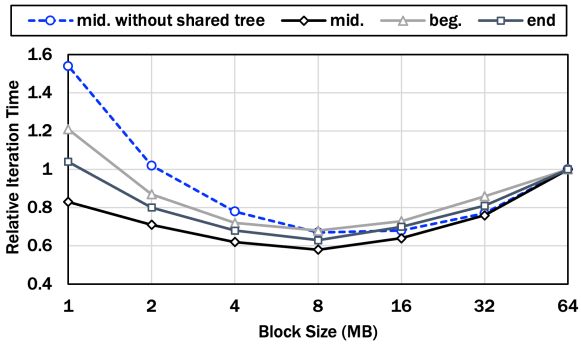


Figure 4. Execution time comparison with different block sizes (relative to the execution time using a block size of 64 MB). The dashed blue line represents the execution time without using the proposed shared Huffman tree strategy.

by evaluating the benefits derived from fine-grained compression. For this experiment, we select three stages of a Nyx Application, performed at a scale of $512 \times 512 \times 512$ using 8 GPUs. Each process in this setup manages a partition size of $256 \times 256 \times 256$, with each data field approximately sized at 64 MB. Figure 4 illustrates the relative execution time with different compression block sizes. The relative execution time is measured in terms of the execution time without fine-grained compression, which is 64 MB in this case. During this experiment, we utilize fine-grained compression, a compressed data buffer of 20 MB, the shared Huffman tree, and employ ExtJohnson+BF for scheduling. To accurately evaluate the design efficiency of fine-grained compression alone, we also utilize the actual compression time, I/O time, and interval values instead of relying on predicted values.

First, we observe that fine-grained compression effectively enhances overall performance across various data distribution scenarios. However, it is crucial to consider the block size used, as excessively small block sizes can nullify the performance benefits of fine-grained compression. This is primarily due to a significant decrease in compression and I/O throughput. We also observe that the shared Huffman tree can significantly improve compression throughput when the data block size is small. In this case, we find that using a block size of 8-16 MB when compressing the data and evenly dividing each data field works best. When comparing different data distributions, we find that fine-grained compression offers similar performance improvements for evenly distributed data during the early stage of the run, as well as for structured data with a wider range of compression ratios, such as the middle stage of the run. This is because the primary advantage of fine-grained compression lies not only in initiating I/O operations earlier but also in accommodating more compression and I/O tasks within the computation application intervals. As a result, the benefits of fine-grained compression are less dependent on the specific characteristics of the data and their compressibility.

Next, we evaluate the effectiveness of utilizing a compressed data buffer. For this experiment, we use the same

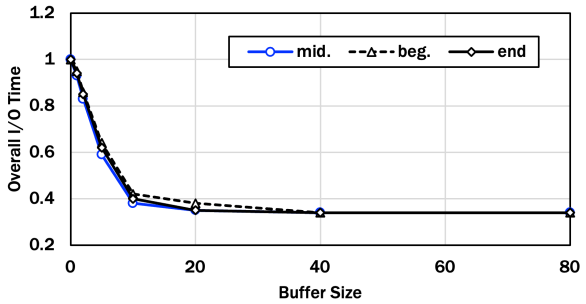


Figure 5. Execution time comparison with different buffer sizes (relative to the execution time without any compressed data buffer).

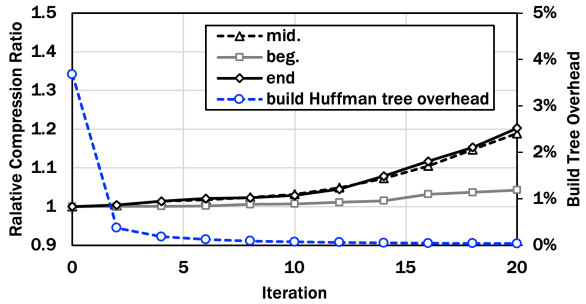


Figure 6. Compression ratio degradation across iterations, assuming the Huffman tree is built based on the data from iteration 0. The relative compression ratio means the ratio degradation between compress while "reusing the shared Huffman tree from x iterations ago" and "building a new tree".

configurations as the fine-grained compression evaluation, with a compression block size of 8 MB. Figure 5 presents the relative time of the combined I/O tasks for compressed data, considering different buffer sizes. Notably, the compressed data buffer demonstrates an efficient reduction in overall I/O time. This is because of the relatively small size of the compressed data, which allows for improved write performance through the consolidation of these small data units. In addition, our evaluation reveals that the performance improvement achieved through the use of a compressed data buffer is consistent across data from different HPC applications, independent of the data structure and compression ratio distribution. Based on our findings, we determine that a compressed data buffer size of 20 MB delivers the optimal performance enhancement while minimizing the memory footprint required for the buffer.

