Optimizing I/O Performance of Parallel Iterated Relational Algebra Applications

Ramlah Ilyas
ilyas.22@osu.edu
The Ohio State University
Columbus, Ohio, USA

Sidharth Kumar
sidharth@uic.edu
University of Illinois Chicago
Chicago, IL, USA

Thomas Gilray
thomas.gilray@wsu.edu
Washington State University
Pullman, WA, USA

Kristopher Micinski
kkmicins@syr.edu
Syracuse University
Syracuse, NY, USA

Suren Byna
byna.1@osu.edu
The Ohio State University
Columbus, Ohio, USA

Abstract

Fixed-point iteration over parallel relational algebra (RA) operators underpins a broad class of symbolic AI and recursive analytic workloads, including graph reachability, points-to analysis, and rule-based inference. At scale, each iteration materializes substantial intermediate and output relations across distributed memory, causing persistence I/O to become a dominant contributor to end-to-end cost. This I/O is inherently write-dominated: every iteration emits a delta of newly derived tuples, while the runtime must periodically checkpoint accumulated relation state to stable storage for fault tolerance and out-of-core execution.

This paper presents a case study in optimizing checkpoints for parallel iterated relational algebra, comparing an existing relation-dumping pipeline against a structured, performance-tuned alternative. The baseline stores relations as opaque binary dumps via POSIX positioned writes or collective MPI-IO with minimal file-system hinting. We extend this pipeline with a parallel HDF5 backend that stores relations as schema-aware datasets and writes rank partitions using *HDF5 hyperslabs* with collective transfers. To avoid the performance cliffs associated with default parallel HDF5 settings, we apply a systematic size-aware tuning policy that configures both HDF5 property lists and MPI-IO hints. The optimized pathway uses a two-phase workflow that creates the file and dataset first and then performs the parallel write. It also applies output-regime-dependent alignment and buffering, configures metadata caching and sieve buffering, selects an appropriate dataset layout, and enforces collective dataset transfers. Across final write sizes from 15 GB to 30 GB, the tuned HDF5 path consistently outperforms the MPI-IO baselines. At higher MPI process counts, it is often 35%-58% faster while producing self-describing outputs.

CCS Concepts: • **Computing methodologies** → *Parallel algorithms*; • **Information systems** → **Query optimization**.

Keywords: relational algebra, Datalog, HDF5, MPI-IO, parallel I/O, checkpointing, I/O optimization

ACM Reference Format:
Ramlah Ilyas, Sidharth Kumar, Thomas Gilray, Kristopher Micinski, and Suren Byna. 2026. Optimizing I/O Performance of Parallel Iterated Relational Algebra Applications. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3805687.3806261

1 Introduction

HPC applications scale to thousands of processes, but this often amplifies bottlenecks in parallel file I/O. At scale, performance depends not only on volume but also on access granularity, alignment and striping effects, MPI-IO buffering/aggregation, and metadata overhead [6, 12, 22]. These sensitivities have motivated extensive HPC I/O research and the development of tools that measure and diagnose real application and storage interactions [6, 22].

In this paper, we study parallel I/O for *iterated relational algebra*, a computation model that underpins many symbolic AI and recursive analytic workloads. Relational algebra (RA) provides a small but expressive basis of operators (e.g., join, projection, union, selection) that, when applied iteratively until a fixed point is reached, are sufficient to express a rich space of recursive computations and deduplication in the pipeline maintain set semantics [1, 13]. Classic examples include graph mining and reachability, knowledge representation and reasoning, and program analysis (e.g., information-flow or points-to analysis), all of which follow a common pattern of repeated operator application over monotonically growing relations [13]. Because these computations

CHEOPS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2604-0/2026/04
https://doi.org/10.1145/3805687.3806261

are inherently iterative and can materialize large intermediate relations at each iteration boundary, checkpoint and restart mechanisms cease to be optional conveniences and become operational necessities for correctness, fault tolerance, and out-of-core scalability [20].

Iterated RA workloads exhibit fundamentally different data representations and access patterns compared to traditional scientific simulations. Simulation codes typically operate over structured grids and produce array-based outputs, whose in-memory layout maps almost directly onto the desired on-disk representation, enabling efficient bulk transfers with minimal reorganization. Relational workloads, by contrast, operate over collections of tuples organized in access-oriented in-memory structures, such as hash indices or trie-based organizations keyed on frequently joined or queried columns, that are optimized for the join, membership test, and duplicate elimination operations central to fixed-point evaluation [1, 13]. These in-memory representations bear little resemblance to a storage-friendly on-disk layout. As a result, storing RA data requires an explicit *packing phase* in which the runtime traverses its index structures and serializes tuples into a contiguous, transfer-ready representation. This reorganization step is analogous to a well-known performance lever in scalable parallel I/O: aggregating and restructuring data prior to writing data in order to promote large, aligned, sequential transfers and minimize small-request and metadata overheads [12, 14]. At large scales, this packing cost can become dominant, and it also shapes the eventual I/O pattern because it determines whether the write phase can be expressed as a small number of large, regular transfers.

