Scaling Embedded In-Situ Indexing with DeltaFS

Qing Zheng[†], Charles D. Cranor[†], Danhao Guo[†], Gregory R. Ganger[†], George Amvrosiadis[†], Garth A. Gibson[†]
Bradley W. Settlemyer[‡], Gary Grider[‡], Fan Guo[‡]
[†]Carnegie Mellon University, [‡]Los Alamos National Laboratory
{zhengq, chuck, danhaog, ganger, gamvrosi, garth}@cmu.edu, {bws, ggrider, guofan}@lanl.gov

***Abstract*—Analysis of large-scale simulation output is a core element of scientific inquiry, but analysis queries may experience significant I/O overhead when the data is not structured for efficient retrieval. While in-situ processing allows for improved time-to-insight for many applications, scaling in-situ frameworks to hundreds of thousands of cores can be difficult in practice. The DeltaFS in-situ indexing is a new approach for in-situ processing of massive amounts of data to achieve efficient point and small-range queries. This paper describes the challenges and lessons learned when scaling this in-situ processing function to hundreds of thousands of cores. We propose techniques for scalable all-to-all communication that is memory and bandwidth efficient, concurrent indexing, and specialized LSM-Tree formats. Combining these techniques allows DeltaFS to control the cost of in-situ processing while maintaining 3 orders of magnitude query speedup when scaling alongside the popular VPIC particle-in-cell code to 131,072 cores.**

I. Introduction

Exascale platforms are poised to set new records for performance, memory capacity, and storage throughput. To use them efficiently, systems software is expected to scale to unprecedented levels of concurrency alongside scientific applications that will generate larger and more detailed output than they do today [1, 2]. The resulting datasets will require scalable analysis codes to unlock the scientific insight buried within the data. In-situ processing, the process of coupling analysis or indexing codes with running scientific simulations, promises to improve the quality and resolution of scientific data analysis. However, scaling in-situ codes alongside scientific applications presents significant performance challenges. This paper presents the challenges and lessons learned when modifying the DeltaFS distributed filesystem for more scalable in-situ performance [3]. Our contribution is a set of techniques forming the underpinning of DeltaFS' scalable in-situ indexing capability. We target scientific applications that run across hundreds of thousands of cores and need fast point and small-range queries. While the techniques we present are optimized for scalable in-situ indexing, they may prove useful for scaling other types of in-situ processing codes and for High Performance Computing (HPC) systems software in general.

Scientific applications generate output by periodically halting computation and persisting memory contents to storage [4]. Platforms such as Trinity [5] at Los Alamos National Lab (LANL) provide petabytes of RAM, and this process is expected to be time-consuming and blocked on storage, leaving idle CPU cycles. Our in-situ technique, *embedded in-situ indexing*, is designed to use these idle CPU cycles to dynamically reorganize and index data as it is streamed to storage. Processing data across large numbers of idle computing and network resources enables the calculation of efficient indexes. However, embedded in-situ processing presents the additional challenge of scaling directly *within* the application.

Traditional post-processing programs use a machine's entire memory to achieve scaling to large numbers of cores, whereas embedded in-situ processing codes must achieve scale while minimizing their memory footprint. An in-situ function co-located with an application must additionally avoid impacting the application's runtime by only scavenging idle resources and disrupting performance as little as possible. Our experiments use Vector Particle-In-Cell (VPIC) [6], a highly scalable particle simulation code developed at LANL, to perform large-scale simulations with a shared scientific goal of performing trajectory analysis across a small subset of one trillion particles. Our techniques allow VPIC particles to be efficiently indexed and queried fast even at extreme scale.

While updating DeltaFS for a more scale-out implementation, our **first set of challenges** were related to achieving efficient data shuffling under intense memory pressure. This pressure stems from the fact that scientific applications typically use almost all available memory. We have found data shuffling to be imperative in constructing an optimized storage layout for efficient data analysis, however DeltaFS must reorganize data with a small memory footprint on each of the application's compute nodes. Limited memory greatly complicates latency hiding for all-to-all communication and requires sophisticated management of buffering throughout our system. We achieve scalable data reorganization through an efficient in-situ pipeline that simultaneously enables scalable network communication (Section III) and efficient I/O to the underlying storage (Section IV-D).

Our **second set of challenges** were specific to constructing an efficient indexing mechanism that generates more optimized storage layouts for fast point and small-range queries. To dramatically improve time-to-insight, scientific data often requires a different set of indexing techniques than is commonly used in popular key-value stores [7, 8]. This is especially true when the indexing code is co-scheduled with the application and is under intense memory pressure. We describe a customized LSM-Tree [9] format that achieves fast queries at the read path while not requiring the write path code to consume excessive I/O bandwidth for storage reorganization (Section IV-A and IV-B). In addition, we show an analysis format that enables both high degrees of parallelism and fast lookups into logged data structures (Section IV-C).

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

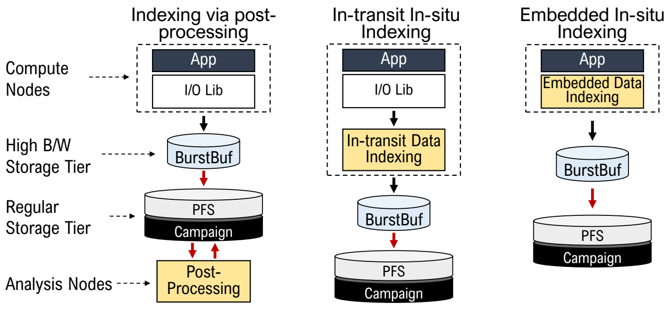


Fig. 1. **Indexing during post-processing is time-consuming for applications with intensive post-hoc analysis. In-transit indexing avoids post-processing but requires extra resources to process data in-situ. Instead, embedded in-situ indexing uses spare resources on the compute nodes of an application to perform data indexing.**

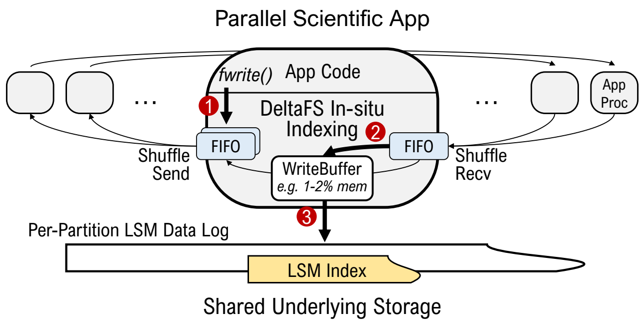


Fig. 2. **DeltaFS in-situ indexing is library code linked into the processes of a parallel job. Data written by the job is first partitioned and shuffled to the process responsible for it (Step 1). Then, the data is received at the other end (Step 2), and indexed using a modified LSM-Tree (Step 3).**

II. Background and Related Work

Many scientific applications are time-based simulations that run in timesteps. In many scenarios simulation state is periodically saved to storage for post-analysis. To operate efficiently, applications minimize their time spent writing state to storage in order to maximize time spent in simulation [4].

To reduce simulation I/O time it is important to fully utilize the underlying storage bandwidth. For simulations whose state involves lots of small objects [10, 11], the simulation state is most efficiently persisted when these small objects are batched together and appended to storage using large sequential writes [12]. As data is not always appended in the optimal order for post-analysis queries [13], a separate *post-processing* step is often used to reorder or index the data after a simulation to enable fast queries [10, 14, 15], as illustrated in Figure 1.

Because post-processing rereads data from storage after each simulation, it has become increasingly inefficient as the compute-I/O gap grows. Reading is even slower when post-processing is performed on a separate, smaller cluster (e.g., one dedicated to data analytics) due to often reduced bandwidth to the underlying storage. Fast burst-buffer storage exists [16–18], but its limited capacity cannot always absorb the entire simulation output. For large simulations a considerable amount of data is still read from regular storage.

One way to reduce post-processing is to build data indexes dynamically as data is written to storage. This is known as *in-situ* indexing and is typically achieved by adding additional nodes to a job to stage data so data indexes can be computed asynchronously (i.e., *in-transit*) while the original simulation proceeds to its next timestep [19–24]. As shown in Figure 1, one drawback to this approach is the extra job nodes that must be dedicated to perform the index calculation.

To avoid dedicating nodes for indexing, our previous work, the DeltaFS Indexed Massive Directory [3], uses only idle computing resources on the compute nodes of a job to perform the indexing calculation. Idle resources are temporarily available because the scientific application is effectively forced to pause its computation during its I/O phases [18]. In this paper we use the term *embedded* in-situ indexing to refer to the scenario where only idle job resources are used.

Our previous implementation achieved effectiveness at modest scale (4,000 processes) [3]. Achieving similar efficiency at larger scales required several code changes. Before we discuss our techniques and lessons learned, this section presents an overview of our system as proposed previously [3], discusses related work, and reviews our previous results.