Finally, we evaluate the effectiveness of utilizing the shared Huffman tree. For this experiment, we use the same configurations as the previous experiments, with a compression block size of 8 MB and a compressed data buffer of 20 MB. Figure 6 shows the relative compression ratio degradation compared to using a native Huffman tree for the given data, when reusing the same Huffman tree for multiple iterations.

We notice that during the early stages/iterations where data movement is relatively stable, the shared Huffman tree can be effectively utilized for a greater number of iterations. Moreover, by constructing the shared Huffman tree based on data from the previous iteration (e.g., iteration number

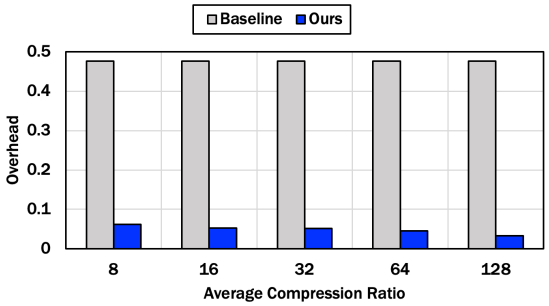


Figure 7. Time overheads (relative to the computation time) of the baseline and our solution with different compression ratios.

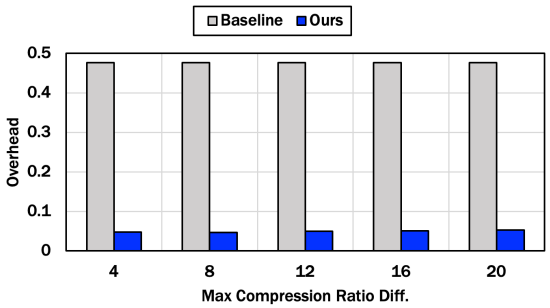


Figure 8. Time overheads (relative to the computation time) of the baseline and our solution with different data distributions (represented as the intra-node maximum compression ratio difference).

1 in Figure 6), we can achieve minimal compression ratio degradation while introducing minimal overhead in terms of rebuilding the Huffman tree at the end of each iteration.

5.4 Overall Performance Improvement

In this section, we combine the aforementioned task scheduling step with our proposed compression design to real-world HPC applications to evaluate the performance improvement from our proposed solution. We begin by evaluating the performance improvement based on simulations, allowing us to gather valuable insights and further validate our design approach. Next, we present the in situ evaluation conducted with Nyx and WarpX, showcasing the practical application and effectiveness of our framework in real-world scenarios.

5.4.1 Simulation-based evaluation. Our simulation evaluations primarily focus on assessing the efficiency of our framework across different overall compression ratios and data distributions, as evaluating with real-world applications is challenging. We gather the base computation intervals from Nyx application. Then, we assume that the uncertainty associated with the start and end times of these intervals between neighboring iterations follows a normal distribution. The variance of this normal distribution is determined by $\sigma = 0.01 \times (end_n - beg_n)$, where end_n represents the end time of the given iteration, and beg_n represents the start time of the given iteration. This assumption is based on our observations from Nyx and WarpX applications. Similarly, we assume a normal distribution for the uncertainties of the compression ratio, compression throughput, and compressed data I/O time estimation. We use $\sigma = 0.1 \times R$ for compression ratio, $\sigma = 0.05 \times T_c$ for compression throughput, and $\sigma = 0.05 \times T_{io}$ for I/O throughput. Here, R is the estimated compression ratio of each data blocks, T_c is the estimated compression throughput, and T_{io} is the estimated I/O time. To handle potential conflicts between compression, I/O, and computation intervals, we make the straightforward assumption that both CPU tasks and background tasks are executed sequentially. This means that if a CPU interval takes longer than anticipated, it can result in delays for subsequent compression tasks, potentially introducing overhead to the application.

During the simulation, our main comparison is between the performance of our solution and the baseline approach where the data is not compressed, and the I/O operations are performed sequentially with the computation. We evaluate the performance by measuring the time overhead incurred on the computation per iteration.

Figure 7 illustrates the performance of our solution across different average compression ratios. We observe that our solution consistently outperforms the baseline approach across various compression ratios. Notably, when the compression ratio is high, our solution exhibits slightly better performance. This can be attributed to the smaller size of the compressed data, resulting in shorter I/O time. As a result, the overall time spent on I/O is reduced, and our scheduling design benefits from more efficient allocation of I/O tasks within the computation intervals.

Figure 8 shows the performance of our solution across different data structure. We use the maximum compression ratio differences to represent the data structure. Lower maximum compression ratio differences typically indicate that the data is evenly distributed among the processes. We observe that our solution consistently outperforms the baseline across various data structures. However, when the maximum compression ratio difference is high, our solution exhibits slightly worse performance. This is primarily due to imbalanced workloads across different processes. Nevertheless, our I/O workload balancing design helps mitigate the negative impact of workload imbalances. By efficiently redistributing tasks among processes, our solution minimizes the effects of workload variations, resulting in overall improved performance compared to the baseline approach.