We ground this study in a representative cluster-scale runtime, where relations are partitioned across MPI ranks, and each iteration applies operators such as join, projection, and union, with deduplication performed in the pipeline to maintain set semantics until a fixed point [13] (see Figure 2). Storing these relations at scale is therefore part of the workflow: it exports intermediate state for inspection, captures outputs for downstream tools, and supports restart through checkpointing [20]. The baseline storage strategy serves as a comparison point, as each rank packs its local data partition into a contiguous buffer and writes it to a disjoint region of a shared file using either collective MPI-IO or POSIX offset writes [12]. Collective MPI-IO can improve throughput by coalescing rank-local requests via two-phase I/O, whereas POSIX offset writes reduce MPI coordination but may exhibit greater contention and variability at high concurrency when aggregation is limited [12].

Although this baseline design is simple and often effective, it exposes two limitations. First, packed binary output is typically opaque, which means the schema and layout descriptions (i.e., metadata) are outside the file, and users must maintain auxiliary bookkeeping to interpret results, validate correctness, or selectively reuse parts of the file. Second, performance is sensitive to the defaults and system settings. Even when each MPI rank logically produces contiguous segments of data, throughput depends on whether collective buffering behaves as intended and whether the system default settings match the I/O patterns [6, 26]. These effects are amplified when persistence is repeated across iterations [20]. These limitations are not fully addressed by MPI-IO tuning alone, as they stem from the lack of structured data representation and explicit metadata management.

To improve usability without sacrificing scalability, we redesign the RA storage path around *parallel HDF5*. HDF5 stores each relation as a self-describing dataset rather than a raw byte stream, enabling portable tooling and selective access [23]. In parallel, each MPI rank writes its partition as a hyperslab of a global dataset, while MPI-IO collective buffering can promote aggregation beneath HDF5 [26]. While optimized MPI-IO can achieve competitive raw throughput, HDF5 provides additional benefits beyond performance, including self-describing data, explicit schema representation, and structured dataset organization. These features simplify data interpretation, enable selective access, and reduce the need for external metadata management. Our results show that with appropriate tuning, HDF5 can match or exceed MPI-IO performance while providing these additional usability advantages. The main challenge is that self-describing formats add metadata and formatting costs, and prior work shows that parallel HDF5 can suffer sharp performance cliffs under default settings, with layout, alignment, metadata placement, and MPI-IO options strongly affecting throughput and variability [26]. We therefore apply and tune established optimization strategies for scalable HDF5 I/O, including file-organization techniques (e.g., subfiling-style approaches) and caching mechanisms (e.g., VOL-based methods) [4, 28]. Size-aware tuning and aggregation strategies are still applied at the MPI-IO layer, but HDF5 enables these to be combined with higher-level layout and metadata optimizations. Finally, we add lightweight checkpointing for RA state using the same structured pipeline so that the same optimizations that benefit normal outputs also reduce checkpoint and restart overheads [20].

Related ideas also appear in data analytics, graph-processing, and database systems that support iterative or recursive workloads, but they usually optimize a different layer of the stack than we do. Spark, for instance, was designed in part for iterative computations and recovers from failures primarily through lineage-based recomputation, using checkpointing more selectively to truncate long dependency chains or stabilize execution state [27]. GraphX extends this model to graph analytics by embedding iterative graph processing within a distributed dataflow framework and reducing repeated work through partitioning, join-aware execution, and materialized graph views [10]. Systems such as Naiad and Differential Dataflow move even closer to recursive and fixed-point evaluation by supporting cyclic, incremental, and nested iterative

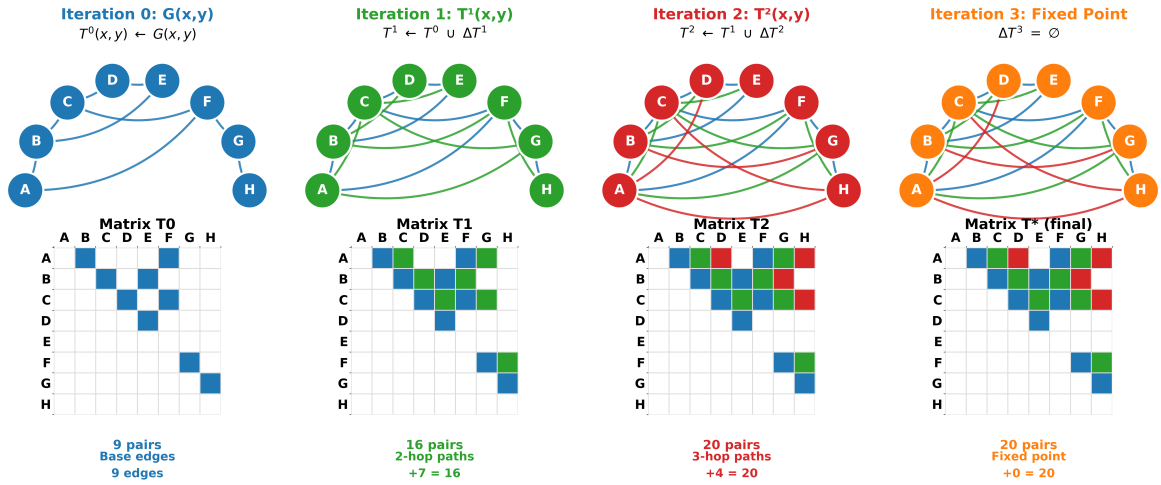

Figure 1. Transitive closure (TC) computes the reachability relation $TC(x, y)$: (x, y) is included if and only if a path exists from x to y. The figure reports end-to-end TC behavior under parallel execution, where each iteration expands reachability via join-based delta generation and merges new pairs until convergence.

