Adaptive Spatially Aware I/O for Multiresolution Particle Data Layouts

Will Usher*‖, Xuan Huang*, Steve Petruzza*†, Sidharth Kumar‡, Stuart R. Slattery§, Sam T. Reeve¶, Feng Wang*, Chris R. Johnson* and Valerio Pascucci*
*SCI Institute, University of Utah. †Utah State University. ‡University of Alabama, Birmingham.
§Oak Ridge National Laboratory. ¶Lawrence Livermore National Laboratory. ‖will@sci.utah.edu

***Abstract*—Large-scale simulations on nonuniform particle distributions that evolve over time are widely used in cosmology, molecular dynamics, and engineering. Such data are often saved in an unstructured format that neither preserves spatial locality nor provides metadata for accelerating spatial or attribute subset queries, leading to poor performance of visualization tasks. Furthermore, the parallel I/O strategy used typically writes a file per process or a single shared file, neither of which is portable or scalable across different HPC systems. We present a portable technique for scalable, spatially aware adaptive aggregation that preserves spatial locality in the output. We evaluate our approach on two supercomputers, Stampede2 and Summit, and demonstrate that it outperforms prior approaches at scale, achieving up to 2.5× faster writes and reads for nonuniform distributions. Furthermore, the layout written by our method is directly suitable for visual analytics, supporting low-latency reads and attribute-based filtering with little overhead.**

***Index Terms*—Parallel I/O, Load Balancing**

I. Introduction

The continuing growth in computational power available on high-performance computing systems has enabled scientists to perform increasingly detailed simulations, modeling physical phenomena with greater accuracy. As the resolution of simulations increases, a proportionally larger amount of data is produced. This growth in simulation size, combined with the growing gap between FLOPS and I/O bandwidth on HPC systems, poses a significant challenge to existing I/O strategies. Prior work has proposed strategies to address these challenges for uniform grids [1]–[5] and AMR [6]. However, relatively little work has focused on particle data.

Particles are widely used in the simulation community to model dynamic and nonuniformly distributed media [7]–[11], due to their ability to handle large dynamic ranges and contact discontinuities often found in cosmology, molecular dynamics (MD), or free-surface and disperse multiphase fluid dynamics. Particle populations in these simulations often span large ranges of space, with localized groups representing, e.g., clustered galactic masses, atomic features, or fluid droplets. Although using particles can greatly facilitate the ability to perform computations when modeling such phenomena, their unstructured nature and nonuniform distributions pose additional challenges to the I/O system.

In algorithms where particles represent the primary unit of computational work, such as classical MD, care is taken to periodically rebalance them via geometric methods, e.g., recursive coordinate bisectioning, resulting in a correspondingly balanced I/O workload. However, particles often do not represent the primary unit of work, or a significant fraction of work is shared by a different algorithm. Examples of such simulations include disperse multiphase Lagrangian-Eulerian techniques [12] and partitioned multiphysics methods that couple particle and grid-based methods [13]. In such simulations, particles remain an important part of the computation and analysis, but move dynamically through the domain, imbalancing the I/O workload as the simulation evolves.

To portably achieve high-bandwidth writes, current parallel I/O libraries use two-phase I/O [6], [14], [15] or subfiling [16]–[18]. Two-phase I/O approaches transfer data across the network to a selected subset of aggregator ranks that are responsible for writing the data to disk. Subfiling is a similar technique that controls the total number of output files by combining data from multiple ranks. Both methods reduce I/O contention and overhead to achieve high-bandwidth writes. However, neither approach ensures the data output to each file contains a spatially coherent subregion. Thus, postprocess visualization and analysis tasks, which often work on spatial subregions, may be forced to perform suboptimal reads on the data.

Visualization tasks are often performed on single workstations or small clusters, with significantly less compute and memory capability than was used to run the simulation. Data access is driven by user exploration, often involving spatial or attribute subset queries [17], [19]–[25]. Support for low-latency multiresolution reads and Level of Detail (LOD) are also desirable [19]–[21] to enable visualization on low-power devices, or while additional data is loaded. However, simulations typically output data as flat arrays without the metadata or hierarchies required to support such queries, requiring a lengthy postprocess conversion step to transform the data to the visualization format.

