Towards Affordable Reproducibility Using Scalable Capture and Comparison of Intermediate Multi-Run Results

Nigel Tan*
University of Tennessee Knoxville
Knoxville, TN, USA
ntan1@vols.utk.edu

Kevin Assogba*
Rochester Institute of Technology
Rochester, NY, USA
kta7930@cs.rit.edu

Walter J. Ashworth
University of Tennessee Knoxville
Knoxville, TN, USA
washwor1@vols.utk.edu

Befikir Bogale
University of Tennessee Knoxville
Knoxville, TN, USA
bbogale@vols.utk.edu

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

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

Michela Taufer
University of Tennessee Knoxville
Knoxville, TN, USA
taufer@acm.org

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL, USA
bnicolae@anl.gov

ABSTRACT

Ensuring reproducibility in high-performance computing (HPC) applications is a significant challenge, particularly when nondeterministic execution can lead to untrustworthy results. Traditional methods that compare final results from multiple runs often fail because they provide sources of discrepancies only a posteriori and require substantial resources, making them impractical and unfeasible. This paper introduces an innovative method to address this issue by using scalable capture and comparing intermediate multi-run results. By capitalizing on intermediate checkpoints and hash-based techniques with user-defined error bounds, our method identifies divergences early in the execution paths. We employ Merkle trees for checkpoint data to reduce the I/O overhead associated with loading historical data. Our evaluations on the nondeterministic HACC cosmology simulation show that our method effectively captures differences above a predefined error bound and significantly reduces I/O overhead. Our solution provides a robust and scalable method for improving reproducibility, ensuring that scientific applications on HPC systems yield trustworthy and reliable results.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**.

KEYWORDS

Results reproducibility, Checkpoint analysis, High-performance computing, Error-bounded hashing

ACM Reference Format:
Nigel Tan, Kevin Assogba, Walter J. Ashworth, Befikir Bogale, Franck Cappello, M. Mustafa Rafique, Michela Taufer, and Bogdan Nicolae. 2024. Towards Affordable Reproducibility Using Scalable Capture and Comparison of Intermediate Multi-Run Results. In *24th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652892.3700780

1 INTRODUCTION

The increasing complexity of scientific applications and the extreme heterogeneity they face from all perspectives (different types of tasks, patterns, accelerators, job scheduling decisions, interleaving and competition for resources, etc.) makes it challenging to reason about reproducibility [11, 18]. For example, numerous studies [23, 39, 40] have shown that concurrency in HPC applications can negatively affect the reproducibility of simulation results. Consequently, prominent HPC publication venues have begun requiring reproducibility assessments for submitted research [6, 44], and major HPC laboratories have increased investments in software tools aimed at characterizing, quantifying, and managing concurrency to enhance computational reproducibility [36, 37].

A naive solution that simply compares the end results of two different application runs that start with the same input data does not enable enough insight. For example, if the end results are different, then there is no information available about what went wrong and when this happened during the runtime. Similarly, if there is a single valid path to reach the end result (which is often the case of HPC simulations), then obtaining a correct end result does not guarantee it was obtained through the valid path that produced correct intermediate results. For example, a study by Stodden and co-workers in Figure 1 compares the outcomes of two runs of a galaxy formation simulation using the Enzo adaptive mesh refinement code [8]. In

*These authors contributed equally to this work.

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12
https://doi.org/10.1145/3652892.3700780

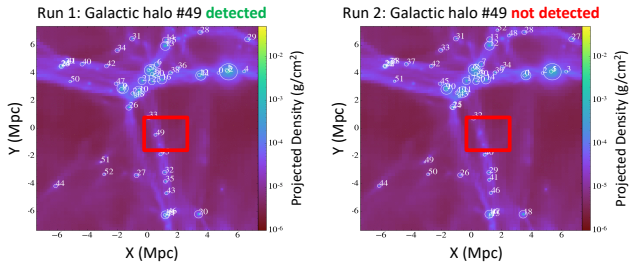


Figure 1: Discrepancy between two runs of galaxy formation simulation using the Enzo adaptive mesh refinement code [8] due to concurrency. Documented by Stodden et al. in [39].

Run 1, a specific galactic halo (49) formed, whereas in Run 2, it did not. Thus, it is important to devise scalable techniques that capture and analyze intermediate results in addition to the end results.

Limitations of State-of-the-Art. In a quest to ensure the reproducibility of scientific simulations, various strategies have been developed to either control the execution or examine the outcome of the simulations. Techniques to control determinism in scientific applications have been proposed to support reproducibility. For example, a sequential implementation of an application such that arithmetic operations follow a pre-defined order ensures that the same result is obtained across multiple runs [5]. However, this approach requires intimate knowledge of the application and can introduce additional costs to refactor legacy systems. An alternative to avoid the computational and storage overhead measures the statistical significance of the end results using derived quantities such as the variance and standard deviation [7]. An element-wise comparison and the computation of a derived quantity have similar complexities, but statistical analyses involve fewer operations, reducing computational overhead. However, the lack of detailed insights into the evolution of the simulation makes it impossible to identify the root cause of non-determinism in the end results.

This objective requires a complete history of intermediate results captured during the simulation. This can be done using checkpointing [14, 19, 26], a technique widely used in scientific applications for various tasks, e.g., suspend-resume of long-running jobs, resilience, fault tolerance, etc., to collect critical data needed to study reproducibility at runtime. Specifically, the intermediate results can be captured into a checkpoint history at key moments during the runtime. State-of-the-art checkpointing techniques use asynchronous multi-level techniques to this end. The principle is to write the intermediate results into a checkpoint file on node-local storage such as NVMe, then flush the file in the background to durable shared storage (e.g., a parallel file system such as Lustre) while the application continues in the foreground. Unfortunately, state-of-the-art checkpointing solutions are not optimized to read back the checkpointing data, which is needed to perform element-wise comparisons. This read-intensive pattern may introduce a significant I/O bottleneck, because the history of checkpoints may grow to massive sizes (many distributed processes, each of which needs to capture a large checkpoint frequently during runtime). Moreover, in addition to I/O bottlenecks, the computational overhead of element-wise comparisons can be significant.

Problem Formulation. In this paper, we study the problem of how to compare the history of checkpoints produced during two runs in order to identify if there are any differences and what variables were affected. Specifically, given a history of checkpoints A_i^j generated by $0 < i < N$ distributed processes over $0 < j < M$ iterations during Run 1, and a corresponding history of checkpoints B_i^j during Run 2, we aim to design and implement a runtime that compares A_i^j with B_i^j and lists all intermediate data (and the corresponding indices if the data are multi-dimensional) that are different between two runs. Since most modern scientific applications are a mix of HPC simulations, ML, and Big Data analytics workloads, they operate with floating-point values. Hence, we assume the intermediate data between two runs is different if $| V_1 - V_2 |> \epsilon$, where ϵ is user-defined. At scale, the amount of data captured as checkpoints is massive. Therefore, our goal is to *design and implement scalable techniques that maximize the throughput of comparing the checkpoint history of the two runs.*

Contributions. The key idea of our approach is to reduce the number of element-wise comparisons performed between the checkpoints of two runs by splitting the checkpoints into chunks, hashing the chunks, and storing additional metadata with each checkpoint. Then, if the hashes of the chunks match, we consider the content of the chunks to match as well with high probability, thereby reducing the number of chunks for which a full element-wise comparison is needed, which ultimately reduces the I/O bottlenecks and computational overheads. While this principle is widely used for verification purposes, we are not aware of any work that explored its applicability in the context of reproducibility. This introduces several opportunities for innovation, as summarized below:

(1) We propose a series of design principles: (1) novel GPU-optimized hashing techniques for groups of floating point values organized into chunks that need to match within a given error bound; (2) novel hierarchic organization of hashes using GPU-optimized data structures to accelerate the comparison of identical contiguous regions; (3) multi-level I/O pipelining of data transfers and overlapping with GPU computations to maximize parallelization and enable scalability (Section 2.1).
(2) We implement these design principles into a practical runtime that leverages modern performance portable frameworks, e.g., Kokkos, advanced data structures, e.g., GPU-aware Merkle trees, and efficient I/O libraries, e.g., io_uring, for offline (using a command line tool) or online (using a library API) scalable capture and comparison of intermediate results to enhance reproducibility in HPC applications (Section 2.5).
(3) We evaluate our method using checkpoints captured from a real-life large-scale HPC application (the HACC cosmology simulation). These experiments demonstrate the effectiveness of our method at capturing differences between checkpoints from multiple runs at a higher comparison throughput compared with popular and state-of-the-art approaches (Section 3.4).