dataflows, while Datalog engines such as Souffle emphasize compilation, indexing, and efficient relational execution for recursive inference tasks [11, 17, 18]. Large-scale dataflow engines such as Flink likewise support iterative pipelines and stateful execution, but their main concern is operator scheduling, state management, and distributed recovery rather than the behavior of shared-file parallel I/O [5]. Unlike these systems, which rely on lineage-based recomputation or fine-grained state management, our focus is on optimizing bulk persistence in MPI-based environments where checkpointing involves large, explicit writes to shared file systems. As a result, performance in our setting is dominated by physical I/O behavior, specifically collective I/O efficiency, layout selection, and metadata costs, rather than execution planning or recomputation strategies typical of dataflow systems.

Overall, we contribute a case study of I/O optimization for parallel iterated relational algebra. We first characterize a baseline persistence design that packs relation data, computes disjoint file regions using global size information, and writes using shared-file collective MPI-IO or POSIX offset writes [12]. We then introduce a parallel HDF5-based design that stores relations as datasets and uses configuration choices motivated by known optimizations. Beyond the case study, the key contribution is the systematic organization of optimizations into a size- and workload-aware policy [2–4, 26].

2 Background

This section reviews the computational model underlying iterated relational algebra (RA) as it arises in symbolic AI workloads, including Datalog evaluation [9]. Rather than treating persistence as a systems-level concern detached from the computation it serves, we characterize iterated RA programs, their fix-point evolution, and the relation growth that makes persistence intrinsic to execution. Transitive closure serves as our running example, it is minimal in formulation yet faithfully captures the fixpoint structure, monotonic data growth, and write-dominated I/O patterns common to a broad class of relational inference tasks [1, 25].

Relational algebra and Datalog. Datalog provides a declarative, rule-based interface that operationalizes into relational algebra (RA) [25]. Base facts reside in extensional relations, while recursive rule evaluation derives new tuples into intensional relations by composing RA operators (join, projection, selection, union, and set difference) over sets of tuples [1]. A rule body typically compiles to one or more joins followed by filtering and projection onto the head attributes; when rules are recursive, evaluation iterates these operator compositions until the intensional relations reach a least fixpoint and no new tuples are produced [25]. While each operator is elementary in isolation, this iterative application under recursion yields sufficient expressiveness for a broad class of deductive inference and analysis tasks. This compilation model is widely adopted in Datalog engines because it preserves clear declarative semantics while exposing opportunities for join ordering, indexing, and data movement optimization [9, 11, 13, 21]. While these systems focus on higher-level aspects such as execution models and scheduling, our work targets the shared-file I/O layer in MPI-based environments. MPI-IO and HDF5 are widely used in HPC systems, where performance is often driven by factors like collective I/O behavior, data layout, and metadata overhead. By focusing on this layer, our approach complements these systems and can be applied directly without requiring changes to the execution model.

Transitive closure as a worked example. Consider a directed graph represented as a binary adjacency relation $G(x, y)$. The transitive closure relation $T(x, y)$ captures reachability: $T(x, y)$ holds when there exists a path from x to y of length at least one, as we show in Figure 1 [25]. A standard Datalog program consists of a base rule: $T(x, y) \leftarrow G(x, y)$ and a recursive rule, $T(x, z) \leftarrow T(x, y),\ G(y, z)$. The first rule inserts all edges into T. The second, recursive, rule extends known reachability by one hop and may be solved bottom-up, starting from the base case, via iterated evaluation to a fixed point [25].

One iteration applies the recursive rule by joining T with G to extend paths, projecting away the intermediate vertex, filtering to newly discovered (x, z) pairs, and unioning them into T [13, 25]. Iterations repeat until no new pairs are produced, yielding a fixed point [25]. Because intermediate states are useful for validation and downstream use, I/O is often dominated by repeated materialization and checkpointing at iteration boundaries [8, 9, 11, 20].

Figure 2 illustrates the execution structure we are targeting. Each iteration partitions and materializes intermediate relations, performs an all-to-all exchange to realign data across ranks, and then evaluates the relational algebra operators for the current round. Because the loop repeats until a global fixed-point condition is met, iteration boundaries naturally induce repeated persistence and recovery points. This makes parallel I/O behavior, particularly checkpointing frequency and volume, a first-order performance factor, and motivates the I/O optimizations developed in this work.

Parallel Iterated RA Storage Needs. Iterated relational workloads execute as rounds, and distributed systems typically synchronize at iteration boundaries before testing for a fixed point [9, 13]. These boundaries are the natural persistence points, end-of-round snapshots capture a coherent logical state, whereas mid-round snapshots may include partially integrated updates [8, 20]. Because relations are hash-partitioned across ranks and stored in access-optimized in-memory structures, persisting state requires each rank to *pack* its local shard by traversing indices/blocks and linearizing tuples. The resulting buffer contiguity largely determines whether the storage layer observes a few large aligned writes or many small fragmented operations, affecting performance and variability across repeated checkpoints [11–14, 20, 21, 24].