A. *DeltaFS In-Situ Indexing API and Semantics*

DeltaFS in-situ indexing is library code embedded inside the distributed processes of a parallel scientific application. An important reason for designing DeltaFS in-situ indexing as an embedded library is to remove the bottlenecks of traditional HPC storage systems and to leverage the idle CPU cycles and fast interconnection networks available on the compute nodes of a parallel job for the computation of data indexes [25, 26]. We assume that the underlying storage is a bottleneck and that the application does not overlap its I/O with the simulation's timesteps so spare computing resources are available during the simulation's I/O phases [4].

By indexing data in-situ, our goal is to completely bypass, or drastically reduce, data post-processing for certain classes of post-analysis queries. Our in-situ indexing implementation consists of a data shuffling component and a data indexing component. Data produced by a job is shuffled within the job according to a user-supplied data partitioning function such as a hash function. Each job process manages a partition and indexes the partition's data using a customized LSM-Tree [9] introduced in Section IV. Indexed data is written to storage as per-partition log files [27]. The entire process is illustrated in Figure 2. The process is currently optimized for point and small-range queries. In the rest of our descriptions, we focus on LANL's VPIC application as an example [6].

VPIC is a scalable particle simulation code used at LANL. In a VPIC simulation, each simulation process manages a region of cells in the simulation space through which particles move. Every few timesteps the simulation stops and each simulation process writes a per-process file containing the state of all the particles currently managed by the process. State for each particle is 48 bytes. Large-scale VPIC simulations have been conducted with trillions of particles, generating terabytes of data for each recorded timestep [28, 29].

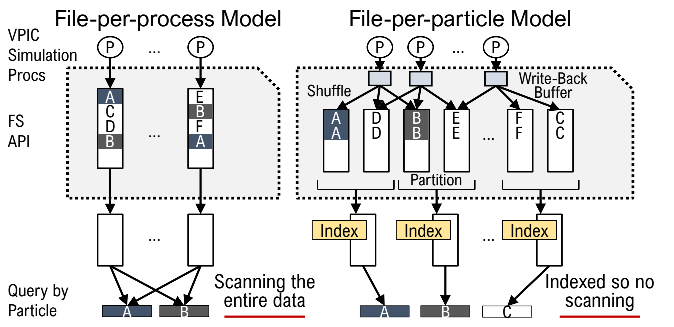


Fig. 3. **Unmodified VPIC writes one file per process without performing any in-situ processing. To index VPIC particles with DeltaFS, we modify VPIC to write the state of each particle into an Indexed Massive Directory using particle IDs as the filenames. Dynamically created directory indexes keyed on filenames allow us to quickly retrieve per-particle information right after a simulation. Indexed particle data is packed and stored by DeltaFS as large per-partition log objects in the underlying storage.**

VPIC domain scientists are often interested in some property (e.g., trajectories) of a tiny subset of particles with specific characteristics (e.g., high energy). While the identities of these particles (i.e., particle IDs) can be known at the end of a simulation, reading back the trajectories of these particles is like finding a needle in a haystack. Because particle state is written out-of-order, without post-processing a complete scan of the simulation output is needed to recall per-particle information. To avoid such scans, DeltaFS dynamically partitions particles and creates an index for each partition that maps particle IDs to particle data locations. As DeltaFS indexes data in-situ, per-particle information can be easily queried by particle ID at each data partition right after a simulation without requiring the data to be further post-processed.

In this example, retrieving the state of a particle at a specific timestep represents a *point* query, and retrieving the trajectory of a particle over a range of timesteps represents a *small-range* query. The range is small as simulations rarely persist massive numbers of timesteps [4]. To use DeltaFS in-situ indexing, one first creates an Indexed Massive Directory.

INDEXED MASSIVE DIRECTORIES are a special type of directory that can be dynamically created inside a DeltaFS namespace. While at their simplest these directories contain a massive number of typically tiny files, each such directory is effectively a distributed key-value store packing and storing data in an underlying object store. Filenames are *keys*. File contents are *opaque values*. Keys are inserted by creating new files. Values are appended to these files. Each directory is independently partitioned and indexed, both keyed on filenames. File contents are stored in one or more per-partition log files in the underlying storage, similar to PLFS [27].

To index particles, VPIC creates a file for each particle using the ID of the particle as the filename. As directory indexes are built on filenames (i.e. particle IDs), per-particle state is effectively indexed and can be quickly located. To retrieve per-particle information, a reader program uses a POSIX-like API provided by DeltaFS to access and open the corresponding particle file. Internally, DeltaFS uses its directory indexes to locate file data. This data locating process is transparent to the reader program. File data, which is opaque to DeltaFS, is read by the reader program and interpreted by it as particle data.

Figure 3 compares the *file-per-process* model used by the original VPIC with the new *file-per-particle* model enabled by DeltaFS. Unmodified VPIC simulations write their output to an underlying storage [30–33] using one file per process. Without post-processing, retrieving the trajectory of a specific particle requires reading the entire simulation output (upwards of TBs of data). With DeltaFS, VPIC stores data within an Indexed Massive Directory. Because data is dynamically partitioned and indexed, locating a particle's trajectory after a simulation requires reading only the smaller indexes from a single directory partition (typically only MBs of data) [3].

B. Related Work

Byna et al. have published the largest petascale particle simulations using VPIC [10, 28, 29]. With two trillion particles and 2000 time steps of simulation the authors produced 350 TBs of data (including checkpoints) and detail the series of optimizations required to use a single shared HDF5 file output model. Some of the difficulties encountered while analyzing the resulting particle outputs motivated the creation of the DeltaFS embedded in-situ indexing pipeline for VPIC.

Rich in-transit data processing capabilities are provided by multiple middleware libraries such as PreDatA [20], GLEAN [21, 34], NESSIE [35], and DataSpaces [22]. These systems all use auxiliary nodes to provide analysis tasks. Similarly, systems such as Damaris [36] and Functional Partitioning [37] co-schedule analysis, visualization, and de-duplication tasks on compute nodes, but require dedicated cores. DeltaFS embeds indexing computation directly within the application processes and performs the processing during the application's regular output methods.

The GoldRush runtime [38] provides an embedded in-situ analytics capability by scheduling analysis tasks during idle periods in simulations using an OpenMP threaded runtime. The analysis tasks leverage the FlexIO [39] capability within ADIOS [40] to create shared memory channels for generating analysis tasks inputs to execute during idle periods of application execution. The embedded in-situ framework within DeltaFS instead co-schedules analysis tasks (i.e. partitioning and indexing) with the application's I/O output phase. While Goldrush is extremely effective at scavenging idle resources within the OpenMP runtime model, DeltaFS instead focuses on co-scheduling analysis tasks for single-threaded bulk-synchronous applications.

The SENSEI in-situ analysis framework [41] provides a generic library capable of running computationally efficient in-situ tasks on dedicated or shared resources. Their studies included instrumenting a variety of codes and mini-apps. Additionally, they concluded that most in-situ analysis tasks require little memory overhead. DeltaFS is able to use only 3% of the system memory to do effective latency hiding for in-situ operations even though the analysis requires shuffling and indexing the entire output dataset.

FastQuery [14, 15], a popular indexing and query library for scientific data, uses parallel, compressed bitmap indexes similar to the bitmap indexing described by FastBit [42], and has been deployed as part of in-situ indexing service to accelerate subsequent reads [43]. DeltaFS creates a similar compressed bitmap index following the shuffle phase to quickly filter subsequences from within a partition. By customizing a bitmap index for partitioned particle data, DeltaFS is able to reduce the overall index size and reduce storage overhead.

Many systems have proposed variants of LSM-Trees to improve performance. WiscKey [44] reduces the I/O amplification associated with compaction by storing keys and values separately and only performing compaction on the keys. Both Monkey [45] and SlimDB [46] use analytical models to generate optimized filter layouts that balance per-filter performance with available memory. LSM-Trie [47] uses an incremental compaction scheme [48] to reduce compaction overhead, and uses clustered indexes to improve query performance. VT-Tree [49] uses a customized compaction procedure that avoids re-sorting in-order data. The DeltaFS's custom LSM-Tree implementation presented in this paper is inspired by these reorganizations, particularly the LSM-Trie, though DeltaFS is primarily optimized for point and small-range queries.

The distributed data partitioning and indexing capability of MDHIM [50] is similar to that of DeltaFS, though there are several key differences. First, DeltaFS uses an LSM-Tree that is more optimized for small value retrieval and in-situ scenarios. Second, DeltaFS uses a POSIX-like file system abstraction while MDHIM uses a key-value store abstraction. Finally, MDHIM relies on MPI for inter-process communication while DeltaFS uses Mercury RPC [51, 52] to run seamlessly across platforms supporting different network transports [53–55].

C. Previous Results

This section reviews our *previous* experimental results [3]. Our previous experiments were performed on LANL's Trinitite cluster, a smaller clone of the Trinity machine used for testing and debugging. Each Trinity or Trinitite compute node has 32 CPU cores and 128GB RAM. For each experiment we ran a real VPIC configuration both with and without DeltaFS. For VPIC baseline runs, the simulation wrote one output file per simulation process. For DeltaFS runs, the VPIC simulation wrote into an Indexed Massive Directory, with DeltaFS dynamically partitioning and indexing the data, and writing the results as parallel logs. Both the data partitioning and the indexing were keyed on particle IDs, and the data was partitioned by a hash function [56]. Our largest simulation simulated 48 billion particles across 3,096 processes.