2 SYSTEM DESIGN

2.1 Design Principles

The design of our solution to manage reproducibility disruptions due to concurrency in large-scale scientific applications on HPC systems must address several complex challenges. These challenges include handling massive datasets, optimizing I/O operations, and effectively leveraging HPC resources. We have strategically defined a series of principles that govern the design of our method to address such challenges.

GPU-Aware Error-Bounded Parallelized Comparison of Checkpoints. To address the reproducibility challenge in scientific simulations, we introduce a GPU-accelerated approach for precision floating-point comparisons within predefined error bounds. Our method starts by aligning initial checkpoints from two different simulation runs and segmenting them into smaller, manageable chunks. These segments are then distributed and processed in parallel across multiple GPUs, enabling simultaneous data comparison. Our method not only leverages GPUs' parallel processing capabilities to enhance computational efficiency but also ensures the required precision for scientific accuracy. By parallelizing the data comparison process, our method substantially decreases the time needed to detect discrepancies between simulation runs, thus improving the reliability of scientific conclusions in large-scale computational simulations.

Multi-Level Overlapping I/O Pipelining from Persistent Storage to GPU Memory. To address the challenges associated with I/O bottlenecks in GPU-based systems, we have developed a multi-level overlapping I/O pipelining technique that efficiently transfers data from persistent storage, e.g., a parallel file system (PFS), directly to GPU memory. Traditional methods involve sending data in a blocking manner, but they severely constrain throughput due to I/O operations. Our method bypasses this constraint by establishing a seamless pipeline that reads data chunks from the parallel file system into the host memory. Concurrently, these chunks are transferred to GPU memory, and comparison operations on the GPUs start without delay. This overlapping of reading, transferring, and processing not only mitigates the I/O bottleneck but also fully leverages the parallelism capabilities of GPUs. By efficiently aligning the data flow across multiple system architecture levels, we can significantly enhance the throughput and efficiency of data-intensive applications.

Reduction of I/O and Comparisons Costs using Error-Bounded Hash-Based Techniques. To address the efficiency challenge of large-scale data operations, we have implemented a novel error-bounded hash-based method to reduce both I/O operations and the number of necessary comparisons. In environments where storage repositories are shared across multiple compute nodes, I/O bandwidth can become a significant bottleneck, especially when dealing with extensive data sizes that require frequent comparisons. Our method involves hashing the data chunks before comparison and using these hash values as a preliminary filter. We proceed with a full data comparison only when hashes do not match. Our hashing method is designed to be fast and capable of generating consistent hash values for floating-point numbers within a specified error bound, using a conservative truncation technique to manage variations in data. By limiting the number of full comparisons needed, our method significantly reduces unnecessary I/O, thus reducing resource utilization in large-scale processing environments.

Compact Hash Metadata using GPU-Aware Parallelized Merkle Trees. To address the challenges of managing extensive metadata generated from hashing large data chunks, we utilize a compact representation based on Merkle trees optimized for GPU acceleration. Merkle trees are effective data structures for summarizing and verifying data integrity. Each data chunk is hashed and serves as the leaves of the tree. These leaf hashes are then combined and rehashed progressively up the tree until a singular hash at the root represents the entire dataset. We effectively minimize the overhead of comparing vast datasets by employing this hierarchical hashing structure. During the checkpoint writing phase, we implement a bottom-up process to construct the Merkle tree, encapsulating all data chunks within this structured format. For comparisons, instead of beginning at the root, we start from the intermediate levels of the tree to optimize the comparison process. This method not only accelerates the detection of discrepancies by focusing on levels where mismatches are likely but also reduces the depth of tree traversal, leveraging GPU capabilities for enhanced parallel processing. By distributing these operations across multiple GPU cores, our method rapidly processes large volumes of data, far outpacing traditional CPU-based methods. The parallelization extends to the comparison phase, where GPUs efficiently identify matching regions through a single high-level hash comparison, avoiding redundant checks of identical sections. We streamline our method, and in doing so, we not only enhance computational efficiency but also reduce the storage requirements by eliminating unnecessary metadata. Large-scale applications that demand frequent data integrity checks and comparisons are the most beneficial of our solution.

Low-Latency Optimizations for Scattered I/O. To address the complexities introduced by Merkle tree hashing in our system, particularly the challenging I/O patterns resulting from many small, scattered data chunks, we have developed a set of low-latency optimizations tailored for such scenarios. While Merkle tree hashing significantly reduces the volume of I/O and the number of comparisons by identifying identical chunks across runs, it disrupts the typical I/O pattern optimized for large, contiguous operations commonly supported by PFS. The resulting scattered I/O can degrade performance due to increased latency and reduced throughput. To counter these effects, we implement an advanced I/O strategy leveraging the io_uring library [4], a modern kernel feature allowing asynchronous enqueuing of I/O operations within a single system call. Our method minimizes the latency of numerous context switches in traditional read operations. Additionally, the asynchronous nature of io_uring optimizes I/O throughput by efficiently managing scattered reads without the overhead of synchronous I/O blocking. By effectively managing the scattered I/O patterns without compromising the benefits of reduced I/O operations in the multi-level pipeline, we enhance the system's overall efficiency and reduce I/O and comparison operations.

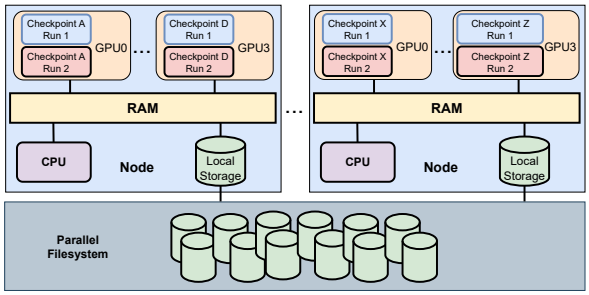


Figure 2: Architecture of a GPU-aware parallel checkpoint comparison runtime that integrates our design principles.

2.2 Architecture and High-level Overview

Our method for a scalable capture and comparison of intermediate results integrates two main components: (1) the compaction of large simulation checkpoints into metadata with a low storage footprint and (2) the comparison of two distinct checkpoint metadata to identify discrepancies. We use a series of design principles, as outlined in Section 2.1 to efficiently analyze the large number of checkpoints captured during scientific simulations. The general architecture of our method integrates the pair-wise comparison of two checkpoints (one per simulation run) on dedicated GPUs, as depicted in Figure 2, to benefit from the GPU's multi-processing capabilities. This enables the comparison of a large number of files at scale. We reduce the I/O overhead of processing large checkpoints using a hierarchical Merkle tree-based compact representation that only stores hashes of the checkpoint data. This enables low overhead tree comparison where we only need to match hash values at the same index in the trees. However, we use a conservative hashing approach leading to a few misclassifications that require accessing potentially non-contiguous regions of the original checkpoints. This type of data access patterns pressure the I/O subsystem. We mitigate the resulting I/O overhead using a multi-level streaming technique, depicted in Figure 3 to overlap I/O operations with checkpoint comparison.

2.3 Merkle-tree Compact Checkpoint Metadata

We propose using Merkle trees as metadata for representing checkpoints and an efficient parallel algorithm for identifying differences between checkpoints. Using Merkle trees, we minimize the amount of checkpoint data read from the PFS while also identifying the amount and location of data that differs between checkpoints.

The algorithm works as follows: During application execution, we construct the Merkle tree at checkpoint time on the GPU, as described in Algorithm 1, and save the metadata to the PFS. The metadata size depends on the checkpoint data length, the user-defined chunk size, and the size of a hash digest. For example, given data length N, chunk size C, and digest size D, the metadata size can be computed as $2 * D * ((N/C) - 1)$. A larger chunk size results in a smaller number of leaves and vice-versa. Merkle trees for large checkpoints can have thousands or millions of nodes. Tree construction can leverage the GPU's massive parallelism by calculating all hashes within a level of the tree in parallel. Once

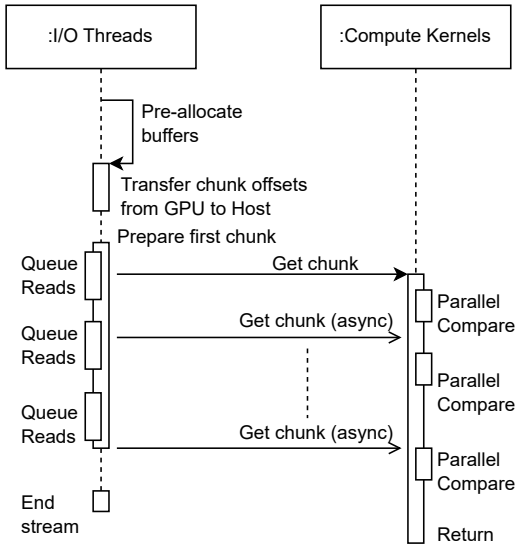


Figure 3: Asynchronous data streaming sequence diagram.