5.4.2 Real-system-based evaluation. Finally, we evaluate our solution with real-world applications running on the HPC system and compare it to the baseline and the previous solution, which only uses asynchronous I/O without compression or task scheduling techniques.

Figure 9 shows the overall performance improvement of our solution in comparison to the baseline and previous solutions on the Nyx application. Additionally, we provide

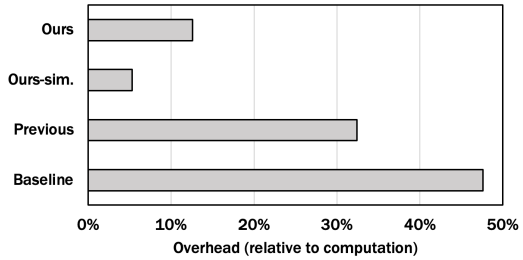
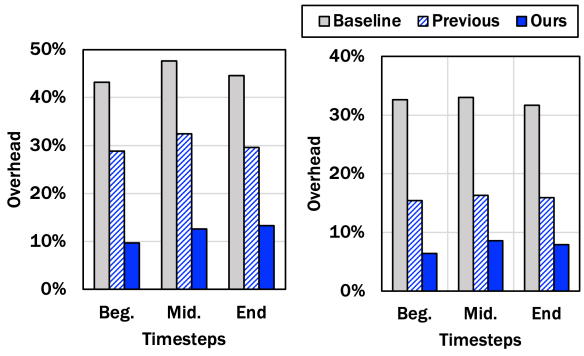
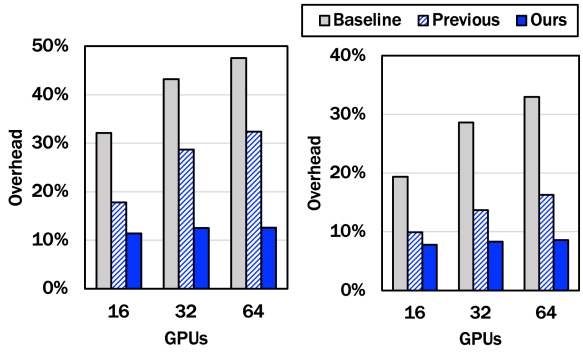


Figure 9. Time overheads (compared to computation time) of the baseline, asynchronous I/O, and our solution (with simulation for reference) with Nyx using 16 nodes and 64 GPUs.



(a) Nyx **(b)** WarpX

Figure 10. Time overheads of (compared to computation time) between the baseline, asynchronous I/O, and our solution across different timesteps with Nyx and WarpX.



(a) Nyx **(b)** WarpX

Figure 11. Time overheads (compared to computation time) of the baseline, asynchronous I/O, and our solution with different scales of Nyx and WarpX.

simulation results for reference. We note that the real implementation results in slightly larger overhead compared to the simulation results. This is because the real implementation encounters more unexpected task interference due to (1) uncertainty of computing application intervals, and (2) inaccurately predicted compression ratio, compression time and/or I/O time for compressed data. Nonetheless, our solution still achieves a significant performance improvement, with a 3.78× and 2.57× improvement over the baseline and previous solutions, respectively.

Figure 10 presents the performance of our solution across different stages of the application: the beginning, middle, and end. Our solution consistently outperforms the previous solution across all stages of the application.

We also conduct a weak scaling evaluation to assess the performance of our solution as the problem size scales with the number of processes (GPUs), as shown in Figure 11. In this experiment, the problem scale for each process is $256 \times 256 \times 256$ for Nyx and $128 \times 128 \times 1024$ for WarpX. Once again, our solution consistently outperforms the previous solution across all tested scales.

It is worth noting that both the baseline and the previous solution experience longer execution times as the scale increases. In contrast, our solution exhibits more consistent performance across different scales. This can be attributed to our data compression approach, where the overhead primarily arises from conflict tasks resulting from mis-predicted compression and I/O operations.

6 Conclusion and Future Work

Lossy compression and asynchronous I/O are two efficient solutions for reducing storage overhead and improving I/O performance in large-scale HPC/scientific applications. However, existing implementations have limitations that hinder the full utilization of lossy compression and can lead to task collisions, limiting the overall application performance. To address these challenges, we propose an optimization approach for the task scheduling problem involving application computation, compression, and I/O. Experimental results with up to 64 GPUs from Summit demonstrate that our solution reduces I/O overhead by up to 3.8× and 2.6× compared to the non-compression and asynchronous I/O solutions, respectively. Note that the efficacy of the proposed solution might decrease when the application is more reliant on CPUs than GPUs, leading to fewer CPU idle periods.

In the future, we plan to expand the integration of our solution to additional parallel I/O libraries, such as ADIOS, and evaluate its performance with a wider range of real-world HPC applications. Furthermore, we intend to extend our proposed task scheduling method and compression design to accommodate multi-file scenarios, where the dumping data is stored in multiple files for specific HPC applications.