Kumar et al. proposed spatially aware two-phase I/O strategies for uniform grids [5] and AMR [6], capable of outputting the data directly in a layout suited to visualization [26], without sacrificing parallel write or read performance. Kumar et al. recently extended this approach to particle data [27]; however, their aggregation strategy is based on an adjustable uniform grid and assumes a uniform density distribution of particles. In contrast, our approach supports arbitrary nonuniform particle distributions with varying density or sparsity that often occur in simulations.

We present a scalable technique for spatially aware adaptive aggregation of particle data for two-phase I/O. We group ranks

for aggregation using a k-d tree decomposition of the domain to assign a similar number of particles to each aggregator. On each aggregator, we construct a low-overhead multiresolution layout on the particles that is directly suitable for postprocess visualization and analysis. We demonstrate the scalability and portability of our I/O strategy on uniform and nonuniform particle distributions on two supercomputers, and evaluate its suitability for low-latency multiresolution visualization queries. Our contributions are:

- A spatially adaptive approach for parallel I/O of multiresolution particle data layouts supporting arbitrary nonuniform particle distributions;
- A novel combination of k-d trees and bitmap indexing for multiresolution attribute and spatial queries on particle data;
- In situ construction of the proposed data structure during spatially adaptive I/O with little overhead; and
- Empirical scalability and portability studies on Stampede2 and Summit up to 24k and 43k cores, respectively.

II. Related Work

We first review relevant work on parallel I/O, with a focus on particle data (Section II-A), and then briefly discuss visualization- and analysis-focused data layouts (Section II-B).

A. Parallel I/O

Organizing and efficiently accessing large-scale grid-based structured data has always been the aim of several high-level I/O libraries. The most prominent examples are Parallel HDF5 [1], Parallel NetCDF [3] (PnetCDF), and ADIOS [4]. These libraries typically store data in row-major blocks. Although HDF5, PnetCDF, and ADIOS are general purpose, robust, and achieve high-write bandwidth, they are less suited to the read access patterns required for interactive, exploratory analysis of large data sets. Parallel IDX (PIDX) [6] is an I/O library that writes data directly in a cache-oblivious, multiresolution data layout using a spatially aware two-phase I/O strategy. The layout output by PIDX supports low-latency multiresolution reads, making it amenable to interactive exploration of large data sets, even on low-power devices.

Although these libraries are effective at dealing with structured data, they provide limited support for particle data, typically treating it as a set of 1D arrays (e.g., [16]). Common approaches for I/O of particle data use either a single shared file [7], [9], [16], [17], [28] or a file per-process [8], [29]. Such approaches can work well on a single HPC resource or provide tolerable performance at some scales, but are known to struggle with large core counts and portability. Recent work has begun to integrate subfiling [10], [18] to address these limitations in the context of particle data. However, as with approaches used for structured data, these techniques work without knowledge of the underlying spatial domain and do not output data in a visualization- or analysis-tailored layout.

Kumar et al. [27] recently presented a two-phase I/O strategy for particle data that aggregates the data into spatially coherent groups. The proposed technique demonstrated the viability of spatially aware two-phase I/O for particle data; however, it aggregates data using an adjustable uniform grid (AUG) and does not adapt to arbitrary nonuniform particle distributions. The method supports adjusting the grid to fit a rectilinear subdomain if the subdomain contains the entire set of particles. However, the particle distribution within the subdomain is assumed to be uniform. Although the paper suggests that more dynamic aggregation approaches can address this limitation, only the AUG is presented and evaluated, and such dynamic strategies are left for future work. In contrast, we build a k-d tree [30] over the ranks' spatial bounds to construct aggregation groups with similar numbers of particles. Our k-d tree-based aggregator construction can adapt to arbitrary nonuniform and time-varying distributions of particles to achieve load-balanced high-bandwidth writes on real-world particle distributions. We demonstrate $2-2.5\times$ faster writes and $3\times$ faster reads compared to AUG [27] on nonuniform and time-varying particle distributions.