Across all runs, simulation data was first written to a burst-buffer storage tier and was later staged out to an underlying Lustre file system. We kept the compute node to burst-buffer node ratio fixed at 32 to 1. Each Trinity or Trinitite burst-buffer node can absorb data at approximately 5.3GB per second.

After each simulation, queries were executed directly from the underlying file system with each query targeting a random particle and reading all of its data. Particle data was written out over time as the simulation ran through timesteps. Each simulation was configured to output all particle data for 5 of those timesteps. To retrieve the trajectory of a particle, the VPIC baseline reader always reads the entire simulation output and each query was repeated only 1 or 2 times. DeltaFS handles queries more efficiently so all DeltaFS queries were repeated 100 times. Each query started with a cold data cache. The average query latency was reported. While DeltaFS used a single CPU core to execute queries, the baseline reader used the number of simulation processes to read data in parallel.

Figure 4(a) shows the read performance. While the baseline reader used all the CPU cores to run queries, a single-core DeltaFS reader was still up-to 5,112x faster. This is because without an index for particles, the baseline reader reads all the particle data so its query latency is largely bounded by the underlying storage bandwidth. As DeltaFS builds indexes in-situ, it is able to quickly locate per-particle information after a simulation and maintain a low query latency (within 300ms in these experiments) as the simulation scales.

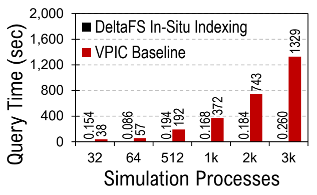
Figure 4(b) shows the I/O overhead DeltaFS adds to the simulation's I/O phases for building the data indexes. Part of the overhead comes from writing the indexes in addition to the original simulation output. The rest is due to the reduced I/O efficiency resulting from DeltaFS performing the in-situ indexing work. DeltaFS has large but decreasing overheads for the first 5 runs. This is because those jobs are not large enough to saturate the burst-buffer storage, so the system is dominated by the extra work DeltaFS performs to build the indexes. For bigger runs the jobs start to bottleneck on the storage and we see a DeltaFS slowdown of approximately 15%.

As will be discussed in Section III, because our previous implementation required excessive memory for efficient communication to achieve data partitioning and indexing, it was unable to scale beyond thousands of processes. This paper shows techniques to overcome this limitation. To further improve performance, Section IV discusses additional LSM-Tree techniques we used to enable data to be more efficiently indexed and queried. Critically, none of the application facing interfaces described previously [3] required any changes to scale DeltaFS to hundreds of thousands of processes.

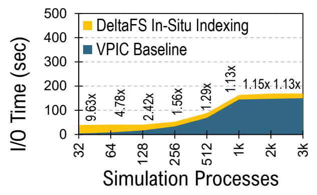
III. Streaming Data Shuffling

Recall from Section II-A that our goal is to efficiently speed up post-analysis queries while drastically reducing post-processing. To achieve this, DeltaFS features an in-situ indexing pipeline consisting of a data shuffling component and a data indexing component. This section shows techniques for this data shuffling component which was limiting the system's scalability. We address data indexing in Section IV.

To improve data placement, DeltaFS implements an all-to-all shuffle to dynamically partition data across all DeltaFS instances running inside the distributed processes of a parallel application [3]. As illustrated in Figure 2, each piece of data written into the DeltaFS pipeline is assigned a destination according to a data partitioning function. According to the destination the data is buffered at one of the sender queues



(a) Previous Query Performance



(b) Previous I/O Performance

Fig. 4. **Results from our previous VPIC simulation runs using the old DeltaFS code on LANL's Trinitite cluster. Our biggest job used 99 compute nodes, 3,096 CPU cores, and simulated 48 billion particles. While the baseline VPIC reader used all the CPU cores to search particles in parallel, all DeltaFS queries were executed on a single CPU core.**

bound to each peer shuffle receiver. When a queue is full, all data in the queue is sent as a single large message to the destination. Each DeltaFS instance is both a shuffle sender and a shuffle receiver. Because data belonging to each data partition is sent to and indexed by a single receiver, DeltaFS is able to efficiently answer queries by looking only at the indexes and data stored by one of its receivers [57–59].

With this paper targeting point and small-range queries, we have chosen to partition data using hash functions. This section mainly focuses on the inter-process communication needed for distributing the data to achieve proper partitioning. We show techniques for scalable all-to-all communication and measure their effectiveness in DeltaFS.

A. *Partitioning streaming data by shuffling*

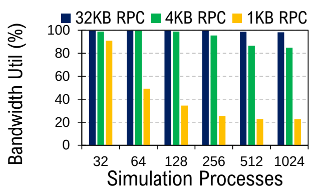
Lesson 1: *Shuffling data in-situ is important for limiting query accesses to a small subset of data, and performing it efficiently requires deep message buffering.*

To quickly shuffle data, we need to efficiently support all-to-all communication for both small- and large-scale simulations. But even with modern HPC interconnects [60], the cost of large-scale all-to-all communication can be high if frequent communication consists of small payloads that prevent us from fully utilizing the network's bandwidth. To hide network latency, we buffer adequate data (e.g., 32KB) before sending it to a remote process. To further increase efficiency, we perform network operations asynchronously.

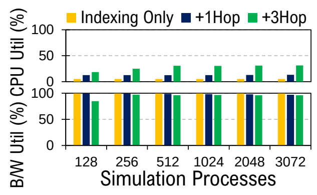
Results. Figure 5(a) compares the efficiency of DeltaFS shuffling under different buffer configurations. Data shuffling in DeltaFS is built atop the Mercury RPC library [51, 52]. Using an 1KB buffer size results in only about 30% utilization of bandwidth to storage, while in contrast, larger buffer sizes like 4KB and 32KB result in significantly higher levels of storage bandwidth utilization. The results highlight the importance of deep message buffering on the overall efficiency of an in-situ indexing pipeline that requires dynamic data placement.

B. *Scalable all-to-all communication via multi-hop routing*

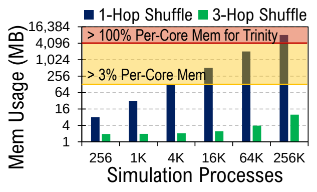
Lesson 2: *All-to-all communication is necessary to dynamically partition data through shuffling, and multi-hop routing can allow that to happen at scale while dramatically reducing communication state per core.*



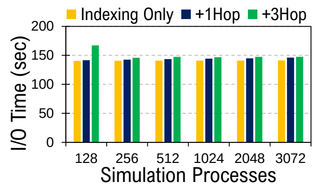
(a) Impact of RPC Buffer Sizes



(b) System Utilization



(c) RPC Buffer Memory Usage



(d) Impact of Shuffle Protocols

Fig. 5. **DeltaFS data shuffling performance from real VPIC simulation jobs on LANL's Trinitite hardware. The largest job used 96 compute nodes, 3,072 cores, and simulated 48 billion particles.**

While deep message buffering helps alleviate the cost of remote communication, the memory cost required to enable it can be prohibitively high. Having each process maintain a direct connection to all other processes requires each connection to buffer multiple kilobytes in order to achieve efficient transfers. As a result, for large-scale simulations comprising hundreds of thousands of processes, the total memory required for buffering at each process quickly rises to gigabytes making this approach infeasible. This was the primary reason our early implementation could not scale, as reviewed in Section II.

One way to limit memory consumption is to route messages via multiple hops. This is achieved by forwarding each message through one or more intermediate shuffle processes before sending the message to its final destination. By merging and sharing communication routes, each process is able to maintain fewer peer connections, and each process's write-back buffers can be filled more quickly. This better bounds the total amount of buffer memory needed at each shuffle process.

As shown in Figure 6, our current multi-hop routing implementation consists of 3 hops. To send a message our protocol first forwards the message to a local *representative* process on the original sender node (i.e., the node containing the source process), and then to a remote representative on the receiver node, which then forwards the message to the final destination. If a message is sent to a process on the same node, the inter-node communication step is bypassed. To reduce communication cost the intra-node communication steps are performed through shared memory. To further improve performance, our implementation has each process on a node act as a representative for a subset of the remote nodes. This reduces the connection state per representative, and distributes communication load evenly among all local processes.

Figure 6 shows an example of three-hop routing with 4 nodes and 16 shuffle processes. Each process is both a shuffle sender and a shuffle receiver. On each node, 3 processes are selected to act as the local representatives for one remote node

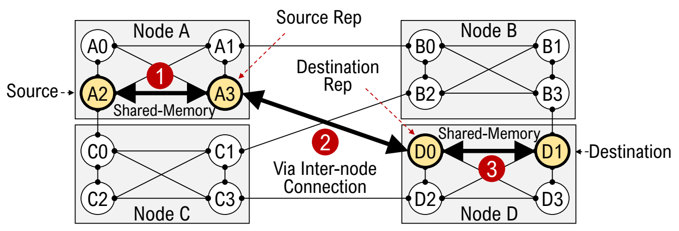


Fig. 6. **Illustration of 3-hop routing across 4 nodes with 4 processes per node. All-to-all communication would require each process to maintain 15 outgoing queues. With 3-hop routing, only 1 remote and 3 local queues are needed. A message is first sent by a sender (A2) to a local representative (A3), and then forwarded to a remote representative (D0), which then forwards the message to the final destination (D1).**

each. Thus, each process only needs to maintain 3 local connections and at most 1 remote connection. In contrast, direct communication would require that each process maintains 15 connections. Given M nodes and N cores per node, three-hop routing only requires M/N remote connections per process on average, while direct all-to-all routing would require $M \times N$. We consider this N^2 reduction important, because it suggests that if the number of cores per node increases faster than the number of nodes in a cluster, the amount of required communication state is further reduced. We expect this to be the case in the future, as higher counts of lightweight or specialized cores become more widespread.