Algorithm 1 Compact Checkpoint Metadata Creation

```
function CREATE_TREE(Tree, Chunks, Leaves)
    n = |Chunks|
    for i ← 0 : n do                                   ▷ do in parallel
        C = Chunks[i]
        seed = 0
        for all block ∈ C do
            for all float ∈ block do
                new_float ← round(float)
            end for
            digest ← hash(block, seed)
            seed ← digest
        end for
        Leaves[i] ← digest
    end for
    level = 1
    for all level ∈ Tree do                            ▷ do in parallel
        if level is not Leaves then
            for all node ∈ level do
                tmp = {Tree(node, left), Tree(node, right)}
                Tree(node) ← hash(tmp)
            end for
        end if
    end for
end function
```

the checkpoints for two runs are available, we start the comparison algorithm.

The comparison algorithm is broken up into two stages. The first stage reads the previously generated metadata and uses the Merkle trees to identify any chunks that may have differences between the

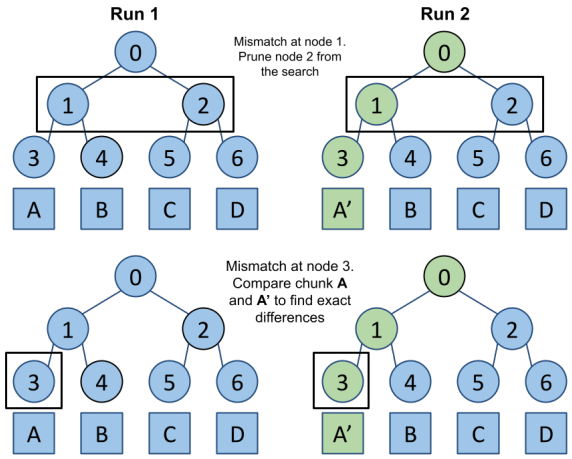


Figure 4: Identifying differences between checkpoints using Merkle trees and breadth-first search with pruning.

two runs. We use a parallel breadth-first search (BFS) to compare individual hashes and identify which chunks of data may have differences. Figure 4 shows an example of the tree comparison. Starting in the middle of the tree, we compare the corresponding nodes between the two trees and see if the hashes match. If the hashes are identical, we know all hashes within the subtree rooted at the node are identical. We then prune the subtree from the search. If the hashes differ, we know that at least one leaf in the subtree does not match, so we add the child nodes to the search. This process repeats until we run out of nodes to visit or reach the leaves. Any leaves that do not match are then added to the list of potentially changed chunks and moved on to the second stage.

The second stage takes the list of chunks and verifies whether there are any differences within each chunk. We asynchronously read and stream data chunks to overlap I/O with the comparison kernel. Figure 3 shows the comparison pipeline. A team of I/O threads reads data from the PFS into a buffer. An asynchronous transfer is initiated once the buffer is filled or all chunks have been read. The main thread retrieves the slice and signals that the buffer is free so that the I/O threads can continue reading data. The main thread then launches the kernel that compares each pair of floats to see if their difference exceeds the error bound. This continues until all chunks in the list have been compared.

2.4 Error-Bounded Checkpoint Data Hashing

We generate hash values for the leaves in our Merkle tree using an error-bounded hashing technique, highly parallelized for GPUs at two levels: (1) the entire checkpoint data is divided into chunks, and our hashing algorithm is concurrently applied on independent chunks; (2) within each chunk, we introduce a conservative rounding method, parallelized at the granularity of individual floating points. We apply the 128-bit Murmur3F hashing algorithm to compute hashes at the granularity of 128 bits. The Murmur3F algorithm offers high collision resistance according to SMHasher [1] quality tests.

We employ a conservative rounding method to ensure that each floating-point number within a chunk is accurately transformed in alignment with a predetermined error bound. We use an application-supplied absolute error that can be tolerated to assume equality and is typically known by domain experts. The rounding process involves three key steps: normalizing the numbers to a standard range, rounding them to reduce precision while maintaining necessary accuracy, and then rescaling them back to their original scale. This method effectively captures and represents variations within the specified error bound.

Next, we utilize a block-based hashing method to process the rounded data. We minimize the size of the Merkle tree by computing one hash for each chunk of the checkpoint data. To that end, the hashing of a chunk is serialized at the granularity of 128-bit blocks, where the current block is hashed using the digest of the previous block as seed. This iterative procedure continues until all data within a chunk is hashed. By leveraging the digest of one block as the seed for the next, the final hash value reflects floating-point variations within the entire chunk. Additionally, the block-based approach allows integration with any hashing algorithm, as the block size is variable and can be adjusted to fit specific application requirements.

Our method seamlessly integrates with the structured framework of a Merkle tree, enabling parallel and non-blocking updates of the tree with computed hash values. Our GPU-aware rounding and hashing methods ensure efficient utilization of GPU resources, improve the hashing throughput and enable processing large volumes of data with reduced overhead.

2.5 Implementation

Our implementation builds on flattened tree structures that enable efficient massive parallelization and io_uring for efficient asynchronous I/O. We integrate Kokkos, a performance portable runtime that can generate parallel code for both CPUs and GPUs, to support cross-platform deployment and ensure that no modification is necessary to run on CPUs.

2.5.1 Efficient Merkle Tree Creation and Traversal. We store Merkle trees as a flattened array. Merkle trees are binary trees that have clear formulas for identifying the parent and children of a node. A pointer-based tree structure is more flexible, but our use case does not dynamically change the tree. Using pointers also leads to inefficient random access patterns. Our BFS implementation is parallelized such that all nodes within the same level of the tree are processed in parallel. The only synchronization is when moving between levels. To improve performance we start the BFS in the middle of the tree where the number of the nodes in the level is greater than the number of concurrent GPU threads. Starting in the middle ensures that more threads are active rather than having them idle for the levels of the tree near the root. Using GPUs ensures that the cost of creating the Merkle tree is minimal. The small storage and creation cost of Merkle trees make it easy to add to

[1] https://github.com/aappleby/smhasher

GPU applications where we want to minimize the interruptions to the application.

2.5.2 Asynchronous Data Streaming with io_uring. Streaming the data from the PFS to the GPU allows us to run the parallel comparison kernel while the next slice of data is prepared. Working with slices of data is necessary for large checkpoints that may not fit both checkpoints in memory. We implement data streaming using the io_uring interface for asynchronous I/O and C++ threads for managing and transferring data. io_uring is a Linux kernel interface for fast asynchronous I/O. io_uring uses a submission queue and completion queue that are shared between the kernel and the application. This removes the overhead of copying data between kernel and user space. This is particularly useful for our method which issues small reads rather than reading the full file at once. We launch several I/O threads to handle issuing the read requests to io_uring. We also spawn a task that waits until all the data has been read into the slice and then asynchronously transfers the data to the device.

3 PERFORMANCE EVALUATION

3.1 Evaluation Setup

We evaluate our approach on the Polaris system at the Argonne Leadership Computing Facility (ALCF) [13], a multi-GPU computing environment with 560 compute nodes. Each node has 4 NVIDIA A100 GPUs (total of 160 GB HBM2 memory), one 2.8 GHz AMD EPYC Milan CPU (32-cores), and 512 GiB DDR4 memory. The nodes are connected using the Slingshot 11 network and can access an external 10 TB POSIX-mounted Lustre parallel file system.