Acknowledgments

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contracts DE-AC02-06CH11357 and DE-AC02-05CH11231. This work was also supported by the National Science Foundation under Grants 2003709, 2303064, 2104023, 2247080, 2247060, 2312673, 2311875, and 2311876.

References

[1] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science* 19, 5–6 (2018), 65–76.

[2] Ann Almgren, Vince Beckner, Chris Daley, Brian Friesen, Zarija Lukic, Andrew Myers, Jean Sexton, and Weiqun Zhang. 2023. *Nyx*. https://github.com/AMReX-Astro/Nyx

[3] Ann S Almgren, John B Bell, Mike J Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A massively parallel amr code for computational cosmology. *The Astrophysical Journal* 765, 1 (2013), 39.

[4] Babak Behzad, Surendra Byna, Stefan M Wild, Mr Prabhat, and Marc Snir. 2014. Improving parallel I/O autotuning with performance modeling. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 253–256.

[5] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. 2013. Taming parallel I/O complexity with auto-tuning. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[6] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. 2016. Accelerating science with the NERSC burst buffer early user program. *Lawrence Berkeley National Laboratory* (2016).

[7] Robert Bird, Nigel Tan, Scott V Luedtke, Stephen Lien Harrell, Michela Taufer, and Brian Albright. 2021. VPIC 2.0: Next generation particle-in-cell simulations. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 952–963.

[8] J. Breit, G. Schmidt, and V. A. Strusevich. 2003. Non-preemptive two-machine open shop scheduling with non-availability constraints. *Mathematical Methods of Operations Research* 57 (2003), 217–234.

[9] Suren Byna, M Scot Breitenfeld, Bin Dong, Quincey Koziol, Elena Pourmal, Dana Robinson, Jerome Soumagne, Houjun Tang, Venkatram Vishwanath, and Richard Warren. 2020. ExaHDF5: delivering efficient parallel I/O on exascale computing systems. *Journal of Computer Science and Technology* 35, 1 (2020), 145–160.

[10] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. 2021. Tuning HDF5 subfiling performance on parallel file systems. *Lawrence Berkeley National Laboratory* (2021).

[11] Surendra Byna, Jerry Chou, Oliver Rubel, Homa Karimabadi, William S Daughter, Vadim Roytershteyn, E Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, et al. 2012. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[12] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* (2019).

[13] Jieyang Chen, Lipeng Wan, Xin Liang, Ben Whitney, Qing Liu, David Pugmire, Nicholas Thompson, Jong Youl Choi, Matthew Wolf, Todd Munson, et al. 2021. Accelerating multigrid-based hierarchical scientific data refactoring on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 859–868.

[14] Sheng Di. 2023. *H5Z-SZ*. https://github.com/disheng222/H5Z-SZ

[15] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 730–739.

[16] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Chicago, IL, USA, 730–739.

[17] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.

[18] Oak Ridge Leadership Computing Facility. 2023. *Summit supercomputer*. https://www.olcf.ornl.gov/summit/

[19] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaim, W. Zhang, J. Vay, and H. Vincenti. 2022. Pushing the Frontier in the Design of Laser-Based Electron Accelerators with Groundbreaking Mesh-Refined Particle-In-Cell Simulations on Exascale-Class Supercomputers. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/SC41404.2022.00008

[20] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 2011. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. 36–47.

[21] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[22] Pascal Grosset, Christopher Biwer, Jesus Pulido, Arvind Mohan, Ayan Biswas, John Patchett, Terece Turton, David Rogers, Daniel Livescu, and James Ahrens. 2020. Foresight: analysis that matters for data reduction. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1171–1185.

[23] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, Vishwanath Venkatram, Lukić Zarija, Sehrish Saba, and Wei-keng Liao. 2016. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy* 42 (2016), 49–65.

[24] Jaehyun Han, Donghun Koo, Glenn K Lockwood, Jaehwan Lee, Hyeonsang Eom, and Soonwook Hwang. 2017. Accelerating a burst buffer via user-level i/o isolation. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 245–255.

[25] HDF Group and others. 2000. Hierarchical data format version 5, Filter.

[26] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. 2019. Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 159–170.

[27] Sian Jin, Sheng Di, Jiannan Tian, Suren Byna, Dingwen Tao, and Franck Cappello. 2022. Improving Prediction-Based Lossy Compression Dramatically Via Ratio-Quality Modeling. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2494–2507.

[28] Sian Jin, Pascal Grosset, Christopher M Biwer, Jesus Pulido, Jiannan Tian, Dingwen Tao, and James Ahrens. 2020. Understanding GPU-based lossy compression for extreme-scale cosmological simulations. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 105–115.

[29] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. 2020. Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 45–56.

[30] Sian Jin, Dingwen Tao, Houjun Tang, Sheng Di, Suren Byna, Zarija Lukic, and Franck Cappello. 2022. Accelerating parallel write via deeply integrating predictive lossy compression with HDF5. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–15.