Results. Figure 5(c) compares the total amount of per-process memory that is needed for maintaining the buffer space for direct and 3-hop communication (assuming 32KB buffers per connection). Because each Trinitite compute node has 32 CPU cores ($N = 32$ cores per node), a 3-hop configuration is able to shuffle data using 1,024x fewer remote connections (N^2 reduction) than its 1-hop counterpart. While the per-process memory usage for direct 1-hop communication grows to gigabytes as the simulation scales, less than 16MB of memory is needed in the 3-hop case even for the largest simulation configuration with 256K processes and 8K nodes.

The cost of reduced per-process communication state is the increased work each process has to do to deliver messages. Figure 5(b) compares the system utilization of DeltaFS under direct 1-hop communication, 3-hop communication, as well as a special configuration where data shuffling is omitted (namely indexing only). Despite the higher CPU usage 3-hop runs had, 3-hop routing didn't significantly reduce the overall efficiency of the in-situ indexing pipeline. Given excess CPU cycles are available, 3-hop routing seems to only slightly increase the total I/O time, as shown in Figure 5(d). Overall, the cost of 3-hop routing is small, especially in comparison to the memory savings it provides compared to direct 1-hop communication. On Trinitite, the total CPU utilization is low for all three types of runs, leaving headroom for other computationally intensive classes of in-situ processing.

IV. Streaming Data Indexing

While shuffling data to partitions limits query accesses to a small subset of data, building data indexes at each data partition further speeds up query processing. To achieve this DeltaFS implements a customized LSM-Tree [9] that packs small scientific objects into large log objects for efficient writing while simultaneously enabling such data to be efficiently queried. This section describes our techniques and compares DeltaFS with traditional LSM-Tree implementations.

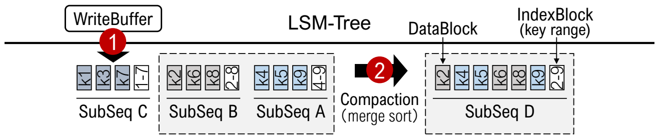


Fig. 7. **Unmodified LSM-Trees use background compactions to bound read latency. Each compaction merges multiple sorted data subsequences into a single, longer subsequence. In the above example, before compaction, searching key K5 needs to check subsequence C, B, and A. After compaction, only subsequence C and D will need to be checked.**

To better interact with storage, LSM-Trees index data by dividing it into *subsequences*, and then individually sorting these subsequences according to a user-specified key [9]. As illustrated in Figure 7, data inserted into an LSM-Tree is first written to a small in-memory write buffer. Once the buffer is full, buffered data is sorted and appended to storage as a sorted run of data blocks. These sorted data blocks then form a data subsequence. A special index block is created for each such subsequence to remember the subsequence's key range. While LSM-Trees have been widely used [8, 50, 61–65], managing data as an embedded library presents unique challenges.

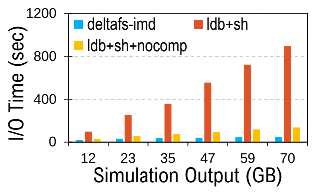
First, an embedded library may only use a small portion of the compute node's memory that the application is willing to relinquish. Consequently, there is often limited space in memory that may be used to coalesce or index data. Second, to minimize interference an embedded library must be ready to yield the CPU and the network when the application restarts its own computation. This limits the total amount of background work the library is able to perform. As embedded I/O libraries can differ drastically from traditional long-running I/O services deployed on dedicated cluster nodes, this section examines and evaluates the way DeltaFS adapts to this unique environment.

A. Fast data access without LSM-Tree compaction

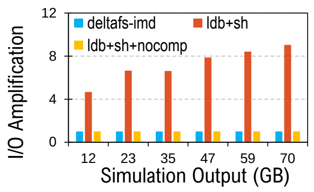
Lesson 3: *Partitioned streaming data indexing allows for fast point and small-range queries without requiring post-write data reorganization operations such as log compaction.*

Recall from Figure 7 that data subsequences generated by LSM-Trees are sorted separately so their key ranges overlap. As such, finding a data element may require checking every subsequence at a certain data partition and reading lots of data. To bound read latency, unmodified LSM-Trees use background *compactions* to incrementally merge new data subsequences into older data subsequences [8, 9, 66]. As shown in Figure 7, by reducing the total number of data subsequences, running compactions allows data to be queried more efficiently.

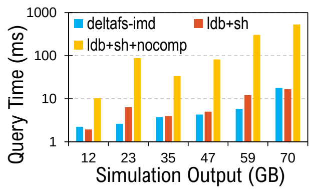
With each compaction reading multiple data subsequences, sorting, and rewriting the same data into a longer subsequence (essentially a merge sort over a subset of data subsequences),



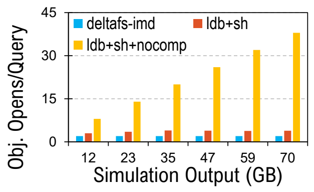
(a) Total I/O Time



(b) I/O Amplification



(c) Average Query Latency



(d) Objects Touched Per Query

Fig. 8. **Results from VPIC simulations on LANL's Trinitite cluster with DeltaFS (deltafs-imd) or LevelDB (ldb) indexing the data. We include LevelDB measurements that perform both shuffle and compaction (sh), and those that do a shuffle without compaction (sh+nocomp).**

compactions can easily consume a large amount of storage bandwidth for reading and rewriting previous data. As storage is typically a bottleneck during application I/O, running background compaction operations would significantly increase the total I/O time, especially when the application dumps massive amounts of data [28, 29]. Moreover, with embedded in-situ processing sufficient CPU cycles to perform the compaction are not always available on compute nodes [38]. Since compaction overwhelms the underlying storage, DeltaFS avoids it and instead uses *filters* [67] and *parallel reads* [44] to achieve fast queries. We discuss these concepts in Section IV-C.

RESULTS. Figure 8 measures the overhead of compaction. Since DeltaFS avoids compaction, we compare DeltaFS with LevelDB [66, 68], a general-purpose LSM-Tree implementation widely used by many HPC storage systems [50, 61, 62, 69, 70]. The LevelDB's LSM-Tree implementation performs compaction. By default, LevelDB allocates an 8MB in-memory space to buffer incoming data. To compare with LevelDB, we configure DeltaFS to match LevelDB's memory usage.

Our experiments used 32 Trinitite compute nodes and one burst-buffer node. We ran VPIC simulations, where data was dynamically partitioned and indexed either by DeltaFS as parallel logs, or by LevelDB using LevelDB's own data formats [68]. We discuss DeltaFS's data formats in the next section. In both cases, data is partitioned by DeltaFS using the all-to-all shuffle mechanism discussed in the previous section (LevelDB does not shuffle data). All data is written to the burst-buffer storage. After each simulation, data is flushed to the underlying Lustre file system for queries. Each query selects a random particle and reads all of its data. We report the average query latency out of 100 such queries, all performed on cold cache.

We compare DeltaFS (`deltafs-imd`) with two LevelDB configurations: LevelDB with an all-to-all shuffle provided by DeltaFS (`ldb+sh`), and LevelDB with DeltaFS shuffle

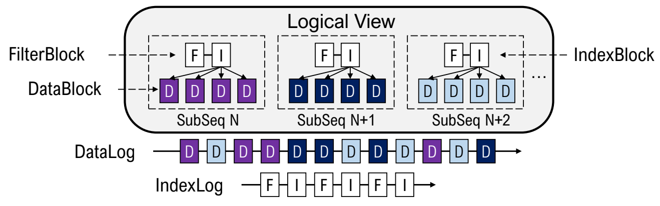


Fig. 9. **In LSM-Trees, each data subsequence consists of a sorted run of data blocks, an index block, and optionally a filter block. The index block stores the locations of the data blocks. To optimize storage accesses, DeltaFS stores indexes and filters in a single per-partition index log for fast retrieval, and stores all data blocks in a per-partition data log to avoid creating lots of small data objects in the underlying storage.**

but additionally modified to not perform any compaction (`ldb+sh+nocomp`). Figure 8(a) shows the total I/O time as a function of the total particle data dumped by a simulation. Due to compaction, LevelDB's total I/O time is substantially higher than both DeltaFS and the LevelDB with compaction disabled. To examine the compaction cost, Figure 8(b) shows the I/O amplification of each simulation trial. I/O amplification measures the ratio of the total amount of data written to and read from the storage, to the total amount of new information persisted. As shown in Figure 8(b), this ratio is 1 if there is no compaction (each write writes new information), and increases significantly if one compacts (due to repeated reads and rewrites). To conclude, because the underlying storage bandwidth available to an application is typically not provisioned to absorb each piece of data multiple times, indexing through compaction (i.e., post-write storage reorganization) is almost always problematic for embedded in-situ indexing.