3.2 Compared Approaches

3.2.1 Python-based Floating-Point Number Comparison (AllClose). We use *allclose* as a naive baseline that represents how a domain scientist may compare results. This method analyzes the results' reproducibility using the in-built *allclose* function of the NumPy package. This function takes two vectors of floating-point numbers and returns true if all the numbers are within an error bound. *allclose*, by default, checks if two arrays (a and b) are element-wise within an error tolerance. The error tolerance is calculated as the sum of the absolute (*atol*) and relative differences (*rtol * abs(b)*). However, this paper focuses on comparisons with the absolute error bound. Therefore, we define the relative error value as zero in all our experiments. *allclose* only detects any differences that exceed the error bound and is not optimized for asynchronous IO. This form of change detection is not useful for locating where changes exceed the bound. The following methods identify where in the checkpoint the different values are and use optimized IO strategies.

3.2.2 Pair-wise Floating-Point Number Comparison (Direct). This method is the most common comparison approach for reproducibility analytics that verifies whether critical data in checkpoint histories of two application runs are within an error bound defined by scientists. We refer to this baseline as *Direct*, and identify two floating-point numbers, a from any checkpoint of the first run and b from the same position in the same checkpoint of the second application run, as different if their absolute difference exceeds the error bound, i.e., $|a - b| > \epsilon$. For best performance, we implement *Direct* in C++ using Kokkos for parallelization and use io-uring, a kernel I/O interface that provides efficient asynchronous I/O operations and bypasses much of the system call overhead.

3.2.3 Our Method. We compare the aforementioned numerical reproducibility analysis methods with our proposed hierarchical hashing-based solution described in Section 2.

3.3 Evaluation Methodology

3.3.1 Application Checkpoints. We consider the Hardware/Hybrid Accelerated Cosmology Code (HACC) [15] as a practical HPC application for our evaluation. HACC is an extreme-scale cosmological simulation suite originally developed for the heterogeneous architecture of the first petascale supercomputer, Roadrunner [16], and later adjusted for deployment at scale on more recent supercomputers, including Polaris. For our evaluation, we simulated direct particle-particle interactions using the particle-particle-particle-mesh method, i.e., P^3M algorithm [17] over 50 iterations. We asynchronously capture particle data (coordinates, velocity, and gravitational potential described in Table 1) using the VELOC checkpointing library [33] at iterations 10, 20, 30, and 40 of each simulation. The simulations are conducted using varying numbers of particles yielding different checkpoint sizes, as summarized in Table 1.

Table 1: Content of HACC checkpoints.

Field	Type	Description
X	F32	x coordinate
Y	F32	y coordinate
Z	F32	z coordinate
VX	F32	x velocity
VY	F32	y velocity
VZ	F32	z velocity
ϕ	F32	grav. potential

#Particles	#Nodes	Chkpt Size
0.5 B	2	14 GB
1 B	2	28 GB
2 B	2	56 GB
17 B	128	563 GB

3.3.2 Performance Metrics. We measure the effectiveness of our approach using three key metrics. We design the metrics to assess our method's accuracy, efficiency, and performance compared to the traditional direct approach. Below, we detail each metric:

- Effectiveness: We assess the accuracy of our method by comparing it to reference results from the direct approach. This metric focuses on the number of differences identified across an entire dataset during a checkpoint comparison.
- Time: We analyze the efficiency of using hashing for reproducibility by comparing the time it takes for our method to analyze hashes and floating-point numbers (as outlined in Section 2) against the time taken by the direct approach.
- Throughput: We measure the total volume of data analyzed per second over the analysis duration. This metric is the ratio between the size of the compared data and the time taken to read checkpoint data from the parallel file system (PFS), the data streaming process to the GPU, and the comparison of hashes and floating-point numbers.

3.3.3 Testing Setting. To comprehensively evaluate our method, we varied several experimental parameters to test its robustness and performance across different conditions. The setting used in our experiments is detailed in Table 2 and includes:

- **Number of Nodes**: To understand how our method scales with increasing nodes, we use a variable number of nodes ranging from 1 to 32.
- **Error Bounds**: Error bound defines the acceptable difference threshold between two checkpoints. Lower error bounds provide stricter criteria for identifying differences, while higher bounds allow for more variation. We test with error bounds set to 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, and 10^{-7}.
- **Chunk Sizes**: The size of the data chunks used for hashing and comparison affects the precision and overhead of identifying differences. Smaller chunk sizes provide more precise identification but may increase overhead due to reading unnecessary data. We consider chunk sizes ranging from 4 KB to 512 KB. Therefore, with 4 KB chunks and 16-byte hash digests, the metadata size for a 7 GB checkpoint (size of the aggregated checkpoint on one node for the HACC experiment with 0.5×10^9 particles) is $\approx 55MB$.

Table 2: Setup used to evaluate performance and scalability.

Description	Values
Number of Nodes	1, 2, 4, 8, 16, 32
Error bounds	10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}
Chunk sizes	4 KB - 512 KB

3.3.4 Experimental Scenarios. We present two scenarios that examine the impact of different parameters on our method in contrast with existing methods. The first scenario studies how chunk size and the error bound affect the comparison throughput of our method. We use the 0.5, 1, and 2 billion particle checkpoints and use two nodes to compare all pairs of checkpoints in parallel. The second scenario is a strong scaling study comparing our method with the optimized direct method. We use the checkpoints generated by the 17 billion particle simulation and vary the number of nodes from 4 to 32 nodes with 4 processes per node. For each node configuration, we compare all 128 pairs of checkpoints. For both scenarios, we use "vmtouch -e" to evict the pages corresponding to the input files from the file system cache in order to enable a fair comparison where each approach starts with a cold cache. Internally, vmtouch uses POSIX_FADV_DONTNEED to clear the cache. This is necessary to ensure that the page cache does not skew our performance metrics.

3.4 Performance Results

3.4.1 Comparison Throughput. We summarize the comparison throughput for the AllClose baseline, optimized direct comparison, and our Merkle tree-based method, across three problem sizes in Figure 5. The baseline method throughput is at most 2.67 GB/s regardless of the error bound. The direct method uses io_uring for more efficient I/O which increases the throughput to at most 5.24 GB/s. We note that varying the error bound does not impact either the baseline or the direct comparison throughput, as all data must be compared regardless of the error bound. Thanks to our design principles and optimization, our method consistently outperforms the baseline and direct comparison methods for all tested chunk sizes and error bounds.

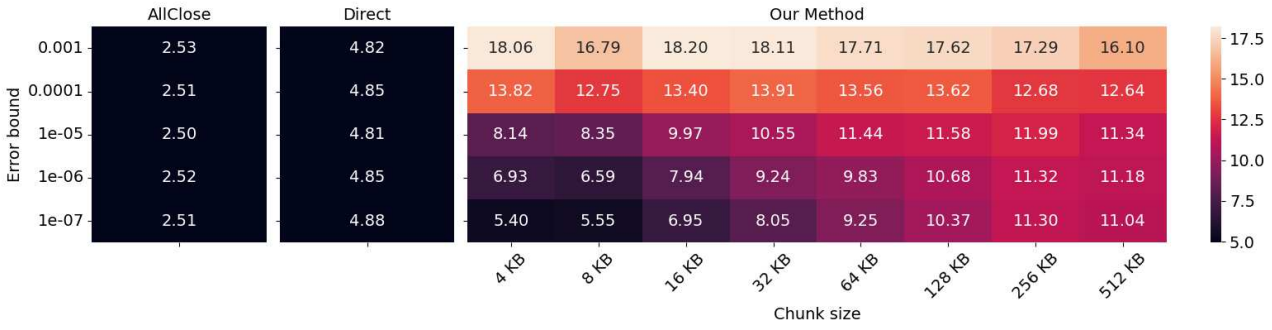
The choice of the chunk size poses an interesting trade-off for our method. Smaller chunks can more accurately determine where differences are located. This reduces the amount of unnecessary I/O, which is why using 4 KB chunks yields the best performance with high error bounds. However, we cannot select the smallest chunk size for all situations. Choosing small chunks for small error bounds such as 10^{-7} significantly lowers throughput. Reading small chunks that are scattered randomly across the file is an extremely challenging I/O pattern that degrades performance. The small error bound increases the number of changed chunks which worsens the impact of the poor I/O pattern. For situations where large amounts of data could exceed the error bound, it is better to improve the I/O pattern by reading larger chunks of data. Increasing the chunk size from 4 KB to 512 KB almost doubles the throughput for all checkpoint sizes. Even so, the chunk size cannot be too large or the performance may degrade from reading too much unnecessary data. This is shown in Figure 5a where for an error bound of 10^{-7}, increasing the chunk size from 256 KB to 512 KB lowers throughput from 11.3GB/s to 11.04GB/s.