3 RA Data Persistence with HDF5

This work focuses on I/O-layer optimization and does not modify the underlying RA execution model. We developed a strategy for storing relations as self-describing datasets in parallel HDF5. Our design goals include addressing the limitations of a binary packed approach and obtaining competitive high performance at scale. To achieve these goals, we have applied a number of I/O optimizations targeting HDF5 and MPI-IO. The challenge lies in the application of these optimizations systematically. We will briefly describe mapping the relations to HDF5 datasets and then describe our performance tuning efforts to achieve high I/O performance, guided by the optimization ladder in Table 1.

3.1 Baseline HDF5 Dataset Mapping

The structured path writes each relation as a two-dimensional HDF5 dataset with element type `uint64_t`. For a relation of arity `col_count` the dataset has shape `(total_rows,col_count)`. Each rank contributes to a contiguous block of rows using HDF5 hyperslabs. Rank r computes its local row count from the packed buffer size,

$$local\_rows_r = \frac{write\_size_r}{\text{sizeof(uint64\_t)} \cdot col\_count},$$

and all ranks perform a global reduction (e.g., `MPI_Allreduce` with `MPI_SUM`) to obtain `total_rows` so that the global dataset dimensions are known collectively prior to dataset creation. To place rank outputs contiguously in the global dataset, we compute a global byte offset via an exclusive prefix sum over per-rank write sizes,

$$write\_offset_r = \sum_{k<r} write\_size_k,$$

and convert this offset to a starting row index in the dataset,

$$start\_row_r = \frac{write\_offset_r}{\text{sizeof(uint64\_t)} \cdot col\_count}.$$

Each rank selects its target region with `H5Sselect_hyperslab`, which describes a selection using a file dataspace and a memory dataspace and supports up to four selection arrays (`start`, `stride`, `count`, `block`); The file dataspace (`filespace`) identifies the region within the global dataset using `start=(` $start\_row_r$ `,0)` and `count=(` $local\_rows_r$, col_count `)`. The memory dataspace (`memspace`) is created with dimensions $(local\_rows_r, col\_count)$, so the selected memory region has an implicit start of `(0,0)`. We pass `stride` and `block` as `NULL`, so HDF5 uses unit stride and unit block and therefore yields a single contiguous rectangular selection in both file and memory space, ensuring that rank r writes rows $[start\_row_r,\ start\_row_r + local\_rows_r)$ from its buffer to the corresponding rows in the global dataset. This mapping is held constant across **Baseline_HDF5** and **HDF5-OPT**.

3.2 Problems with baseline HDF5

A major source of overhead in the baseline HDF5 configuration is file and dataset management. When many ranks concurrently create or open HDF5 objects, metadata traffic increases and can stall progress, which our evaluation of Darshan logs reflects as inefficient scaling, where metadata and synchronization overheads dominate the total time. These effects are amplified for small relations, as the fixed cost of parallel I/O setup (metadata operations, collective buffering

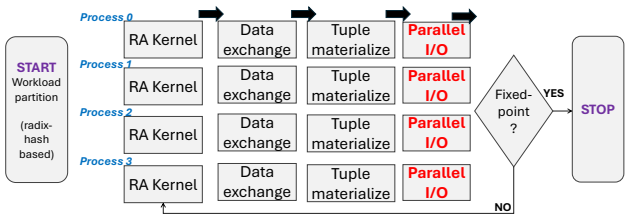

Figure 2. End-to-end workflow of a parallel iterated RA runtime (shown with 4 MPI ranks). Radix-hash partitioning and tuple materialization produce per-rank shards; iterations alternate between all-to-all exchanges and RA kernel execution, with parallel I/O used for persistence/checkpointing; termination is determined by a fixed-point test.

initialization, and hyperslab coordination) can exceed the cost of the data transfer itself. In iterated RA workloads, intermediate relations are materialized and checkpointed at iteration boundaries, so this setup cost is paid repeatedly and becomes a persistent component of runtime rather than a one-time overhead. Finally, hash-based partitioning can introduce substantial skew in per-rank data output sizes, causing collective writes to become straggler-bound when a small number of ranks dominate the total output volume. We therefore treat load imbalance as an I/O concern and quantify it using global reductions over per-rank write sizes (max-to-min ratio and coefficient of variation), and we observe in the progressive breakdown that mitigating imbalance reduces straggler effects, particularly at high process counts where coordination costs dominate.

3.3 I/O Optimizations

Table 1 summarizes the seven optimizations in our HDF5-OPT tuning ladder, each targeting a specific scalability bottleneck in baseline parallel HDF5/MPI-IO. We describe each step in detail below. These optimizations are not applied using a single fixed configuration; instead, we employ a size- and workload-aware policy that selects parameter settings based on output size, process count, and data skew. Not all optimizations in Table 1 are only HDF5-specific. MPI-IO tuning (e.g., collective buffering, aggregator selection, and buffer sizing) applies equally to MPI-IO and HDF5 backends, while layout selection, metadata management, and dataset creation strategies are specific to HDF5 for our experiment.