[31] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2021. COMET: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. *Proceedings of the VLDB Endowment* 15, 4 (2021), 886–899.

[32] S. M. Johnson. 1954. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* 1, 1 (1954), 61–68.

[33] Donghun Koo, Jaehwan Lee, Jialin Liu, Eun-Kyu Byun, Jae-Hyuck Kwak, Glenn K Lockwood, Soonwook Hwang, Katie Antypas, Kesheng Wu, and Hyeonsang Eom. 2021. An empirical study of I/O separation for burst buffers in HPC systems. *J. Parallel and Distrib. Comput.* 148 (2021), 96–108.

[34] Matthew Larsen and Peter Lindstrom. 2023. *cuZFP*. https://github.com/LLNL/zfp/tree/develop/src/cuda_zfp

[35] Chung-Yee Lee. 1997. Minimizing the makespan in the two-machine flowshop scheduling problem with an availability constraint. *Operations Research Letters* 20, 3 (1997), 129–139. https://doi.org/10.1016/S0167-6377(96)00041-7

[36] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. IEEE, 39–39.

[37] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data*. IEEE, 438–447.

[38] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2022. SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors. *IEEE Transactions on Big Data* (2022), 1–14. https://doi.org/10.1109/TBDATA.2022.3201176

[39] David A. Lifka. 1995. The ANL/IBM SP scheduling system. In *Job Scheduling Strategies for Parallel Processing*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 295–303.

[40] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[41] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Matthew Wolf, Tong Liu, and Zhenbo Qiao. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 348–357.

[42] Huizhang Luo, Dan Huang, Qing Liu, Zhenbo Qiao, Hong Jiang, Jing Bi, Haitao Yuan, Mengchu Zhou, Jinzhen Wang, and Zhenlu Qin. 2019. Identifying Latent Reduced Models to Precondition Lossy Compression. In *2019 IEEE International Parallel and Distributed Processing Symposium*. IEEE.

[43] Oak Ridge Leadership Computing Facility. 2023. *WarpX, granted early access to the exascale supercomputer Frontier, receives the high-performance computing world's highest honor.* https://www.olcf.ornl.gov/2022/11/17/plasma-simulation-code-wins-2022-acm-gordon-bell-prize/ Online.

[44] Santosh Pokhrel, Miguel Rodriguez, Alireza Samimi, Gerd Heber, and Jamesina J Simpson. 2018. Parallel I/O for 3-D global FDTD earth–ionosphere waveguide models at resolutions on the order of˜ 1 km and higher using HDF5. *IEEE Transactions on Antennas and Propagation* 66, 7 (2018), 3548–3555.

[45] Andrew Siegel, Erik Draeger, Jack Deslippe, Thomas Evans, Marianne Francois, Timothy C Germann, Daniel F Martin, and William Hart. 2022. *Application Results on Early Exascale Hardware.* Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[46] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing Frontiers and Innovations* 1, 2 (2014), 76–88.

[47] Houjun Tang, Suren Byna, N Anders Petersson, and David McCallen. 2021. Tuning parallel data compression and I/O for large-scale earthquake simulation. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2992–2997.

[48] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. 2021. Transparent asynchronous parallel i/o using background threads. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 891–902.

[49] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. 2022. Transparent Asynchronous Parallel I/O Using Background Threads. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 891–902. https://doi.org/10.1109/TPDS.2021.3090322

[50] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1129–1139.

[51] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1129–1139.

[52] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2019. Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP. *IEEE Transactions on Parallel and Distributed Systems* 30, 8 (2019), 1857–1871.

[53] The HDF Group. [n. d.]. *Hierarchical data format version 5*. http://www.hdfgroup.org/HDF5

[54] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, et al. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 3–15.

[55] Shu-Mei Tseng, Bogdan Nicolae, Franck Cappello, and Aparna Chandramowlishwaran. 2021. Demystifying asynchronous I/O Interference in HPC applications. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 391–412.

[56] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (1992), xviii–xxxiv.

[57] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. Analysis and modeling of the end-to-end i/o performance on olcf's titan supercomputer. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 1–9.

[58] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. 2017. Comprehensive measurement and analysis of the user-perceived I/O performance in a production leadership-class storage system. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1022–1031.

[59] Daoce Wang, Jesus Pulido, Pascal Grosset, Jiannan Tian, Sian Jin, Houjun Tang, Jean Sexton, Sheng Di, Zarija Lukić, Kai Zhao, et al. 2023. AMRIC: A Novel In Situ Lossy Compression Framework for Efficient I/O in Adaptive Mesh Refinement Applications. *arXiv preprint arXiv:2307.09609* (2023).

[60] Chengming Zhang, Sian Jin, Tong Geng, Jiannan Tian, Ang Li, and Dingwen Tao. 2022. CEAZ: accelerating parallel I/O via hardware-algorithm co-designed adaptive lossy compression. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–13.

[61] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software* 4, 37 (2019), 1370.