B. Visualization- and Analysis-Focused Layouts

Tree hierarchies have been widely used to achieve progressive multiresolution rendering of large-scale particle data [19], [20], [22], or to accelerate nearest neighbor [30], [31] and region queries [30], [32]. In cosmology, effective octree [20], k-d tree [22], and cluster hierarchy [19] approaches have been demonstrated for multiresolution rendering. A concern with these approaches is the additional memory required to store the LOD representations of the data. Adding representative particles [19], [20] or duplicating particles [22] introduces significant memory overhead. Prior work has compensated for this overhead by quantizing and compressing the data [19], [20] or discarding subsets of it [22].

A large body of work has explored data layouts focused on general analysis queries, treating the particles more akin to entries in a database. Simulation data follows a write-once read-many usage model, for which bitmap indexing has been demonstrated to be effective for accelerating queries [17], [23]–[25], [33]–[35]. However, bitmap indices can require a significant amount of storage, from 30% [35] to 66% [33] of the original data size, and in some cases approaching the original data size [17], [36].

As the simulation typically does not write the data in the layout used for analytics, a postprocess conversion must be performed. The cost of this process grows with the data size, and can take from a few minutes [17] or hours [20], [33] to days [21], depending on the hardware used and layout being built. To alleviate this postprocess conversion cost, prior work has constructed the visualization hierarchy [22] or bitmap indices [35], [37] in situ. However, the simulation may still need to write its own layout for checkpoint restart reads, significantly increasing the total data volume written.

Our proposed layout builds on ideas from both visualization- and analysis-focused layouts. We provide LOD by reordering particles and leverage spatial coherence to provide bitmap indexing-based attribute filtering with little memory overhead. The layout is built during I/O, eliminating the need for

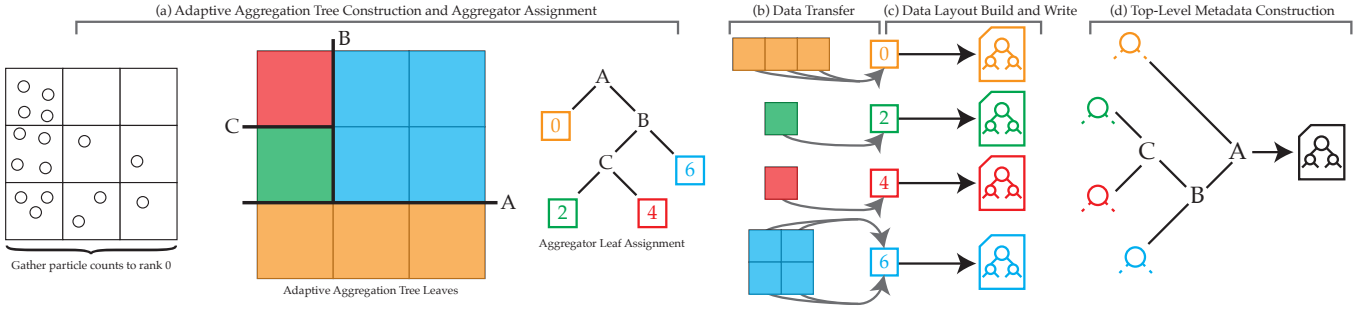


Fig. 1: An overview of our adaptive two-phase I/O pipeline. (a) Given the number of particles on each rank, rank 0 constructs the Aggregation Tree to create leaves with similar numbers of particles. Each leaf is assigned to a rank responsible for aggregating the data and writing it to disk. (b) Each rank sends its data to its aggregator. (c) Each aggregator constructs our multiresolution data layout and writes it to disk. (d) The aggregators send the local value ranges and root node bitmaps for each attribute to rank 0, which populates the Aggregation Tree with the bitmaps and writes it out.

postprocess conversion, and is suitable for high-bandwidth reads, allowing a single copy of the data to be written.

III. Spatially Aware Adaptive Aggregation for Progressive Multiresolution Particle Layouts

To balance the I/O workload as particles move through or are injected into the domain, we employ a k-d tree partitioning computed over the ranks' spatial bounds. Each leaf node in this tree contains a similar number of particles and corresponds to a set of ranks to aggregate together and output to a file. In contrast to the adjustable uniform grid approach of Kumar et al. [27], by building a k-d tree over the ranks to balance the particle distribution among the leaves (i.e., the subfiles), we are better able to adapt to the particle distribution to improve load balance and I/O performance. An overview of our two-phase I/O pipeline is shown in Figure 1.