Figure 8(c) shows the time it takes to query a particle trajectory. Since data is partitioned, each query hits one partition. Within each partition, LevelDB with compaction (`ldb+sh`) answers queries significantly faster than LevelDB with no compaction (`ldb+sh+nocomp`). This is because compaction bounds the total number of places a query needs to check so data can be found with fewer storage lookups. While DeltaFS does not perform compaction, it answers queries almost as fast as the LevelDB that performs compaction (`ldb+sh`). This is because DeltaFS performs more aggressive packing to control the size and the number of the storage objects it creates so each query touches less objects. This allows data to be searched more efficiently, as our next section discusses. LevelDB performs less packing so its queries tend to touch more objects, as shown in Figure 8(d).

B. Maximizing storage throughput via data packing

LESSON 4: *Packing index data based on expected query types allows storage bandwidth to be more fully utilized for subsequent queries.*

To ease compaction [71], LSM-Tree implementations typically store data subsequences in separate files [68]. While this layout allows each subsequence to be added and deleted independently, potentially lots of files are going to be created in the underlying storage [30], causing considerable metadata overhead. Moreover, because indexes associated with different data subsequences are stored with subsequence data in separate files, they are read individually using non-contiguous read operations. This further increases the time spent reading data to satisfy queries [8, 47]. As reading indexes is costly, LSM-Tree implementatons typically cache a subset of indexes in memory to avoid reading them repeatedly [62, 66, 68]. But post-hoc queries typically start with a cold cache, and may not possess sufficient locality to benefit from caching generally.

Since DeltaFS avoids compaction, it uses a more specialized storage layout similar to PLFS [27] with clustered indexes, as shown in Figure 9. Recall from Figure 7 that data subsequences are independently constructed, and each consists of an index block and a sorted run of data blocks. While these blocks are traditionally stored in per-subsequence files [68], DeltaFS remaps these blocks into two types of log files: one per-partition data log holding all the data blocks, and one per-partition index log holding all the index blocks. This creates fewer files in the underlying storage, and allows per-partition indexes to be bulk retrieved using large sequential reads.

RESULTS. Figure 10 shows the benefits of DeltaFS's specialized storage layout by comparison with LevelDB. In each read experiment, a random VPIC particle is selected and its trajectory over time is read. In the smallest configuration, each such trajectory contains 1 timestep. In the largest configuration, each contains 5 timesteps. All queries were executed on a single CPU core. Each query was repeated 100 times, all with cold data caches. We report the average query latency.

We compare DeltaFS with LevelDB (`ldb+sh+nocomp`). This section focuses on the results in the absence of Bloom filters (`bf`). We discuss Bloom filters in Section IV-C. Figure 10(a) shows the query speedup of DeltaFS and LevelDB against the VPIC baseline reader. By partitioning and indexing data, LevelDB answers queries up to 20x faster than the baseline reader, while using only a single core to execute the queries. With a better storage layout, DeltaFS is up to 310x (or 910x with Bloom filters) faster than the baseline reader. This speedup is less than what we reported early [3] because the total amount of particle data generated in this experiment is smaller than that of our earlier results.

Figure 10(b) shows the total number of underlying storage files created by each DeltaFS and LevelDB process. Because DeltaFS remaps data and data indexes into per-partition log objects, the total number of files created by each process in the underlying storage is fixed and does not increase with the simulation size. LevelDB creates a file in the underlying storage for every few megabytes of VPIC's output, so the total number of such files increases linearly with the simulation.

Different storage layouts have profound impact on query processing, as shown in Figure 10(c), 10(d), and 10(e). Because LevelDB does not cluster indexes, a large number of random storage reads are executed in order to collect necessary indexes before data can be located and fetched. As a result, LevelDB's query process was dominated by random index reads, causing it to experience much higher query latency.

C. Using filters and parallelism to accelerate queries

LESSON 5: *Sorting data post-write is unnecessary, and we can use parallelism and filters to overcome the lack of contiguity in data storage.*

While clustered per-partition indexes can be efficiently retrieved, our coarse-grained indexes cannot precisely locate each data element. To explain this with an example, consider each data subsequence as all the word-definition pairs within an English dictionary (e.g. Webster's). The dictionary is sorted alphabetically and has an index showing the first word of each page. In this case, the index describes where to find a word but it cannot indicate whether a word exists in the dictionary. One needs to check a specific page (i.e., a specific data block) in order to make sure. In our case, without compaction there are potentially many data subsequences (English dictionaries) within each data partition. To find a specific data element (an English word), all these subsequences will have to be checked, one-by-one, until the target data element is found.

To avoid reading many data subsequences, a Bloom filter [67] is created for each subsequence and is stored with the subsequence's indexes. Each Bloom filter is a probabilistic data structure capable of checking whether a data element may, or must not, exist in a data subsequence. With Bloom filters, potentially many data subsequences can be skipped that are known to not contain a certain data element. This bounds the total number of data lookups each query has to perform. The cost of adding Bloom filters is the extra I/O that is needed to persist filter information, which is typically small (about 3-4% in our experiments) in comparison to the total data size.

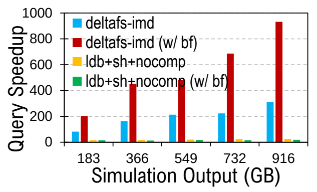
To answer queries even faster, DeltaFS reads data elements in parallel. For trajectory reads this means reading multiple data points in time concurrently. DeltaFS achieves this by sending multiple asynchronous reads to the underlying storage. The cost of sending multiple reads in parallel is the increased memory for buffering and sorting partial results, which can be bounded by setting a maximum query concurrency level.

RESULTS. Figure 10(e) shows the average query latency for both DeltaFS and LevelDB runs. Because DeltaFS sends multiple read requests in parallel, its query latency increases slowly with the size of the query (from reading 1 timestep to 5 timesteps). While using Bloom filters substantially reduced the query latency for DeltaFS, for LevelDB the reduction was largely offset by the excessive random storage reads LevelDB had to perform to readback the filters during the execution of each of its queries. Since LevelDB does not cluster its filters, using Bloom filters only reduces its query performance.

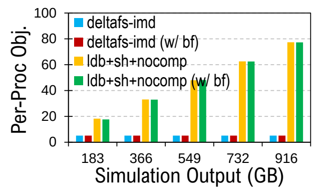
D. Avoiding computation and communication bottlenecks

LESSON 6: *Overlapping data shuffling and computation with storage I/O allows one to more fully utilize available storage write bandwidth.*

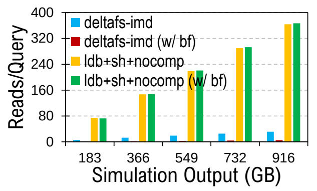
An important goal in our work is to ensure that any communication or computation we perform does not prevent full utilization of the available storage bandwidth and avoids



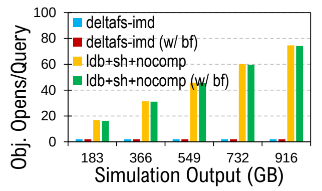
(a) Query Speedup Against VPIC



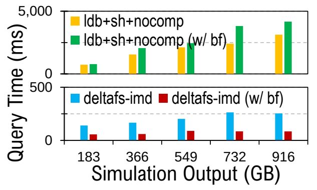
(b) Total Objects Created Per-Process



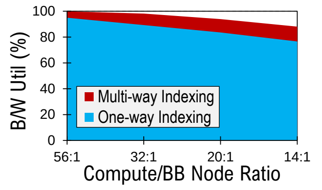
(c) Random Storage Reads Per Query



(d) Objects Touched Per Query



(e) Average Query Latency



(f) Impact of Indexing Concurrency

Fig. 10. **Results from real VPIC simulations on LANL's Trinitite cluster comparing VPIC (baseline), DeltaFS (deltafs-imd), and LevelDB (ldb)'s performance in handling small-range queries. While VPIC queries were executed on 1,024 CPU cores, all DeltaFS and LevelDB queries were executed on a single CPU core. A single-core DeltaFS reader may send multiples read requests in parallel.**

extending the simulation's I/O phase significantly. One challenge we faced was trying to maintain a steady data flow in the communication between the shuffle component that partitions the data and the indexing component that indexes data. The indexing component uses a background thread for indexing and writing data to the underlying storage, but originally the background RPC thread used by the shuffle component was coded such that it directly inserted data into the indexing component as part of its message handling procedure. As storage is usually slower than the network, insertions made by the RPC thread into the indexing component were often blocked waiting for the storage to finish writing and allow new insertions. Blocking the RPC thread prevented timely handling of important network events, including forwarding messages to other processes and sending replies. Delaying both forwarding and replies caused back-pressure that unnecessarily slowed progress. To alleviate this inefficiency, a delivery queue was added between the shuffle and indexing components, as shown in Figure 11. Messages received by the background RPC thread are first put into this delivery queue. A separate delivery thread is responsible for inserting data into the indexing component. This ensures that the RPC thread is always available to service network requests.