While several of the individual techniques in our optimization ladder (e.g., centralized aggregation for small outputs or two-phase I/O) are well established in the parallel I/O literature, their effectiveness is highly dependent on workload characteristics and scale. In iterated RA workloads, output size, degree of skew, and concurrency vary significantly across iterations, making a single fixed strategy ineffective. The key challenge is selecting and coordinating optimizations based on workload characteristics. Our contribution lies in organizing these techniques into a size and workload-aware policy that adapts to runtime conditions and avoids performance cliffs associated with static configurations.

Aggregation of small data writes. For outputs where the total data size across all ranks is below 10 MB, we adopt a centralized gather-and-write strategy, a well-known optimization for minimizing parallel I/O overhead at small scales. In this regime, the cost of coordination (e.g., collective buffering, metadata synchronization) can dominate the actual data transfer. To avoid this, rank-local buffers are gathered to rank 0 using `MPI_Gatherv`. Rank 0 then creates the output directory if needed, opens the HDF5 file using a serial file access property list, creates the dataset with a contiguous layout, and writes the complete dataset with a single `H5Dwrite` call. An `MPI_Barrier` ensures global completion before returning. We treat this as a size-dependent optimization that is beneficial only in the small-output regime. For larger outputs, centralized I/O becomes a bottleneck, and data writes are performed fully in parallel across ranks, with rank 0 involvement limited to lightweight metadata operations **[O1]**. Centralized allocation in this regime is a well-established optimization, as coordination and metadata overheads can dominate actual data transfer for small outputs. In our setting, we incorporate it as part of a size-aware policy rather than a standalone technique, selecting it only when it avoids unnecessary synchronization and contention.

Two-phase workflow for large outputs (serial create, parallel write). For large outputs, concurrent file and dataset creation by many ranks can lead to significant metadata contention. To mitigate this, we employ a two-phase workflow that separates metadata operations from data movement. In Phase 1, rank 0 performs file and dataset creation using a serial file access property list, based on the globally computed `total_rows`. If the file already exists, this phase is skipped. All ranks then synchronize to ensure that the dataset layout is visible before initiating any I/O. In Phase 2, all ranks reopen the file with a parallel file access property list and write their respective hyperslabs using collective I/O (`H5FD_MPIO_COLLECTIVE`), so that data transfer proceeds fully in parallel without routing through a single rank. We size transfer buffers at 64 MB by default and increase them to 256 MB for very large outputs (> 100 GB) to promote large, regular transfers **[O2]**.

Creation-time policies. When checkpoints write all elements explicitly, HDF5's default fill behavior can cause unnecessary work. During dataset creation, we enable early allocation (`H5D_ALLOC_TIME_EARLY`) to reserve space upfront and disable fill initialization (`H5D_FILL_TIME_NEVER`) **[O3]**. These settings remove redundant initialization I/O and reduce time spent in the creation phase

Layout selection: contiguous vs. chunked. We choose the dataset layout based on output size and skew **[O4], [O7]**.

Table 1. Optimization ladder used in HDF5-OPT

ID	Baseline bottleneck	Solution	Implementation / details
O1	Parallel setup dominates tiny outputs	Tiny-output gather	If total < 10 MB, gather all rank buffers to rank 0 using `MPI_Gatherv`. Rank 0 uses a serial FAPL, creates a contiguous dataset, and writes via a single `H5Dwrite`. Finalize with `MPI_Barrier`.
O2	Concurrent create/open triggers metadata contention	Two-phase create/open	Rank 0 creates file/dataset (serial FAPL; skip if exists), then synchronize. All ranks reopen with parallel FAPL and write hyperslabs collectively (`H5FD_MPIO_COLLECTIVE`).
O3	Avoidable initialization during dataset creation	Early alloc + no-fill	Enable early allocation (`H5D_ALLOC_TIME_EARLY`) and no-fill (`H5D_FILL_TIME_NEVER`) for full overwrites to avoid initialization overhead.
O4	Layout mismatch adds overhead or reduces efficiency	Layout policy	Contiguous for < 10 MB, > 10 GB, or imbalanced (ratio > 10). Otherwise use chunked + `deflate`(1). Chunk sizing: 64K (> 1M rows), 32K (100K–1M), 8K (otherwise).
O5	ROMIO defaults unstable across size and scale	ROMIO hints	Tiered `cb_nodes`: < 1 GB → $\min(N/4, 64)$; typical → $\min(N/8, 32)$; > 10 GB → $\min(N/16, 16)$. `cb_buffer_size` from 4–8 GB budget (clamped to 32–512 MB). Enable collective buffering; disable write sieving; force collective I/O.
O6	HDF5 defaults waste bandwidth	HDF5 tuning	Alignment tiers: 4/16 MB (< 1 GB), 8/32 MB (1–10 GB), 16/64 MB (> 10 GB), 4 KB/16 KB (< 10 MB). Metadata cache: 4–256 MB; sieve buffer: 32–256 MB.
O7	Skew makes collective I/O straggler-bound	Skew-aware phases	Compute ratio/CV via `MPI_Allreduce`. Use 4 phases if ratio > 10 or CV > 1, 8 phases if ratio > 100 or CV > 2. Partition via `MPI_Comm_split`; enforce barrier per phase.