As the error bound grows, the throughput increases for all chunk sizes across the three simulation sizes. Larger error bounds lead to fewer changes that exceed the threshold, reducing the amount of data read from the PFS. Our comparison method is up to 11 times faster than the direct comparison method. This is especially true for larger checkpoints and error bounds. The throughput for larger error bounds such as 0.001 nearly doubles as the checkpoint size increases. This indicates that the runtime stays nearly constant as the size of checkpoints increases.

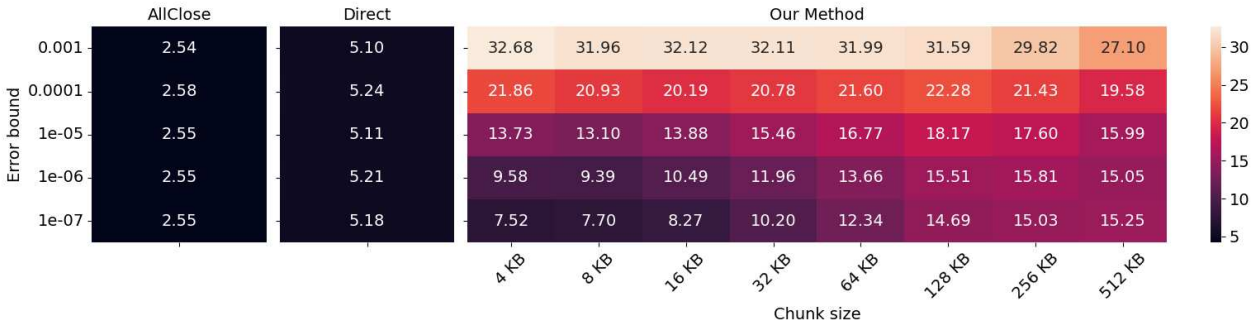
The trade-offs between I/O patterns and reducing the amount of data read from the PFS make chunk size selection vital for high performance. Throughput behavior is consistent between the three checkpoint sizes. This suggests that the optimal choice of error bound and chunk size for checkpoints from a small-scale problem will also achieve high throughput for larger simulations.

3.4.2 Comparison Cost Breakdown. To identify the main bottleneck and better understand the I/O pattern and data size trade-off, we look at a breakdown of the comparison runtime for two cases. Figure 6 shows the individual timers for the comparison process. The first case shown in Figure 6a, is when the error bound is low and the second case is when the error bound is high. Of the five timers, the time spent on deserializing and comparing the Merkle trees is negligible compared to the rest of the timers. The setup section is for allocating buffers and data structures which is why the time spent is very consistent. Reading the Merkle-tree metadata is very cheap for reasonable chunk sizes. The time spent in the verification phase (Compare direct time) identifying changes in the data is the most important. For small error bounds, we need to load more data which is why the verification time is dominant. The verification time decreases as chunk size increases due to the better I/O pattern. However, the performance improvements level off as the chunks approach 1 MB.

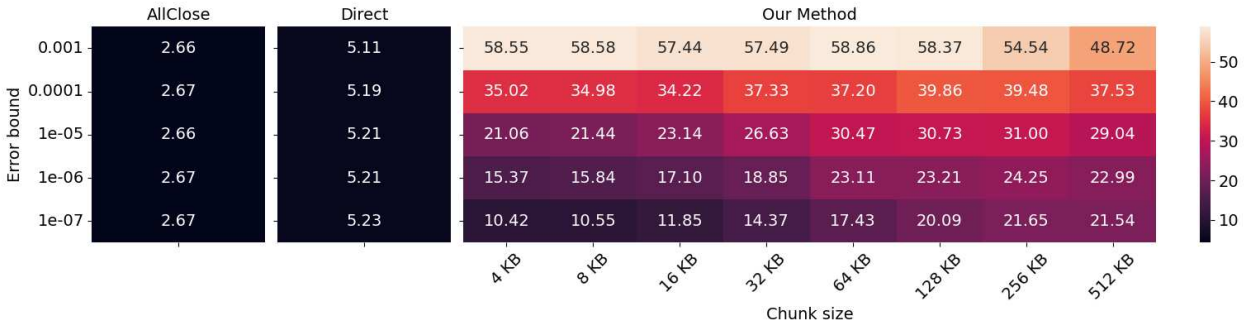
Reducing the amount of data read can overcome the poor random small reads I/O pattern. Figure 6b shows that the setup time is similar to when the small error bound case. The total runtime is shorter and does not change as much when the chunk size varies. This indicates that the number of identified chunks is very small. As the chunk size grows, the verification phase runtime increases due to reading more unnecessary data. The read time also decreases



(a) 500 Million particles (7GB per checkpoint)



(b) 1 Billion particles (14GB per checkpoint)

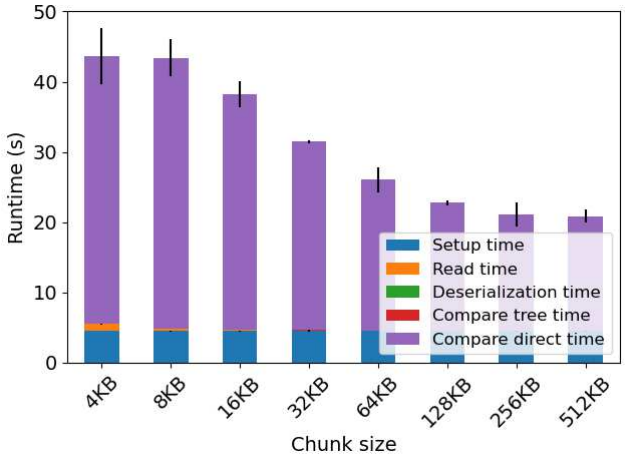


(c) 2 Billion particles (28GB per checkpoint)

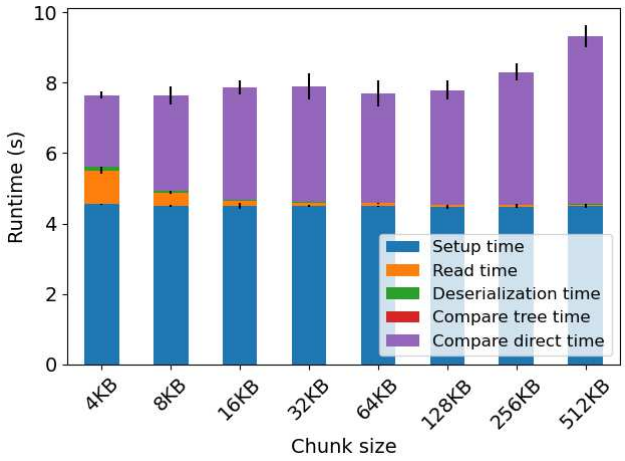
Figure 5: Comparison of our approach vs. value-by-value comparison. The direct approach streams data from the PFS to the GPU. Throughput is measured as the amount of checkpoint data over the total runtime.

since larger chunks result in fewer hashes and smaller trees. For small error bounds with many changes, it is best to choose a large chunk size. Small chunk sizes are best in cases where there are fewer changes. This is emphasized in Figure 6a vs. Figure 6b. The trade-off between the number of identified chunks, error bound, and the amount of data is complex, which is why we study it next in greater detail.