Each write proceeds as follows: All ranks send their particle counts and domain bounds to rank 0, which constructs the "Aggregation Tree" (Figure 1a, Section III-A). Each leaf in the tree contains a set of ranks with a similar total number of particles and is assigned to an aggregator rank responsible for receiving the data in the leaf and writing it to disk. Each rank sends its local data to the aggregator for the leaf containing it (Figure 1b, Section III-B). After receiving the particles for the leaf, the aggregator constructs and writes out our multiresolution data layout (Figure 1c, Section III-C). We then populate a top-level metadata file on rank 0 containing the Aggregation Tree and references to the leaf files written by the aggregators (Figure 1d, Section III-D). We provide a C API to ease integration of our proposed I/O strategy into simulations written in a range of programming languages. The API follows an array-based attribute storage model similar to HDF5 [1], ADIOS [4], and Silo [38].

A. Constructing the Adaptive Aggregation Tree

To construct the Aggregation Tree (Figure 1a), we gather each rank's bounds in the simulation domain and the number of particles it owns on rank 0. Rank 0 then performs a modified k-d tree build to construct a set of leaves containing similar numbers of particles. As the actual particle distributions within each rank are unknown, we restrict the tree to select split positions that lie on rank boundaries. By not splitting a rank's data between multiple aggregators, we also reduce the data processing and transfers that take place during aggregation.

When building the tree, we want to find a split position that partitions the ranks such that a similar number of particles fall into each subtree. To do so, we select the longest axis of the aggregate bounds of the current set of ranks that have particles and find a set of candidate split positions to test. The candidate splits are the unique edges of each rank's bounds along the chosen splitting axis. For each candidate split, we determine which ranks would fall into the left and right subtrees and compute the corresponding number of particles in each subtree, n_l and n_r respectively. The cost of the split measures how uneven the partitioning is, $c = |0.5 - n_l/(n_l + n_r)|$. We test each candidate to find the one with the minimum cost, and repeat this process to construct the subtrees. The tree construction is parallelized top-down using Intel TBB. A task is spawned to construct the right subtree, while the current thread proceeds with the left. Users can also optionally configure the tree to find and use the best split across all spatial axes.

A leaf node is created when the data contained within the current node falls below a user-specified target file size. Each leaf in the tree corresponds to a file that will be written to disk, containing the data of the ranks within the leaf's bounds. The target size determines the size and number of the output files and the amount of network traffic during the aggregation phase. Lower sizes output more smaller files with less data transferred; larger sizes output fewer larger files with more data transferred. The best size varies across HPC systems and scales and is exposed as a tunable parameter for portability.

To avoid forcing the creation of extremely imbalanced leaves to satisfy the target file size, the tree can be configured to allow the creation of "overfull" leaf nodes. An overfull leaf is created when the minimum split cost exceeds a user-set threshold, and the current data size is within a user-set factor of the target size, balancing between avoiding bad splits and creating excessively large files. The target size can also be exceeded if a single rank exceeds it, as data within a rank is not partitioned.

To ensure even utilization of the network, we distribute the assignment of leaves to aggregators evenly across the rank space [39]. Although this assignment does not ensure that each aggregator is contained in its assigned leaf, it provides better work distribution for the aggregation stage. For example, ranks on the same node likely operate on neighboring regions of the domain. If this region is densely populated with particles, more leaf nodes will be created and assigned to be aggregated by ranks on the node, leading to oversubscription. Similarly, nodes