Another source of inefficiency was the lack of parallelism in our indexing component. Early implementations had data indexing and storage I/O serialized within a single background

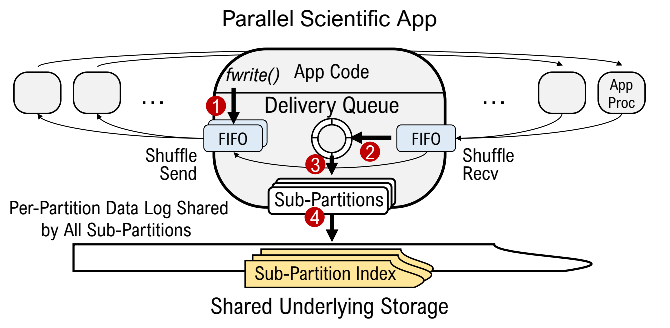


Fig. 11. **Updated DeltaFS in-situ indexing pipeline design with a new delivery queue structure, and a multi-way indexing mechanism. The original pipeline design is shown in Figure 2.**

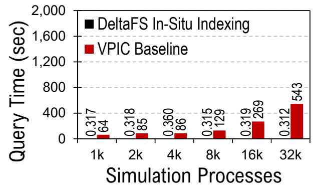
thread. As a result, this indexing component would switch between data indexing and storage I/O, creating a series of gaps between I/O operations. These gaps generate idle I/O periods, under utilizing the underlying storage. To prevent such gaps, we changed our implementation to have multiple sub-partitions so that each sub-partition can be indexed independently, as shown in Figure 11. We refer to this as *multi-way data indexing*. It allows data in one sub-partition to be indexed while data in another sub-partition is being written to the storage. Thus, at the time an I/O operation is completed, the data for the next I/O operation is already indexed so it can be written to storage immediately. This prevents I/O gaps and allows us to approach full bandwidth utilization.

RESULTS. Figure 10(f) shows the importance of overlapping indexing computation with storage I/O. DeltaFS achieves this through multi-way data indexing. Results show that without such optimization (1-way indexing) DeltaFS loses up to 10% I/O efficiency. Such reduction increases as compute to burst-buffer ratio decreases (i.e., as there is more burst-buffer bandwidth available to a parallel scientific job).

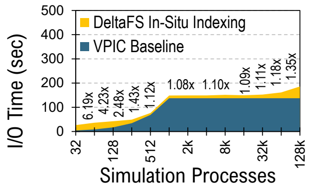
V. FINAL SCALING RESULTS

To show the performance of DeltaFS enabled by our new techniques, we did the same experiments as described in Section II-C but this time with a much larger scale on LANL's Trinity machine instead of the Trinitite. Our biggest run used 131,072 CPU cores, 4,096 Trinity compute nodes, simulated 2 trillion particles, and generated 470TB of data. Recall from Section II-C that we ran a real VPIC configuration both with and without DeltaFS for each experiment. For VPIC baseline runs, the simulation wrote one file per process. For DeltaFS runs, the simulation wrote into an Indexed Massive Directory. Across all runs, simulation data was first written to a burst-buffer storage tier and was later staged out to the Trinity's underlying Lustre file system. The compute node to burst-buffer node ratio was fixed at 32 to 1.

Recall also from Section II-C that after each simulation, queries were executed from the underlying Lustre file system and each query targets a random particle and reads all of its data. On Trinity, each DeltaFS query was repeated 1,000 times, all starting with a cold data cache. We report the *median* query



(a) LANL Trinity Query Performance



(b) LANL Trinity Write Performance

Fig. 12. **Results from real VPIC simulation jobs on LANL's Trinity hardware. Our biggest job used 4,096 compute nodes, 131,072 CPU cores, simulated 2 trillion particles, and wrote 96TB of data per timestep. While our baseline VPIC reader used all the CPU cores to search particles in parallel, all DeltaFS queries were executed on a single CPU core.**

latency. Each VPIC baseline query reads the entire simulation output and was repeated up-to 2 times. DeltaFS queries were executed on a single CPU core while baseline queries used the number of simulation processes to search particles.

Figure 12(a) shows the read performance. On Trinity, a single-core DeltaFS reader answers queries up-to 1,740x faster than the VPIC baseline reader that uses all the CPU cores. Again, this is because without an index, finding each particle's trajectory requires reading all the particle data so the baseline query time is bounded by the underlying storage bandwidth. Because the underlying Lustre file system on Trinity has higher bandwidth, the slowdown of VPIC is less on Trinity than that on Trinitite as we reported previously in Section II-C. DeltaFS builds particle indexes in-situ, so it can locate per-particle information much more quickly and keeps query latency within 500ms. The latency was higher on Trinity than on Trinitite because Trinity is a much larger cluster with more concurrent jobs so the Lustre on Trinity tends to be busier.

With techniques discussed in this paper, DeltaFS is able to scale much further than its previous implementation. As shown in Figure 12(b), on Trinity DeltaFS reliably built data indexes with up-to 131,072 CPU cores. While in the beginning the jobs were too small to saturate the burst-buffer storage, starting from the sixth run the jobs began to bottleneck on the storage, and we see a modest DeltaFS slowdown of about 10%. This is less than that of our previous implementation, which had a slowdown of about 15%. For the last 2 runs, the job sizes are deliberately increased to demonstrate the limitations of our scaling techniques. At 131,072 processes the increased all-to-all communication overhead due to data shuffling had caused the overall I/O overhead DeltaFS added to the simulation to increase from 10% to 35%, suggesting more techniques are needed for more efficient data movement to better support in-situ processing beyond hundreds of thousands of cores.

VI. Conclusion

In this paper we described a set of techniques that enable the scaling of DeltaFS to more than a hundred thousand processes. The lessons we learned designing and applying these techniques can be used to address scalability challenges in a variety of in-situ and analytics middleware. We categorized our techniques as providing either improvements to the scalable shuffling of data or improving the efficiency of data indexing.

Latency hiding and efficient bandwidth utilization are critical for scalable shuffling. Our analysis shows that careful buffer management is the key to keeping latency low and bandwidth high. Buffers must be large enough to make efficient use of the network without being so large as to waste memory. In our configuration 32KB buffers are sufficient, but we anticipate that larger buffers may be required with lightweight cores. To slow the increase in the number of buffers as the system scales we introduced our 3-hop all-to-all communication technique. By limiting the number of off-node connections per process, we believe the applicability of the 3-hop technique will increase for supercomputer platforms if intra-node parallelism increases faster than inter-node parallelism.

Our indexing techniques demonstrate that on-disk data reorganization (e.g. compaction) is not necessary if the dominant access regimes are point and small-range queries. In particular, clustered indexes can be efficiently constructed and accessed on modern HPC platforms, and coarse-grain subsequence filters are able to balance efficient searching with optimal storage system access. We also note the importance of effectively overlapping communication and indexing with storage access, ensuring that the storage system is idle as little as possible during the output phase.

More generally, we believe embedded in-situ indexing provides a compelling advantage in its ability to scavenge temporarily available resources to improve the efficiency of post-hoc analysis. Although embedded in-situ processing introduces scalability challenges, we believe that these challenges are manageable. The techniques described in this paper demonstrate efficient scaling to a hundred thousand processes. We believe that additional techniques exist to improve embedded in-situ scaling even further. In addition to further scaling techniques, it is clear to us that improving the performance of queries when partitioning functions cannot provide an evenly balanced distribution is important to furthering the adoption of our techniques. Support for multiple simultaneous indexes to enable multivariate analysis is also important to diverse types of scientific analysis. Adding this capability to our embedded in-situ pipeline will enable new classes of scientific applications to leverage DeltaFS.

Acknowledgement

The authors would like to thank Suren Byna, John Bent, anonymous reviewers, and our shepherd for their comments on the paper, and Phil Carns, Jerome Soumagne, Shane Snyder, and Robert Ross for their guidance on the *Mochi* software stack. This material is based on work supported in part by the US DOE and Los Alamos National Laboratory, under contract number DE-AC52-06NA25396 subcontract 394903 (IRHPIT), and by the US DOE, Office of Science, Advanced Scientific Computing Research (ASCR) under award number DE-SC0015234. We also thank the member companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, and Western Digital).

References

[1] *Exascale computing project (ECP)*, https://www.exascaleproject.org/.

[2] *ANL aurora*, https://www.alcf.anl.gov/alcf-aurora-2021-early-science-program-data-and-learning-call-proposals.

[3] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, and F. Guo, "Software-defined storage for fast trajectory queries using a deltafs indexed massive directory," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*, 2017, pp. 7–12. DOI: 10.1145/3149393.3149398.

[4] *APEX workflows*, https://www.nersc.gov/assets/apex-workflows-v2.pdf, Mar. 2016.

[5] *LANL trinity*, http://www.lanl.gov/projects/trinity/.