3.4.3 Effectiveness of Error-Bounded Data Hashing. To evaluate our error-bounded hash function, we measure the percentage of checkpoint data loaded for further comparison vs. the false positive rate of the error-bounded hash function. Figure 7 summarizes the impact of error bound and chunk size on the effectiveness of the error-bounded hash function. Figure 7a shows that increasing the chunk size also increases the percentage of data that must be read. The percentage increase does not scale linearly with the chunk size. Using 8 KB chunks instead of 4 KB chunks increases the percentage by less than 5%. The nonlinear increase is because of how the changes are distributed. If two contiguous 4 KB chunks are marked as changed then increasing the chunk size to 8 KB will result in the same amount of data being marked. As the chunk size continues to grow, the percentage increase also grows as more unnecessary data is marked as changed. Increasing the error bound has a larger impact on the percentage of data read from the PFS. Increasing the error bound from 10^{-5} to 10^{-4} results in a 9.7% increase which has more impact than the 9% increase when going from 128 KB to 4 KB chunks. This pattern is also shown in Figure 5c where the throughput increases by 13.96 GB/s and 9.67 GB/s respectively. Figure 7b

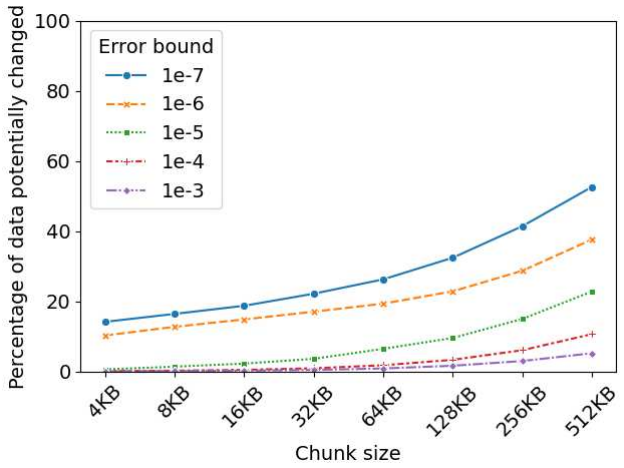


(a) Runtime breakdown with an error bound of 10^{-7}

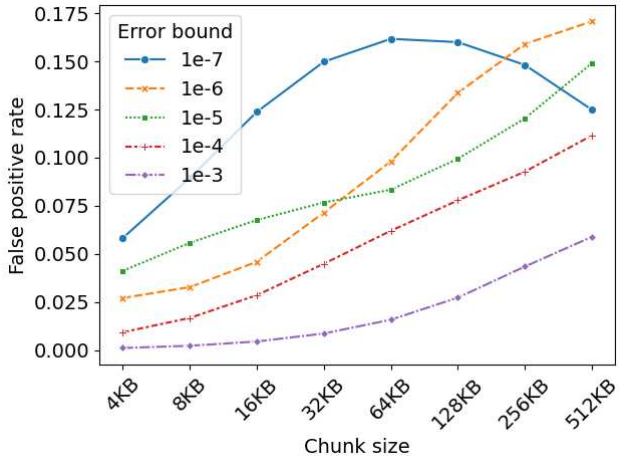


(b) Runtime breakdown with an error bound of 10^{-3}

Figure 6: Impact of error bound and chunk size on the comparison runtime. Total runtime is split into separate timers for each part of the comparison process. Error bounds of 10^{-7} and 10^{-3}. Chunk size varies from 4 KB to 512 KB.



(a) Percentage of the checkpoint data marked as potentially changed.



(b) False positive rate

Figure 7: Effectiveness of the error-bounded hash function. Checkpoints are from the 2 billion particle simulation.

presents the false positive rate for the conservative error-bounded hash function. The hash function correctly identifies all chunks that contain changes that exceed the error bound. However, the hash also has false positives which result in more unnecessary data being streamed from the PFS. Except for the error bound of 10^{-7}, increasing the chunk size leads to more false positives. The false positive rate drop for larger chunks with an error bound of 10^{-7} has surprisingly little effect on the percentage of data changed. This is because rate drop has less of an effect on the total percentage of data compared to the doubling of the chunk size. These results show that the error-bounded hash function is most effective with small chunk sizes. Unfortunately, the poor I/O access pattern negates the benefits.

The ideal case for our method is when there are no changes. In this situation, we can use the metadata to verify that there are no changes that exceed the threshold without needing to read any checkpoint data. This property makes our method particularly well-suited for studying reproducibility. Reproducible applications will have a clearly defined error bound such that there are no run-to-run differences that exceed the threshold. This makes our checkpoint comparison method an excellent tool for enhancing reproducibility in HPC applications thanks to the low storage costs for metadata and the high comparison throughput.

3.4.4 Cost of Constructing Merkle Trees. Our Merkle tree implementation uses Kokkos for parallelization on multiple architectures and is optimized for GPUs. We evaluate the benefits of our GPU-optimized Merkle tree implementation by comparing the tree construction time on the CPU and GPU in Figure 8. Tree construction on GPUs is four orders of magnitude faster than the CPU thanks to the higher bandwidth and computing resources. Chunk size does not affect runtime because the same amount of data is being hashed

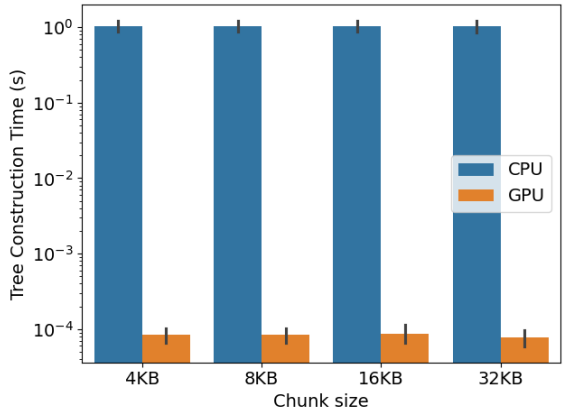


Figure 8: Tree construction cost (500 million particles) using the CPU or GPU. Error tolerance is 10^{-7}. Y-axis is log-scale.

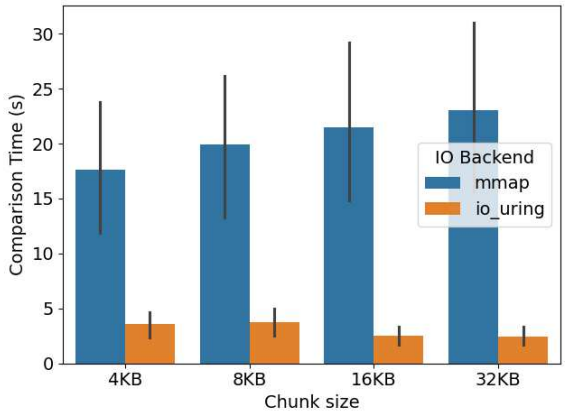
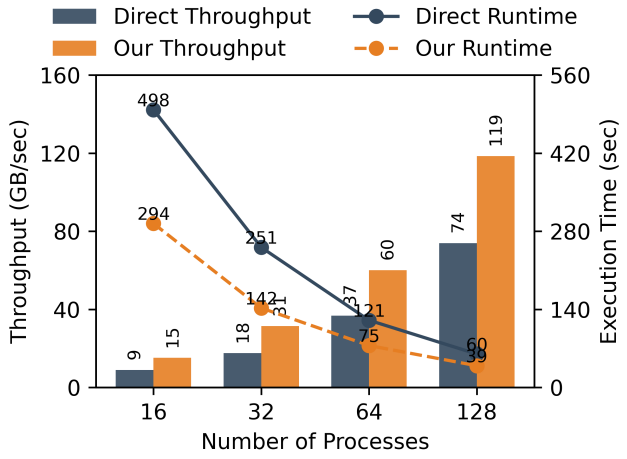
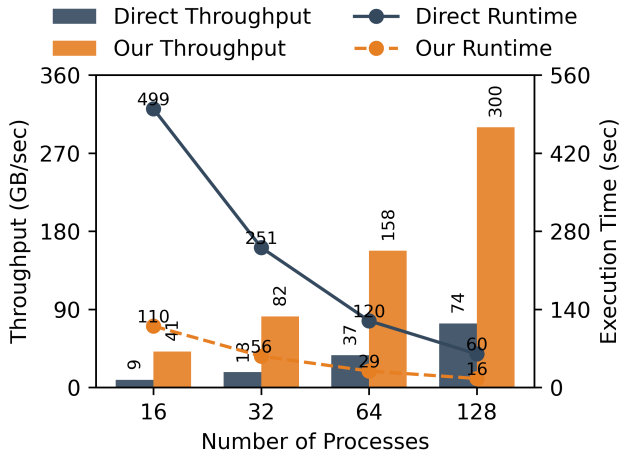


Figure 9: Comparison of I/O backends for scattered I/O (500 million particles). Error tolerance is 10^{-7}.



(a) Error Bound = 10^{-7}



(b) Error Bound = 10^{-3}

Figure 10: Throughput of comparing 1024 checkpoints for an increasing number of processes (four per node). Higher is better. Both approaches show near-perfect scalability and our method maintains its higher throughput and lower runtime across all settings.

regardless of chunk size. Our GPU-optimized Merkle tree construction algorithm has minimal overhead and can be easily integrated with checkpointing runtimes and data analysis tools. The low cost of tree construction can potentially be used to determine when to take checkpoints or perform more costly analyses.