[62] Huihuo Zheng, Venkatram Vishwanath, Quincey Koziol, Houjun Tang, John Ravi, John Mainzer, and Suren Byna. 2022. HDF5 Cache VOL: Efficient and Scalable Parallel I/O through Caching Data on Node-local Storage. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 61–70.

[63] Q Zhou, C Chu, NS Kumar, P Kousha, SM Ghazimirsaeed, H Subramoni, and DK Panda. 2021. Designing High-Performance MPI Libraries with On-the-fly Compression for Modern GPU Clusters. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 444–453.

A Integer Linear Program

We provide the ILP discussed in Section 3.2. We need to introduce some new variables for this ILP:

- The binary variable $first_{i,j}^{(X)}$ is equal to 1 if and only if task $X_{n,i}$ precedes task $X_{n,j}$ (for X being either R or B). We always have $first_{i,j}^{(X)} = 1 - first_{j,i}^{(X)}$. Hence, to decrease the number of variables, we will only consider the variables $first_{i,j}^{(X)}$ when $i < j$.
- The binary variable $\delta_{i,h}^{(X)}$ is true if and only if task $X_{n,i}$ is executed between the $(h-1)$-th and the h-th unavailabiliy interval (for X being either R or B). That is, if we consider compression tasks, between the times $beg_n + b_{n,h-1}$ and $beg_n + a_{n,h}$, for $1 \leq h \leq 1+k$, with the convention that $b_{n,0} = 0$ and $a_{n,k+1} = +\infty$.
- $\mathbb{Z}$ is a very large value, larger than the makespan of the optimal solution. One can choose for $\mathbb{Z}$ the makespan of a naive schedule (schedule the tasks as soon as possible in the order 1, ..., m.).

The ILP to find an optimal solution to the scheduling problem is presented in Figure 12 (where for each equation, we have $1 \leq i < j \leq m$, and $X \in \{R, B\}$). Here is a walkthrough:

- Equation (1): the iteration completion time is greater than the maximum of the completion time of the I/O tasks.
- Equation (2): an I/O task cannot start before the completion of the corresponding compression task.
- Equations (3) and (4): the completion time of a task is equal to its start time plus its execution time.
- Equation (5): if task $X_{n,i}$ precedes task $X_{n,j}$ in the schedule, then task $X_{n,i}$ must be completed before task $X_{n,j}$ can start. If $X_{n,i}$ precedes $X_{n,j}$ then $first_{i,j}^{(x)}$ is equal to 1 and Equation (5) is equal to $t_{start}(X_{n,j}) \geq t_{end}(x_{n,i})$. If, on the contrary, $X_{n,j}$ precedes $X_{n,i}$ then $first_{i,j}^{(X)}$ is equal to 0 and Equation (5) is equal to $t_{start}(X_{n,j}) \geq t_{end}(X_{n,i}) - \mathbb{Z}$ and Equation (5) is not constraining.
- Equation (6): if task $X_{n,j}$ precedes task $X_{n,i}$ in the schedule, then task $X_{n,j}$ must be completed before task $X_{n,i}$ can start. If $X_{n,j}$ precedes $X_{n,i}$ then $first_{i,j}^{(X)}$ is equal to 0 and Equation (6) is equal to $t_{start}(x_{n,i}) \geq t_{end}(x_{n,j})$. If, on the contrary, $X_{n,i}$ precedes $X_{n,j}$ then $first_{i,j}^{(X)}$ is equal to 1 and Equation (6) is equal to $t_{start}(X_{n,i}) \geq t_{end}(X_{n,j}) - \mathbb{Z}$ and Equation (6) is not constraining.
- Equations (7) and (8): if binary variable $\delta_{i,h}^{(R)}$ (resp. $\delta_{i,h}^{(B)}$) is equal to 1, then task $R_{n,i}$ (resp. $B_{n,i}$) can start at the earliest at time $end_n + b_{n,i-1}$ (resp. $end_n + b'_{n,i-1}$), the end of the $(i-1)$-th unavailability interval.
- Equations (9) and (10): if binary variable $\delta_{i,h}^{(R)}$ (resp. $\delta_{i,h}^{(B)}$) is equal to 1, then task $R_{n,i}$ (resp. $B_{n,i}$) can complete at the latest at time $beg_n + a_{n,i-1}$ (resp. $beg_n + a_{n,i-1}$), the start of the i-th unavailability interval.
- Equations (11) and (12): all tasks must be executed at some point.