[6] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 7, 2008.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*, 2006, pp. 205–218.

[8] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. DOI: 10.1145/1773912.1773922.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996. DOI: 10.1007/s002360050048.

[10] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 12)*, 2012, 59:1–59:12. DOI: 10.1109/SC.2012.92.

[11] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015 001, 2009.

[12] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*, 2009, 40:1–40:12. DOI: 10.1145/1654059.1654100.

[13] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: Reading patterns for extreme scale science io," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC 11)*, 2011, pp. 49–60. DOI: 10.1145/1996130.1996139.

[14] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne, "Parallel index and query for large scale data analysis," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*, 2011, 30:1–30:11. DOI: 10.1145/2063384.2063424.

[15] J. Chou, K. Wu, and Prabhat, "Fastquery: A parallel indexing system for scientific data," in *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*, 2011, pp. 455–464. DOI: 10.1109/CLUSTER.2011.86.

[16] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*, 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232369.

[17] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at los alamos national lab," in *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*, 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376.

[18] J. Bent, B. Settlemyer, and G. Grider, "Serving data to the lunatic fringe: The evolution of HPC storage," *USENIX ;login:*, vol. 41, no. 2, Jun. 2016.

[19] A. Nisar, W. k. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *Proceedings of the 2008 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 08)*, 2008, pp. 1–12. DOI: 10.1109/SC.2008.5214358.

[20] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "Predata - preparatory data analytics on peta-scale machines," in *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS 10)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454.

[21] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*, 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178.

[22] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 12)*, 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[23] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*, 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232382.

[24] J. Lofstead and R. Ross, "Insights for exascale IO apis from building a petascale IO api," in *Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 13)*, 2013, 87:1–87:12. DOI: 10.1145/2503210.2503238.

[25] R. B. Ross, *From file systems to services: Changing the data management model in hpc*, Presented at the Salishan Conference on High-Speed Computing, http://www.mcs.anl.gov/research/projects/mochi/files/2016/11/ross_salishan-2016-16x9.pdf, 2016.

[26] P. Carns, *Building blocks for user-level hpc storage systems*, Presented at Dagstuhl Seminar: Challenges and Opportunities of User-Level File Systems for HPC, http://www.mcs.anl.gov/research/projects/mochi/files/2016/11/carns-dagstuhl-2017-v3.pdf, 2017.

[27] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*, 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081.

[28] S. Byna, A. Uselton, D. K. Prabhat, and Y. He, "Trillion particles, 120,000 cores, and 350 tbs: Lessons learned from a hero i/o run on hopper," in *Cray User Group (CUG)*, https://cug.org/proceedings/cug2013_proceedings/includes/files/pap107-file2.pdf, 2013.

[29] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol, "Tuning parallel i/o on blue waters for writing 10 trillion particles," in *Cray User Group (CUG)*, https://cug.org/proceedings/cug2015_proceedings/includes/files/pap120-file2.pdf, 2015.

[30] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Ottawa Linux Symposium (OLS 03)*, 2003, pp. 380–386.

[31] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, vol. 2000, no. 80es, Nov. 2000.

[32] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008, 2:1–2:17.

[33] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002, pp. 231–244.

[34] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*, 2011, pp. 1–11. DOI: 10.1145/2063384.2063409.

[35] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock, "Trilinos i/o support trios," *Sci. Program.*, vol. 20, no. 2, pp. 181–196, Apr. 2012. DOI: 10.1155/2012/842791.

[36] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER 12)*, 2012, pp. 155–163. DOI: 10.1109/CLUSTER.2012.26.

[37] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 10)*, 2010, pp. 1–12. DOI: 10.1109/SC.2010.28.

[38] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 13)*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503279.

[39] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "Flexio: I/o middleware for location-flexible scientific data analytics," in *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 13)*, 2013, pp. 320–331. DOI: 10.1109/IPDPS.2013.46.

[40] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich io methods for portable high performance io," in *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 09)*, 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161052.

[41] U. Ayachit, A. Bauer, E. P. N. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. E. Jansen, B. Loring, Z. Lukić, S. Menon, D. Morozov, P. O'Leary, R. Ranjan, M. Rasquin, C. P. Stone, V. Vishwanath, G. H. Weber, B. Whitlock, M. Wolf, K. J. Wu, and E. W. Bethel, "Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures," in *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*, 2016, 79:1–79:12.

[42] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, Mar. 2006. DOI: 10.1145/1132863.1132864.

[43] J. Kim, H. Abbasi, L. Chacón, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu, "Parallel in situ indexing for data-intensive computing," in *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*, 2011, pp. 65–72. DOI: 10.1109/LDAV.2011.6092319.

[44] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 133–148.

[45] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 17)*, 2017, pp. 79–94. DOI: 10.1145/3035918.3064054.

[46] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "Slimdb: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2037–2048, Sep. 2017. DOI: 10.14778/3151106.3151108.

[47] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 71–82.

[48] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 97)*, 1997, pp. 16–25.

[49] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 17–30.

[50] H. N. Greenberg, J. Bent, and G. Grider, "MDHIM: A parallel key/value framework for HPC," in *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015, pp. 10–10.

[51] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER 13)*, 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702617.

[52] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," in *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP 83)*, 1983, pp. 3–. DOI: 10.1145/800217.806609.

[53] P. Carns, W. Ligon, R. Ross, and P. Wyckoff, "Bmi: A network abstraction layer for parallel i/o," in *Proceedings of the 2005 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 05)*, 2005, pp. 1–8. DOI: 10.1109/IPDPS.2005.128.

[54] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The common communication interface (cci)," in *Proceedings of the 2011 IEEE Annual Symposium on High Performance Interconnects (HOTI 11)*, 2011, pp. 51–60. DOI: 10.1109/HOTI.2011.17.

[55] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *Proceedings of the 2015 IEEE Annual Symposium on High-Performance Interconnects (HOTI 15)*, 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19.

[56] *Mochi ch-placement*, https://xgitlab.cels.anl.gov/codes/ch-placement.

[57] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *Proceedings of the 2004 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 04)*, 2004, pp. 4–. DOI: 10.1109/SC.2004.22.

[58] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC 97)*, 1997, pp. 654–663. DOI: 10.1145/258533.258660.

[59] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP 07)*, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281.

[60] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," Cray Inc., Tech. Rep. WP-Aries01-1112, Nov. 2012, http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[61] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2Nd International Workshop on Petascale Data Storage (PDSW 07)*, 2007, pp. 35–44. DOI: 10.1145/1374596.1374606.

[62] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*, 2014, pp. 237–248. DOI: 10.1109/SC.2014.25.

[63] Q. Zheng, K. Ren, and G. Gibson, "BatchFS: Scaling the file system control plane with client-funded metadata servers," in *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*, 2014, pp. 1–6. DOI: 10.1109/PDSW.2014.7.

[64] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "DeltaFS: Exascale file systems scale better without dedicated servers," in *Proceedings of the 10th Parallel Data Storage Workshop (PDSW 15)*, 2015, pp. 1–6. DOI: 10.1145/2834976.2834984.

[65] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson, "ShardFS vs. IndexFS: Replication vs. caching strategies for distributed metadata management in cloud storage systems," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*, 2015, pp. 236–249. DOI: 10.1145/2806777.2806844.

[66] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 4:1–4:26, Jun. 2008. DOI: 10.1145/1365815.1365816.

[67] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. DOI: 10.1145/362686.362692.

[68] *Leveldb*, https://github.com/google/leveldb/.

[69] K. Ren and G. Gibson, “TABLEFS: Enhancing metadata efficiency in the local file system,” in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 145–156.

[70] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, “Locofs: A loosely-coupled metadata service for distributed file systems,” in *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*, 2017, 4:1–4:12. DOI: 10.1145/3126908.3126928.

[71] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP 91)*, 1991, pp. 1–15. DOI: 10.1145/121132.121137.

Appendix A
Artifact Description Appendix: Scaling Embedded In-Situ Indexing With DeltaFS

A. Abstract

This section describes the steps for reproducing some of the experiments of DeltaFS presented in this paper. **The DeltaFS artifact associated with this paper comprises multiple DeltaFS in-situ indexing related codebases, simulation decks, and build instructions that can be used to run DeltaFS in-situ indexing with the LANL's VPIC application.** The DeltaFS code, as well as all related material, is available at `URL:` https://github.com/pdlfs/deltafs-umbrella.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** The VPIC application is configured to perform various *collisionless magnetic reconnection* simulations
- **Program:** C and C++ binaries with `pthread` and MPI (tested with Open MPI 1.6.5, MPICH 3.2, and Cray MPI 7.7.0)
- **Compilation:** GNU `g++` (tested with 4.6, 4,8, 5, 6, 7, and 8), Clang `clang++` (tested with 3.8), or Intel `icpc` (tested with 17.0.4)
- **Binary:** VPIC (under MPI), DeltaFS C++ library (dynamically loaded via `LD_PRELOAD`)
- **Data set:** VPIC input deck (included with the VPIC sources)
- **Run-time environment:** Ubuntu Linux (tested with 14.04 and 16.04) or the Cray programming environment (tested with 2.5.13), MPI (tested with Open MPI 1.6.5, MPICH 3.2, and Cray MPI 7.7.0), and CMake (tested with 3.10.3).
- **Output:** VPIC particle data, various performance statistics
- **Experiment workflow: 1)** download the artifact; **2)** compile the code; **3)** run the test scripts; **4)** observe the results
- **Publicly available?: Yes** (under a `BSD` license)

2) How software can be obtained: Use `URL:` https://github.com/pdlfs/deltafs-umbrella to obtain the `deltafs-umbrella` all-in-one package. This special package features a highly-automated process that downloads, builds, and installs DeltaFS, VPIC, and all their dependencies in a single configurable step. This package further includes the scripts we used to perform our experiments, and can be used to reproduce our results.