3.4.5 Enhancing Scattered I/O. The choice of I/O backend is important for efficient comparisons. We compare the runtime performance between the *mmap* and *io_uring* backends in Figure 9 using eight processes. Using *io_uring* is over three times faster than *mmap* and demonstrates less variance. *io_uring* is an asynchronous API that allows queuing and submitting multiple independent read operations with very few kernel calls. *mmap* performance suffers because the I/O operations are synchronous and trigger numerous expensive page faults. The *mmap* backend scales with the amount of data that is being read. *io_uring* is less affected by the quantity of data and even shows improvements when the chunk size increases.

3.4.6 Scalability Study. Our last set of experiments is a strong scaling study of our method and the direct comparison approach. We analyze HACC checkpoints captured on 128 nodes (a total of 512 checkpoints per simulation run) and experiment with low (10^{-7}) and high (10^{-3}) error bounds to evaluate the performance of our method at scale in two scenarios: (1) using a low error bound to illustrate a worst-case scenario with more I/O operation during the second phase of our method; (2) using a high error bound to capture the best-case scenario with a minimum number of I/O requests. Figure 10 presents the per-process comparison throughput for an increasing number of processes (four processes per node). As can be observed, both approaches scale with the number of processes maintaining an average speedup of 1.9× for every increment in the number of processes. Our method maintains a higher throughput than the direct comparison for both scenarios. This is an important

observation that highlights the benefits of our low-latency optimization for scattered I/O. As a result, with 50% less value-by-value comparisons than the direct comparison approach, our method maintains a minimum throughput and runtime speedup of 1.6× at scale when the error bound is 10^{-7}. Despite performing more value-by-value comparisons compared to when the error tolerance is higher, our method maintains better performance than the direct comparison approach, as depicted in Figure 10a. This performance shows that our optimizations efficiently manage scattered I/O, minimizing the overhead of reading chunks from non-contiguous offsets of checkpoint files located on a PFS. A similar trend is visible for a higher error bound with a smaller I/O overhead, yielding up to 4.6× speedup compared to the direct comparison approach.

4 RELATED WORK

The scientific integrity and transparency of HPC workflows are defined by the ability of scientists to reproduce the results and performance of an application when executed multiple times on the same computational platform using the same code, input parameters, and datasets [27, 35, 42]. Several recent studies highlight the importance of reproducibility in computational workflows [25, 32, 39], identifying various sources of performance and results variations including shared memory bandwidth contention [3], non-associative floating-point operations [22], dynamic scheduling of parallel processes [1], variability in network bandwidth [43], etc. Preemptive solutions, e.g., packaging experiments [20], sandbox computational environments [9, 28] and workflow management systems [12] improve reproducibility by preventing interference from external processes and enabling workflow replication. Post-simulation analysis frameworks further reinforce the state of practice in performance reproducibility through queryable systems for performance metrics and workflow provenance analysis [34]. Although these solutions contribute to stability in computational experiments, the increasing scale of HPC workflows and existing non-determinism sources within a single workflow highlight the importance of further investigating the reproducibility of computational results.

Existing studies on results reproducibility explore strategies to improve numerical correctness and convergence by reducing numerical roundoff errors introduced with floating-point arithmetic, e.g., error-free transformation for reproducible summation [24, 29]. These solutions focus on ensuring bitwise identical floating-point results but do not account for errors induced by runtime variations due to I/O patterns (common I/O operations of flushing large files to the PFS may create interleaves that introduce varying errors in intermediate results) or silent errors occurring during execution. Error detection techniques, e.g., checksums, mitigate such issues but can also become a source for non-determinism if obtained using non-associative operations. Scientific workflows primarily operate on floating-point numbers and results are often validated if the difference between two simulation runs is within an acceptable error bound. However, critical applications, e.g., drug or nuclear reactor design, may require bitwise reproducibility achievable at the cost of computational performance using sequential execution and fixed-order arithmetic operations [5]. To mitigate the waste of computational resources by waiting until the final outputs of two distinct runs are captured, a detailed analysis of intermediate results can identify the exact process or simulation stage where results start diverging [2]. Hash-based de-duplication techniques for binary data have been used in a variety of storage scenarios to save space [21, 30, 31]. Furthermore, Merkle trees are commonly used to check integrity in protocols such as BitTorrent or databases such as Cassandra [10], Dynamo[38], and Riak [41]. Our approach is unique in that it focuses on reproducibility, the goal being to be able to compare two scientific datasets (usually consisting of floating point numbers) very fast.

5 CONCLUSIONS

This paper presents a scalable method for capturing and comparing intermediate multi-run results for enhancing reproducibility in HPC applications. To this end, we use Merkle trees as a compact metadata representation of checkpoints and use the tree structure in addition to I/O pipelining, error-bounded hash techniques, and optimized low-latency scattered I/O to accelerate checkpoint comparison. We use these key ideas to improve comparison throughput by up to an order of magnitude over the optimized direct comparison method. We highlight the trade-off between the I/O volume from the PFS and the efficiency of the I/O pattern. Our method shows near-perfect strong scaling and achieves 300 GB/s comparison throughput.

We plan to investigate multi-node parallel online checkpoint compaction and comparison. Our method reduces the I/O overhead from loading all checkpoints from the PFS. Online checkpoint comparison can further reduce the I/O overhead since only the previous checkpoint history needs to be read from the PFS. We can also compact the checkpoints online to reduce the I/O overhead and storage costs for the checkpoint history. Our method also shows promise as a potential continuous integration tool. Applications with a defined error bound can save a Merkle tree for the expected results of a test. If the method detects any differences then the developers know that the code change may introduce a reproducibility issue.

6 ACKNOWLEDGMENTS

This material is based upon work supported by: the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357; the National Science Foundation under Grants #1900888, #1900765, #2223704, #2331152, #2411386, #2411387, #2106635.

REFERENCES

[1] Peter Ahrens, James Demmel, and Hong Diep Nguyen. 2020. Algorithms for Efficient Reproducible Floating Point Summation. *TOMS'20: ACM Transactions on Mathematical Software* 46, 3 (2020), 1–49.

[2] Kevin Assogba, Bogdan Nicolae, Hubertus Van Dam, and M. Mustafa Rafique. 2023. Asynchronous Multi-Level Checkpointing: An Enabler of Reproducibility using Checkpoint History Analytics. In *SC'23: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Association for Computing Machinery, New York, NY, USA, 1748–1756.

[3] Emre Ates, Yijia Zhang, Burak Aksar, Jim Brandt, Vitus J. Leung, Manuel Egele, and Ayse K. Coskun. 2019. HPAS: An HPC Performance Anomaly Suite for Reproducing Performance Variations. In *ICPP'19: The Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan). Association for Computing Machinery, New York, NY, USA, Article 40, 10 pages.

[4] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf

[5] Pavan Balaji and Dries Kimpe. 2013. On the Reproducibility of MPI Reduction Operations. In *HPCC'13: The IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on*

Embedded and Ubiquitous Computing. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 407–414.

[6] Marek Baranowski, Braden Caywood, Hannah Eyre, Janaan Lake, Kevin Parker, Kincaid Savoie, Hari Sundar, and Mary Hall. 2017. Reproducing ParConnect for SC16. *Parallel Computing* 70 (2017), 18–21.

[7] Magnus Borga, André Ahlgren, Thobias Romu, Per Widholm, Olof Dahlqvist Leinhard, and Janne West. 2020. Reproducibility and Repeatability of MRI-based Body Composition Analysis. *Magnetic Resonance in Medicine* 84, 6 (2020), 3146–3156.

[8] Greg L Bryan, Michael L Norman, Brian W O'Shea, Tom Abel, John H Wise, Matthew J Turk, Daniel R Reynolds, David C Collins, Peng Wang, Samuel W Skillman, et al. 2014. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement Series* 211, 2 (2014), 19.

[9] R. Shane Canon. 2020. The Role of Containers in Reproducibility. In *CANOPIE-HPC'20: The Proceedings of the 2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC*. IEEE Computer Society, Los Alamitos, CA, USA, 19–25.

[10] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. 2015. A Big Data Modeling Methodology for Apache Cassandra. In *BigData'15: 2015 IEEE International Congress on Big Data*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 238–245.

[11] Peter V Coveney, Derek Groen, and Alfons G Hoekstra. 2021. Reliability and Reproducibility in Computational Science: Implementing Validation, Verification and Uncertainty Quantification in silico. *Philosophical Transactions of the Royal Society A* 379, 2197 (2021), 20200409.