Contiguous layout is used for small outputs (<10 MB), large outputs (>10 GB), and for imbalanced cases where chunking overhead or straggler effects dominate. For balanced intermediate outputs, we use chunked layout with lightweight compression (deflate level 1). Chunk height is selected by scale: 64K rows for outputs above 1M rows, 32K rows for 100K-1M rows, and 8K rows otherwise. This policy targets the regime where chunking improves locality and reduces bytes without adding excessive metadata overhead.

Size-Aware Configuration. We configure the parallel file access property list via `configure_hdf5_io(comm,total_data_size)`, which selects MPI-IO **[O5]** and HDF5 **[O6]** parameters based on output size and process count.

At the MPI-IO layer, we pass ROMIO collective buffering hints through an `MPI_Info` attached to the HDF5 FAPL. We choose the number of aggregators (`cb_nodes`) as $\min(N/8, 32)$ by default, increase to $\min(N/4, 64)$ for outputs below 1 GB to expose more aggregation, and reduce to $\min(N/16, 16)$ for outputs above 10 GB to limit coordination overhead, where N is the process count. The collective buffer size (`cb_buffer_size`) is computed from a global budget (4 GB up to 10 GB outputs), divided by N and clamped to 32-512 MB per process. We enable collective buffering for writes (`romio_cb_write=enable`), disable write data sieving (`romio_ds_write=disable`), and enforce collective semantics (`romio_no_indep_rw=true`) **[O5]**.

At the HDF5 layer, we set `H5Pset_alignment(threshold,block_size)` with size tiers: 4/16 MB below 1 GB, 8/32 MB for 1-10 GB, and 16/64 MB above 10 GB, while using 4 KB/16 KB for very small outputs (below 10 MB) to avoid padding. We scale the metadata cache (`H5Pset_cache`) from 4 MB to 256 MB based on expected I/O volume, and set the sieve buffer (`H5Pset_sieve_buf_size`) to match the collective buffering scale (typically 32–256 MB) **[O6]**.

Parameter selection policy. The parameter selection policy is driven by three runtime-visible signals: total output size, MPI process count, and skew in per-rank write sizes. Output size is the primary driver of the I/O configuration, determining both the overall strategy (centralized vs. parallel) and key parameters such as layout, alignment, buffering, and aggregation behavior through coarse-grained regimes (e.g., < 10 MB, 1–10 GB, and > 10 GB). Process count influences aggregation behavior in conjunction with output size, including the number of ROMIO aggregators and per-process buffer sizes, which are derived from a global memory budget and scaled by concurrency with bounds to maintain efficiency. Skew in per-rank output sizes determines whether phased I/O is required, based on imbalance metrics such as max-to-min ratio and coefficient of variation, with predefined thresholds triggering additional I/O phases (e.g., 4 or 8 phases) to mitigate straggler effects.

Checkpointing. At fixed iteration boundaries (every 400 iterations in our experiments), each relation writes a full snapshot of all tuples and a delta snapshot containing only tuples produced since the previous checkpoint, using the same HDF5-OPT mechanisms and iteration-encoded filenames. We time full and delta writes separately. On failure,

recovery restarts from the latest full snapshot plus subsequent deltas, avoiding replay of earlier iterations.

4 Evaluation

We evaluate the performance of persisting data using three real-world graph datasets running transitive closure (TC) computations at scale. We measure the write time of persisting checkpoints of growing relations in TC. We first describe the experimental setup and compare the execution time of the TC algorithm with three datasets and using different I/O implementations.

Table 2. Dataset characteristics.

Dataset	Edges	Iterations	TC Paths	Output Size
vsp_finan	552K	1,040	0.91B	15 GB
fe_ocean	410K	494	1.67B	29 GB
usroad	165K	1,212	0.87B	20 GB

Experimental setup. We ran the experiments on the Perlmutter system at NERSC [19] using its Lustre parallel file system and MPI/HDF5 with Lustre-aware tuning. We scaled the experiments from 32 to 2048 processes, with 4X increments. Although the datasets are on the order of tens of gigabytes, our goal is to evaluate scalability under increasing concurrency rather than data volume alone.This design intentionally stresses coordination and metadata overheads, which become the dominant factors at high concurrency. As process count increases, per-rank output size decreases, making coordination, metadata overhead, and request granularity dominant factors in I/O performance. Evaluating up to 2048 processes therefore allows us to expose scalability bottlenecks and assess the effectiveness of our optimizations under high-concurrency conditions typical of HPC environments. For datasets, we use three SuiteSparse graphs that induce distinct convergence and checkpoint behaviors: `usroad` (many iterations), `vsp_finan` (moderate iterations), and `fe_ocean` (fewer iterations but larger output) [7, 15, 16]. In Table 2, we show the characteristics of the graph datasets and their output sizes. For performance evaluations, we compare four configurations: baseline MPI-IO, optimized MPI-IO (labeled as MPI-IO OPT), baseline HDF5, and optimized HDF5 (i.e., HDF5-OPT). MPI-IO OPT applies MPI-IO tuning strategies such as collective buffering and ROMIO hint configuration (e.g., `cb_nodes` and `cb_buffer_size`), but does not include HDF5-specific optimizations such as layout selection, metadata management, or size-aware tuning policies. These additional capabilities in HDF5-OPT enable both performance tuning and structured data representation. For the HDF5-OPT, we show the performance with all HDF5 optimizations enabled. Checkpointing is performed every 400 iterations in our experiments. This interval is chosen as a fixed parameter to enable consistent comparison across configurations. In general, the optimal checkpoint frequency depends on factors such as system failure rate, recomputation cost, and storage overhead (e.g., as modeled by Young/Daly formulas). Our focus in this work is on optimizing the cost of checkpoint I/O itself, and the relative performance trends we observe are independent of the specific checkpoint interval.