3) Hardware dependencies: None, but the number of cores and amount of available memory will determine the size of VPIC simulations.

4) Software dependencies: Our evaluation software requires `C`, `C++`, `MPI`, `pthreads`, `make`, `cmake`, `autoconf` `automake`, `libtool`, `pkg-config`, `libpapi-dev`, `libnuma-dev`, `libboost-dev`, `libltdl-dev`, `bash`, and `perl` (the latter two are for the scripts that run our experiments). These software packages must be installed before using `deltafs-umbrella`. On an Ubuntu Linux, these packages can all be installed through the Ubuntu's `apt-get` system. On a Cray programming environment, these packages can be dynamically loaded via the Cray's `module` system. See Appendix A-C for details.

In addition to the VPIC and DeltaFS code, we also make use of the following public codebases: `libch-placement`, `ssg`, `mercury-rpc`, and `bmi`. These codebases are developed and maintained by the Argonne National Laboratory's Mathematics and Computer Science Division, and the HDF5 group. `deltafs-umbrella` is responsible for downloading, compiling, and installing these software packages so they do *not* need to be prepared before-hand.

5) Datasets: The VPIC input decks used for our experiments are included in the VPIC source code repository. They are publicly available at `URL:` https://github.com/pdlfs/vpic. Check the `decks` folder for details. Compiling and configuring the input decks are part of `deltafs-umbrella`'s automation process so they do *not* need to be handled manually.

C. Installation

First, find an Ubuntu box (14.04 LTS or later) and install all software prerequisites using the following commands.

```
sudo apt-get install gcc g++ make pkg-config \
 autoconf automake libtool libpapi-dev libnuma-dev \
 libboost-dev libmpich-dev mpich cmake
sudo apt-get install git
```

Next, clone the `deltafs-umbrella` git repository available at `URL:` https://github.com/pdlfs/deltafs-umbrella.git. After that, create a build directory, change directory to it, and configure the system using `cmake`.

```bash
#!/bin/bash

git clone \
 https://github.com/pdlfs/deltafs-umbrella.git
cd deltafs-umbrella
git checkout "1.0-rc45"
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/deltafs/rc45 \
 -DCMAKE_BUILD_TYPE=RelWithDebInfo \
 -DUMBRELLA_BUILD_TESTS=OFF \
 -DUMBRELLA_SKIP_TESTS=ON \
 -DMERCURY_NA_INITIALLY_ON="bmi;sm" \
 -DMERCURY_POST_LIMIT=OFF \
 -DMERCURY_CHECKSUM=ON \
 ..
make
```

This `make` process takes about 10 minutes to run. It downloads, configures, compiles, and installs all necessary parts of the system in the installation directory (specified via `CMAKE_INSTALL_PREFIX`). This includes the DeltaFS code, the VPIC application, the input decks for the experiments, the helper test scripts, as well as all third-party dependencies.

To run experiments across multiple compute nodes, make sure `CMAKE_INSTALL_PREFIX` is accessible to all these nodes. For single-node tests, `CMAKE_INSTALL_PREFIX` may be placed on a local filesystem.

D. Experiment workflow

After following the installation steps above, the scripts for running our experiments will be in the installation directory (specified via `CMAKE_INSTALL_PREFIX` in the previous step) under the `scripts` subdirectory. To run experiments, use the `vpicexpt_gen.pl` script to generate the final test scripts.

Each our test performs a real VPIC simulation with DeltaFS in-situ indexing the simulation's output. For a minimal test, use the following test options.

```
#!/bin/bash

cd ${CMAKE_INSTALL_PREFIX}/scripts
mkdir try-1
cd try-1

ip_prefix="127.0"   # FIXME

std="--experiment minimal --iterations 1 \
 --tag rc45-try1 --tests deltafs \
 --ipsubnet=${ip_prefix} \
 --env XX_IGNORE_DIRS=fields:hydro:rundata:names \
 --env XX_BYPASS_CH=1 \
 --env XX_SKIP_PAPI=1 \
 --env XX_SKIP_SAMP=0 \
 --env XX_SKIP_SORT=1 \
 --env XX_NO_PRE_FLUSH_WAIT=0 \
 --env XX_NO_PRE_FLUSH_SYNC=0 \
 --env XX_NO_PRE_FLUSH=0 \
 --env XX_HG_PROTO=bmi+tcp \
 --env XX_BG_PAUSE=1"

for r in 1 2 3; do
 eval ../vpicexpt_gen.pl ${std} --run ${r} .
done
```

This will give us three test scripts named in the form of `rc45-try1-minimal-*-deltafs.sh`. More specifically, `run 1` will use 1 node and 4 processes per node for a total of 4 processes, `run 2` will use 2 nodes and 4 processes per node for a total of 8 processes, and `run 3` will use 4 nodes and 4 processes per node for a total of 16 processes.

Run the generated test scripts to kick off the test runs. Each test run consists of a write phase that generates particle dumps and a read phase that performs queries on one or more particle trajectories. Set the `JOBDIRHOME` environmental variable (default: `$HOME/jobs`) to control where to put the job output. In addition, use the `JOBHOSTS` environmental variable (default: `localhost`) to specify the nodes to run the tests.

In some situations, a compute node may possess multiple network interface cards (i.e., NICs) and has multiple IP addresses. To ensure a fixed IP subnet is used for all DeltaFS instances, set the `ip_prefix` as shown above. For single-node tests, it suffices to set `ip_prefix` to `"127.0"`.

```
#!/bin/bash

cd ${CMAKE_INSTALL_PREFIX}/scripts/try-1

# FIXME
export JOBDIRHOME="$HOME/deltafs-jobs"
export JOBHOSTS="h0,h1,h2,h3"

# THIS KICKS OFF RUN 3
./rc45-try1-minimal-3-deltafs.sh 2>&1 \
  | tee JOBOUT.txt
```

The example shown above uses four compute nodes and does `run 3`. For single-node tests, `JOBHOSTS` may be set to `localhost` but only `run 1` can be launched. A copy of the job's `stdout` and `stderr` prints will be logged to `JOBOUT.txt` and can be reviewed after the job.

Specially configured for a minimal test, each such job takes less than one minute to finish. Each run performs 50 timesteps, 2 dumps, simulating approximately 40K particles per process, and generating 2MiB of particle data per process per timestep.

E. Evaluation and expected result

After each job, the job's `JOBOUT.txt` can be parsed to obtain performance results. This is achieved through the `vpic_report.sh` script in the installation directory (specified via `CMAKE_INSTALL_PREFIX` in the previous step) under the `scripts` subdirectory, as shown below.

```
+ ./$CMAKE_INSTALL_PREFIX/scripts/vpic_report.sh \
 JOBOUT.txt
Parsing results from JOBOUT.txt ...
exp=minimal, run=1, node=1, ppn=4,
bb_nodes=0 (mode=off), test=deltafs, mpi=[]
2 lines of "-INFO- all done"...
OK!

Extracting important results...

TOTAL IO TIME (0 BB NODES)
+ 0.217708
+ 0.275676
---------------
= 0.493384 (secs)

TOTAL OUTPUT SIZE
15850810 bytes
= 0.000 TiB

CPU UTIL (usr time + sys time)
> 92.21%
> 95.80%
---------------
= 94.00%

QUERY LATENCY
1.026 (med: 0.931000, min: 0.806, max 1.429) ms

Done
```

F. Experiment customization

To compile code on a Cray programming environment, use the following commands to launch the `make` process.

```
#!/bin/bash

export CRAYPE_LINK_TYPE="dynamic"
export CRAYOS_VERSION=6

module unload craype-hugepages2M
module load craype-haswell PrgEnv-intel cmake

env CC=cc CXX=CC cmake \
 -DCMAKE_PREFIX_PATH="${PAT_BUILD_PAPI_BASEDIR}" \
 -DCMAKE_INSTALL_PREFIX=${HOME}/deltafs/r45 \
 -DCMAKE_BUILD_TYPE=RelWithDebInfo \
 -DUMBRELLA_BUILD_TESTS=OFF \
 -DUMBRELLA_SKIP_TESTS=ON \
 -DMERCURY_NA_INITIALLY_ON="bmi;sm" \
 -DMERCURY_POST_LIMIT=OFF \
 -DMERCURY_CHECKSUM=ON \
 ..
make
```

Check the `README` files in the root directory of the `deltafs-umbrella` repository for more information.