[12] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow Enables Reproducible Computational Workflows. *Nature Biotechnology* 35, 4 (2017), 316–319.

[13] Argonne Leadership Computing Facility. n.d.. Polaris. https://www.alcf.anl.gov/polaris. Accessed: May 24, 2024.

[14] Mikaila J. Gossman, Bogdan Nicolae, and Jon C. Calhoun. 2024. Scalable I/O Aggregation for Asynchronous Multi-level Checkpointing. *Future Generation Computer Systems* 160 (2024), 420–432.

[15] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Venkatram Vishwanath, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, and Zarija Lukić. 2016. HACC: Extreme Scaling and Performance Across Diverse Architectures. *Communications of the ACM* 60, 1 (dec 2016), 97–104.

[16] Salman Habib, Adrian Pope, Zarija Lukić, David Daniel, Patricia Fasel, Nehal Desai, Katrin Heitmann, Chung-Hsing Hsu, Lee Ankeny, Graham Mark, Suman Bhattacharya, and James Ahrens. 2009. Hybrid Petacomputing Meets Cosmology: The Roadrunner Universe Project. *Journal of Physics: Conference Series* 180, 1 (jul 2009), 012019.

[17] R. W. Hockney and J. W. Eastwood. 1988. *Computer Simulation Using Particles*. Taylor & Francis Group, New York, NY, USA.

[18] Bin Hu, Shane Canon, Emiley A Eloe-Fadrosh, Michal Babinski, Yuri Corilo, Karen Davenport, William D Duncan, Kjiersten Fagnan, Mark Flynn, Brian Foster, et al. 2022. Challenges in Bioinformatics Workflows for Processing Microbiome Omics Data at Scale. *Frontiers in Bioinformatics* 1 (2022), 826370.

[19] Jie Jia, Yi Liu, Yanke Liu, Yifan Chen, and Fang Lin. 2024. AdapCK: Optimizing I/O for Checkpointing on Large-Scale High Performance Computing Systems. In *Euro-Par'24: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26–30, 2024, Proceedings, Part III* (Madrid, Spain). Springer-Verlag, Berlin, Heidelberg, 342–355.

[20] Kate Keahey, Jason Anderson, Mark Powers, and Adam Cooper. 2023. Three Pillars of Practical Reproducibility. In *eScience'23: The IEEE 19th International Conference on e-Science*. IEEE Computer Society, Los Alamitos, CA, USA, 1–6.

[21] Andrzej Kochut, Alexei Karve, and Bogdan Nicolae. 2015. Towards Efficient On-demand VM Provisioning: Study of VM Runtime I/O Access Patterns to Shared Image Content. In *IM'15: 13th IFIP/IEEE International Symposium on Integrated Network Management*. Ottawa, Canada, 321–329.

[22] Ignacio Laguna. 2020. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. In *IPDPS'20: The IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 622–633.

[23] Philippe Langlois, Rafife Nheili, and Christophe Denis. 2016. Recovering Numerical Reproducibility in Hydrodynamic Simulations. In *ARITH'16: The IEEE 23nd Symposium on Computer Arithmetic*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 63–70.

[24] Kuan Li, Kang He, Stef Graillat, Hao Jiang, Tongxiang Gu, and Jie Liu. 2023. Multi-level Parallel Multi-layer Block Reproducible Summation Algorithm. *Parallel Computing* 115 (2023), 102996.

[25] Xin Liu, JD Emberson, Michael Buehlmann, Nicholas Frontiere, and Salman Habib. 2023. Numerical Discreteness Errors in Multispecies Cosmological N-body Simulations. *Monthly Notices of the Royal Astronomical Society* 522, 3 (2023), 3631–3647.

[26] Avinash Maurya, M. Mustafa Rafique, Thierry Tonellot, Hussain J. AlSalem, Franck Cappello, and Bogdan Nicolae. 2023. GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching. In *HPDC'23: The Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, 73–85.

[27] Robert D. McIntosh and Christopher D. Chambers. 2020. The Three R's of Scientific Integrity: Replicability, Reproducibility, and Robustness. *Cortex* 129 (2020), A4–A7.

[28] David Moreau, Kristina Wiebels, and Carl Boettiger. 2023. Containers for Computational Reproducibility. *Nature Reviews Methods Primers* 3, 1 (2023), 50.

[29] Ingo Müller, Andrea Arteaga, Torsten Hoefler, and Gustavo Alonso. 2018. Reproducible Floating-Point Aggregation in RDBMSs. In *ICDE'18: Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 1049–1060.

[30] Bogdan Nicolae. 2013. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In *IPDPS'13: The 27th IEEE International Parallel and Distributed Processing Symposium*. Boston, USA, 19–28.

[31] Bogdan Nicolae. 2015. Leveraging Naturally Distributed Data Redundancy to Reduce Collective I/O Replication Overhead. In *IPDPS'15: 29th IEEE International Parallel and Distributed Processing Symposium*. Hyderabad, India, 1023–1032.

[32] Bogdan Nicolae, Tanzima Z. Islam, Robert Ross, Huub Van Dam, Kevin Assogba, Polina Shpilker, Mikhail Titov, Matteo Turilli, Tianle Wang, Ozgur O. Kilic, Shantenu Jha, and Line C. Pouchard. 2023. Building the I (Interoperability) of FAIR for Performance Reproducibility of Large-Scale Composable Workflows in RECUP. In *eScience'23: The IEEE 19th International Conference on e-Science*. IEEE Computer Society, Los Alamitos, CA, USA, 1–7.

[33] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *IPDPS'19: The Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 911–920.

[34] Line Pouchard, Sterling Baldwin, Todd Elsethagen, Shantenu Jha, Bibi Raju, Eric Stephan, Li Tang, and Kerstin Kleese Van Dam. 2019. Computational Reproducibility of Scientific Workflows at Extreme Scales. *IJHPCA'19: The International Journal of High Performance Computing Applications* 33, 5 (2019), 763–776.

[35] Jan Provazník, Radim Filip, and Petr Marek. 2022. Taming Numerical Errors in Simulations of Continuous Variable Non-Gaussian State Preparation. *Scientific Reports* 12, 1 (2022), 16574.

[36] Kento Sato, Ignacio Laguna, Gregory L Lee, Martin Schulz, Christopher M Chambreau, Simone Atzeni, Michael Bentley, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Geof Sawaya, et al. 2019. PRUNERS: Providing Reproducibility for Uncovering Non-deterministic Errors in Runs on Supercomputers. *IJHPCA'19: The International Journal of High Performance Computing Applications* 33, 5 (2019), 777–783.

[37] Geof Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H Ahn. 2017. FLiT: Cross-platform Floating-point Result-consistency Tester and Workload. In *IISWC'17: The IEEE International Symposium on Workload Characterization*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 229–238.

[38] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD/PODS'12: International Conference on Management of Data* (Scottsdale, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 729–730.

[39] Victoria Stodden and Matthew S Krafczyk. 2018. Assessing Reproducibility: An Astrophysical Example of Computational Uncertainty in the HPC Context. *ResCuE-HPC'18: The 1st Workshop on Reproducible, Customizable and Portable Workflows for HPC at SC'18*.

[40] Michela Taufer, Omar Padron, Philip Saponaro, and Sandeep Patel. 2010. Improving Numerical Reproducibility and Stability in Large-scale Numerical Simulations on GPUs. In *IPDPS'10: The IEEE International Symposium on Parallel & Distributed Processing*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 1–9.

[41] Basho Technologies. 2009. Riak. https://www.riak.com/.

[42] Krishna Tiwari, Sarubini Kananathan, Matthew G Roberts, Johannes P Meyer, Mohammad Umer Sharif Shohan, Ashley Xavier, Matthieu Maire, Ahmad Zyoud, Jinghao Men, Szeyi Ng, et al. 2021. Reproducibility in Systems Biology Modelling. *Molecular Systems Biology* 17, 2 (2021), e9982.

[43] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is Big Data Performance Reproducible in Modern Cloud Networks?. In *NSDI'20: The Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA). USENIX Association, USA, 513–528.

[44] GR Williams, GP Behm, T Nguyen, A Esparza, VG Haka, A Ramos, B Wright, JC Otto, CP Paolini, and MP Thomas. 2017. SC16 Student Cluster Competition Challenge: Investigating the Reproducibility of Results for the ParConnect Application. *Parallel Computing* 70 (2017), 27–34.