Overall performance. In Figure 3, we summarize the end-to-end TC time across the four I/O configurations. Two trends are consistent across all datasets. First, basic HDF5 is dominated by metadata and synchronization costs at scale and thus performs worse than the baseline MPI-IO. Second, HDF5-OPT eliminates these bottlenecks and becomes competitive with, and often faster than (14.6%–58.6%), MPI-IO OPT. On the `usroad` dataset, where checkpointing is frequent and iteration count is high, HDF5-OPT achieves the largest improvements over its baseline. At larger scales, the optimized path is faster than baseline HDF5, which indicates that coordination overhead is the primary limiting factor for default configurations rather than raw device bandwidth.

Progressive optimizations of HDF5. In Figure 4, we show progressive improvement of HDF5 optimizations for the `usroad` use case. The baseline HDF5 implementation is consistently the slowest, as expected. Enabling chunking and compression yields the first substantial reduction by improving the write pattern and, when data are compressible, reducing the number of written bytes. Subsequent layout refinements further improve performance by improving locality and avoiding inefficient materialization access. MPI-IO hints provide additional gains by improving collective transfer behavior as process count increases. Load-imbalance-aware phased I/O then mitigates straggler effects by coordinating participation in I/O, which becomes increasingly important at higher scales. Finally, HDF5-OPT achieves the best performance across all scales, with performance gains increasing from 14–25% at 32 processes to 35–58% at 2048 processes, where per-rank I/O becomes small and request granularity and coordination dominate. With checkpointing every 400 iterations, the optimized I/O reduces checkpoint overhead by limiting metadata traffic, aligning transfers to Lustre, and coalescing writes through buffering.

5 Conclusion

Parallel HDF5 can be a practical persistence format for large, write-heavy iterative workloads when its configuration is treated as a first-class design choice rather than a default. By separating creation from bulk transfer and applying size-aware settings for metadata, alignment, buffering, layout, and MPI-IO behavior, we avoid the high-concurrency performance cliffs that otherwise dominate runtime. At scale, the resulting pipeline matches or exceeds the throughput of conventional packed-binary baselines while retaining the benefits of self-describing data.

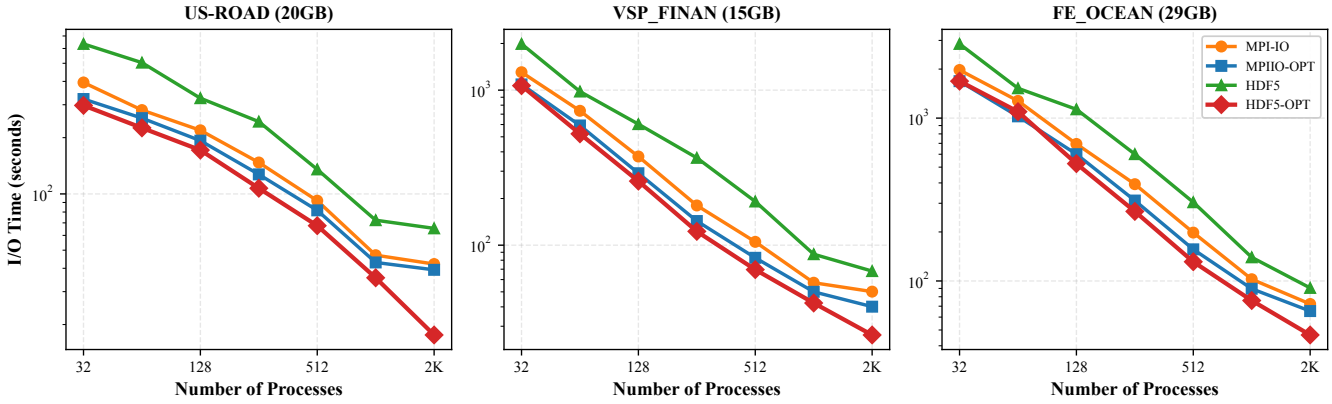

Figure 3. I/O configuration comparison across three datasets. Basic HDF5 suffers from metadata and synchronization overheads, while HDF5-OPT consistently improves scalability and overall runtime relative to MPI-IO baselines.