Minimize $T_n^{overall}$ subject to

$$
\begin{cases}
T_n^{overall} \geq t_{end}(B_{n,i}) & (1)\\
t_{end}(R_{n,i}) \leq t_{start}(B_{n,i}) & (2)\\
t_{end}(R_{n,i}) = t_{start}(R_{n,i}) + c_{n,i} & (3)\\
t_{end}(B_{n,i}) = t_{start}(B_{n,i}) + c'_{n,i} & (4)\\
t_{start}(X_{n,j}) \geq t_{end}(X_{n,i}) - (1 - first_{i,j}^{(X)})\mathbb{Z} & (5)\\
t_{start}(X_{n,i}) \geq t_{end}(X_{n,j}) - first_{i,j}^{(X)}\mathbb{Z} & (6)\\
\sum_{h=1}^{1+k} \delta_{i,h}^{(R)}(beg_n + b_{n,i}) \leq t_{start}(R_{n,i}) & (7)\\
\sum_{h=1}^{1+o} \delta_{i,h}^{(B)}(beg_n + b'_{n,i}) \leq t_{start}(B_{n,i}) & (8)\\
t_{end}(R_{n,i}) \leq \sum_{h=1}^{k+1} \delta_{i,h}^{(R)}(beg_n + a_{n,i}) & (9)\\
t_{end}(B_{n,i}) \leq \sum_{h=1}^{o+1} \delta_{i,h}^{(B)}(beg_n + a'_{n,i}) & (10)\\
\sum_{h=1}^{1+k} \delta_{i,h}^{(R)} = 1 & (11)\\
\sum_{h=1}^{1+o} \delta_{i,h}^{(B)} = 1 & (12)
\end{cases}
$$

Figure 12. Integer Linear Program optimally solving the scheduling problem.

B Artifact Appendix

B.1 Artifact DOI

10.5281/zenodo.8394043

B.2 Abstract

Within this artifact, we offer a comparative analysis, benchmarking our solution against two alternative approaches: (1) the previous method employing asynchronous writes without data compression, and (2) the baseline solution uses synchronous data writes without compression. This alignment with our paper underscores the performance enhancements our solution delivers.

Furthermore, we conducted our artifact implementation on Chameleon Cloud, utilizing a Singularity container to ensure optimal applicability across various computing environments. The test node on Chameleon Cloud is equipped with two Intel Xeon E5-2660 CPUs and 128 GB of memory, specifically configured with gpu.model=P100. We strongly recommend to use Chameleon Cloud platform for assessments with consistency and reproducibility.

B.3 Description & Requirements

B.3.1 How to Access. https://github.com/jinsian/EuroSys-AsyncSchedule4IO.

B.3.2 Description of Experiment Workflow. The entire workflow takes approximately 15 minutes to execute, including downloading container image and preparing environment (4 mins), running WarpX simulation (5 mins), running Nyx simulation (5 mins), and evaluating performance (1 min).

B.3.3 Minimum System Requirements.

- OS: Ubuntu (20.04 is recommended)
- GPU: Nvidia GPUs with CUDA >= 12.2
- Memory: >= 16 GB RAM
- Processor: >= 16 cores
- Storage: >= 32 GBs

B.4 Set-up

• Step 1: Install Singularity

Please refer to:
https://singularity-tutorial.github.io/01-installation/.

• Step 2: Download, Build, and run the image file (need root privilege) with singularity

You can download, build, and run the image file that encompasses all the necessary components.

```
sudo pip3 install gdown
gdown  https://drive.google.com/uc?id=1o0AumoDJgnZK\
cLXv-ZH7b5lGhKzP4_6A
sudo singularity build --sandbox artiAsync \
AsyncSchedule.sif
sudo singularity shell --writable artiAsync
```

Now, you are running inside of the container.

B.5 Evaluation Workflow

• Step 3: Set up environmental variables

```
export OMPI_DIR=/opt/ompi
export OMPI_VERSION=4.1.1
export PATH=$OMPI_DIR/bin:$PATH
export LD_LIBRARY_PATH=$OMPI_DIR/lib:$LD_LIBRARY_PATH
export MANPATH=$OMPI_DIR/share/man:$MANPATH
export C_INCLUDE_PATH=/opt/ompi/include\
:$C_INCLUDE_PATH
export CPLUS_INCLUDE_PATH=/opt/ompi/include\
:$CPLUS_INCLUDE_PATH
export OMPI_ALLOW_RUN_AS_ROOT=1
export OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
```

• Step 4: Run Nyx simulation with (1) baseline, (2) previous, and (3) ours

```
cd /home/EuroSys-AsyncSchedule/
bash ./runnyx.sh
```

• Step 5: Run WarpX simulation with (1) baseline, (2) previous, and (3) ours

```
cd /home/EuroSys-AsyncSchedule/
bash ./runwarpx.sh
```

• Step 6: (Optional): We retaine log files for all runs, and now you can check them out

```
head -n 200 ./Nyx/Exec/LyA/test1.txt
head -n 100 ./WarpX/test1.txt
```