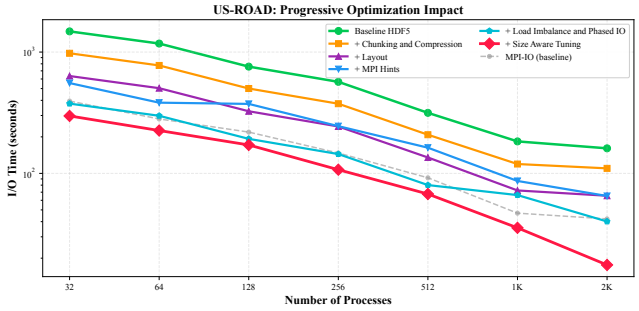

Figure 4. Progressive optimization impact on `usroad`. Starting from the first working HDF5 implementation, we incrementally enable chunking and compression, layout refinements, MPI-IO hints, load-imbalance aware phased I/O, and finally size-aware tuning.

In future, we plan to reduce reliance on manual tuning by selecting HDF5 and MPI-IO parameters automatically from lightweight runtime signals such as output skew, file-system responsiveness, and iteration-to-iteration stability. We also plan to investigate file-organization strategies that curb metadata pressure under extreme concurrency, including group-oriented layouts and subfiling-style approaches where available.

Acknowledgment

This effort was supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL, under an LBNL subcontract to OSU (GR130493), and NSF Award numbers 2316158, 2529911, and 2316159.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley, Reading, MA.

[2] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing I/O Performance of HPC Applications with Autotuning. *ACM Transactions on Parallel Computing* 5, 4 (2019), 15:1–15:27. doi:10.1145/3309205

[3] 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. doi:10.1007/s11390-020-9822-9

[4] Suren Byna, Mohamad Chaarawi, Quincey Koziol, John Mainzer, and Frank Willmore. 2017. Tuning HDF5 Subfiling Performance on Parallel File Systems. In *Proceedings of the Cray User Group (CUG 2017).* https://cug.org/proceedings/cug2017_proceedings/includes/files/pap106s2-file1.pdf

[5] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* (2015).

[6] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 2011. Understanding and Improving Computational Science Storage Access through Continuous Characterization. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST).* 1–14. doi:10.1109/MSST.2011.5937212

[7] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1, Article 1 (Dec. 2011), 25 pages. doi:10.1145/2049662.2049663

[8] Ke Fan, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. 2021. Exploring MPI Collective I/O and File-per-process I/O for Checkpointing a Logical Inference Task. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 965–972. doi:10.1109/IPDPSW52791.2021.00153

[9] Thomas Gilray, Sidharth Kumar, and Kristopher K. Micinski. 2021. Compiling Data-Parallel Datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC 2021).* ACM, 23–35. doi:10.1145/3446804.3446855

[10] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[11] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification (CAV 2016) (Lecture Notes in Computer Science, Vol. 9780).* Springer, 422–430. doi:10.1007/978-3-319-41540-6_23

[12] Qiao Kang, Sunwoo Lee, Kai-yuan Hou, Robert Ross, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2019. Improving MPI Collective I/O Performance With Intra-node Request Aggregation. arXiv preprint arXiv:1907.12656. https://arxiv.org/abs/1907.12656

[13] Sidharth Kumar and Thomas Gilray. 2020. Load-Balancing Parallel Relational Algebra. In *High Performance Computing (ISC High Performance 2020)*. Lecture Notes in Computer Science, Vol. 12151. Springer, Cham, 288–308. doi:10.1007/978-3-030-50743-5_15

[14] Sidharth Kumar, Steve Petruzza, Will Usher, and Valerio Pascucci. 2019. Spatially-aware Parallel I/O for Particle Data. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. ACM. doi:10.1145/3337821.3337875

[15] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Stanford University. http://snap.stanford.edu/data Accessed 2026-02-11.

[16] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. 2005. On trip planning queries in spatial databases. In *Proceedings of the 9th International Conference on Advances in Spatial and Temporal Databases*. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/11535331_16

[17] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. https://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[18] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*.

[19] National Energy Research Scientific Computing Center (NERSC). [n. d.]. Perlmutter System Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/. Accessed: 2026-02-11.

[20] Konstantinos Parasyris, Giorgis Georgakoudis, Leonardo Bautista-Gomez, and Ignacio Laguna. 2021. Co-Designing Multi-Level Checkpoint Restart for MPI Applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. doi:10.1109/CCGrid51090.2021.00020

[21] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, New York, NY, USA. doi:10.1145/2892208.2892226

[22] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2016. Modular HPC I/O Characterization with Darshan. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. IEEE. doi:10.1109/ESPT.2016.006

[23] The HDF Group. 1997. Hierarchical Data Format, version 5 (HDF5). Software and documentation. https://www.hdfgroup.org/solutions/hdf5/ 1997–present.

[24] Yuichi Tsujita, Atsushi Hori, and Yutaka Ishikawa. 2015. Striping Layout Aware Data Aggregation for High Performance I/O on a Lustre File System. In *High Performance Computing (ISC High Performance 2015) (Lecture Notes in Computer Science, Vol. 9137)*. Springer, 282–290. doi:10.1007/978-3-319-20119-1_21

[25] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press.

[26] Bing Xie, Houjun Tang, Suren Byna, Jesse Hanley, Quincey Koziol, Tonglin Li, and Sarp Oral. 2021. Battle of the Defaults: Extracting Performance Characteristics of HDF5 under Production Load. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. doi:10.1109/CCGrid51090.2021.00015

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–28. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

[28] 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 IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. doi:10.1109/CCGrid54584.2022.00015