The expected results for Nyx's log are:

```
Nyx::est_time_step at level 0:  estdt = 1.365270159e-07
Integrating a from time 1.345051497e-07 by dt = 1.36527
Old / new A time      1.345051497e-07 2.710321656e-07
Old / new A           0.00631247703 0.0063755786
Old / new z           157.416418 155.8485094
Re-integrating a from time 1.345051497e-07 by dt = 1.3
Old / new A time       1.345051497e-07 2.710321656e-07
Old / new A              0.00631247703 0.0063755786
Old / new z              157.416418 155.8485094
[Level 0 step 2] ADVANCE at time 1.345051497e-07 with
Gravity ... multilevel solve for old phi at base level
ParticleContainer::AssignCellDensitySingleLevel) time:
 ... subtracting average density 3.760710576e+10 from
 ... subtracting -2.861022949e-06 to ensure solvabil
MLMG: Initial rhs               = 103869.8512
MLMG: Initial residual (resid0) = 103869.8512
MLMG: Final Iter. 10 resid, resid/bnorm = 2.480104285
MLMG: Timers: Solve = 1.333416157 Iter = 1.276759186
moveKickDrift ... updating particle positions and vel
Gravity ... single level solve for new phi at level 0
ParticleContainer::AssignCellDensitySingleLevel) time:
 ... solve for phi at level 0
 ... subtracting average density from RHS in solve ...
 ... subtracting 3.531575203e-06 to ensure solvability
MLMG: Initial rhs               = 101952.6971
MLMG: Initial residual (resid0) = 6501.955658
MLMG: Final Iter. 7 resid, resid/bnorm = 2.565720933
MLMG: Timers: Solve = 0.940721187 Iter = 0.901259195
```

The expected results for WarpX's log are:

```
STEP 3 starts ...
--- INFO    : Writing plotfile diags/plt000003
STEP 3 ends. TIME = 3.25787071e-16 DT = 1.085956903e-16
Evolve time = 10.47966713 s; This step = 3.457123984 s;

STEP 4 starts ...
```

• Step 7: Evaluate Nyx's perormance between (1) baseline, (2) previous, and (3) ours

```
cd $TEST_HOME/Nyx/Exec/LyA
python3 ./readresults.py test1.txt test2.txt \
test3.txt test4.txt
```

• Step 8: Evaluate WarpX's perormance between (1) baseline, (2) previous, and (3) ours

```
cd $TEST_HOME/WarpX/
python3 ./readresults.py test1.txt test2.txt \
test3.txt test4.txt
```

The expected results for Nyx's performance comparison are:

```
Sample from 10 iterations.
-------------------- Baseline --------------------
Baseline: no compression, no asynchronous write.
Nyx simulation with Baseline solution time: 47.08 s
Baseline overhead compared to computation only: 37.2 %
-------------------- Previous --------------------
Baseline: no compression, no asynchronous write.
Nyx simulation with Previous solution time: 47.04 s
Previous overhead compared to computation only: 37.1 %
---------------------- Ours ----------------------
Baseline: no compression, no asynchronous write.
Nyx simulation with Our solution time: 37.12 s
Ours overhead compared to computation only: 8.2 %
------------------- Improvement ------------------
Our improvement compared to previous: 4.53 times
----------------------- End ----------------------
```

The expected results for WarpX's performance comparison are:

```
Sample from 10 iterations.
-------------------- Baseline --------------------
Baseline: no compression, no asynchronous write.
WarpX simulation with Baseline solution time: 38.74 s
Baseline overhead compared to computation only: 121.9 %
-------------------- Previous --------------------
Baseline: no compression, no asynchronous write.
WarpX simulation with Previous solution time: 38.52 s
Previous overhead compared to computation only: 120.6 %
---------------------- Ours ----------------------
Baseline: no compression, no asynchronous write.
WarpX simulation with Our solution time: 23.87 s
Ours overhead compared to computation only: 36.7 %
------------------- Improvement ------------------
Our improvement compared to previous: 3.29 times
----------------------- End ----------------------
```

Please note that the performance may vary on different machines and environments. Nevertheless, you should be able to discern the performance improvements our solution offers compared to previous approaches. These results are consistent with our paper's findings. Please be aware that the runtime may vary, particularly when resources are limited. We highly recommend running steps 4, 5, 7, and 8 multiple times to observe consistent results.

This result is primarily correlated to the main claim of our paper, shown in Figure 9. When comparing the relative overhead with both the original and previous solutions, our approach effectively enhances the end-to-end performance of the simulation.

Additionally: (1) You can adjust the number of simulation iterations by modifying "max_step = 10" in "EuroSys-AsyncSchedule4IO/Nyx/Exec/LyA/inputs" for Nyx, or "max _step = 10" in "EuroSys-AsyncSchedule4IO/WarpX/inputs.", and re-run steps 4, 5, 7, and 8. You should observe consistent performance improvements, regardless of the number of simulation iterations, as shown in Figure 10. (2) you can modify the number of processes by changing all parameters of "-np 16" in "EuroSys-AsyncSchedule4IO/runnyx.sh" and "EuroSys-AsyncSchedule4IO/runwarpx.sh.", and re-run steps 4, 5, 7, and 8. You should observe consistent performance improvements, regardless of the number of processes, as shown in Figure 11.