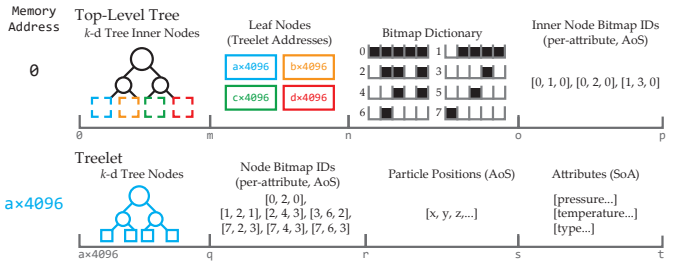


Fig. 2: The BAT layout is organized on disk for fast spatial and attribute query traversal and LOD via memory mapping. The top-level tree is stored at the start of the file in two parts. The first section contains the k-d tree inner nodes, followed by the leaf nodes. The leaf nodes are addresses of the corresponding treelet in the file. The shared bitmap dictionary is also stored at the start of the file, along with the IDs of the top-level tree's inner node bitmaps. Each treelet is aligned to a 4KB page boundary for fast access via memory mapping. Each treelet consists of the k-d tree and the bitmap IDs for each node, followed by the particle data.

with ranks in less populated regions would be underutilized.

Rank 0 scatters to each rank its assigned aggregator ID and the number of particles that it will receive if it is an aggregator, or zero if not. Each aggregator is also sent a list of the ranks that it is assigned to receive data from and the number of particles on each assigned rank.

B. Data Transfer to Aggregators

Each aggregator allocates buffers to store the particles it will receive and their attributes, and posts nonblocking receives for each rank in its leaf. Each rank sends its data (if any) to its assigned aggregator by initiating nonblocking sends (Figure 1b). If a rank does not have particles, the data transfer is skipped. We do not create subcommunicators for leaf aggregation groups since the rank assigned to receive data for the leaf may not be contained in the leaf, and thus must simultaneously participate in a different, disjoint group.

C. Construction of Our Multiresolution Data Layout

After receiving the particles on each aggregator, we construct our data layout, the Binned Attribute Tree (BAT), to support spatial and progressive multiresolution reads (Figure 1c). To support multiresolution reads, each inner node in the tree stores a fixed number of LOD particles to provide a coarse representation of the subtree without additional memory overhead. Attribute queries are accelerated by storing bitmap indices at each node. We build the tree in two parallel steps: a data-parallel bottom-up build that constructs a shallow k-d tree (Section III-C1) and top-down builds of treelets in each leaf of the shallow tree (Section III-C2). These steps are executed in parallel and are further parallelized internally. We then compact the tree to a memory layout optimized for fast multiresolution reads (Section III-C3, Figure 2). After compaction, the tree can be used for in transit visualization and analysis on the aggregators before or instead of being written to disk. Each aggregator writes its tree to an independent file.

1) Bottom-Up Shallow Tree Construction: We use Karras's parallel bottom-up k-d tree construction algorithm [40] to build the tree in parallel, with a small modification to the input set to construct a shallow tree. Karras's algorithm works as follows: The Morton code of each particle is computed and placed into a sorted array, forming the leaf nodes of the tree. The inner nodes of the tree are computed in parallel by finding the common bit prefixes of the Morton codes, producing a radix tree over the points. The resulting radix tree can be directly interpreted as a k-d tree, where each inner node's Morton code prefix indicates the split axis and position. The construction is fast and can be performed entirely in parallel on CPUs and GPUs. However, the tree stores a single particle per-leaf, which is not suited to our goal of multiresolution reads and low memory overhead.

To build a shallow tree, we use a subprefix of each particles' Morton code and merge shared subprefixes to produce a smaller set of codes. The build process then proceeds as described above. The leaves of the shallow tree correspond to rectangular regions containing large numbers of particles. We have found that a 12-bit subprefix provides satisfactory results with respect to the number of leaves and particles within each. We construct a treelet within each leaf of the shallow tree.

2) Parallel Treelet Construction: The treelet builds for each leaf in the shallow tree are independent and run in parallel. For each leaf, we find the set of particles sharing the leaf's subprefix using a binary search over the sorted set of Morton codes and construct a median split k-d tree over them.

To support multiresolution reads of the data, we store a fixed number of LOD particles at each treelet inner node. When creating an inner node, we perform a stratified sampling of the input particles to select the LOD particles and set them aside for the inner node. By taking subsets of particles for LOD, we avoid duplicating them or introducing new representative ones, avoiding additional memory overhead.

To support attribute-based filtering, each leaf node computes a 32-bit bitmap index for each attribute of its contained particles, which are propagated back up the treelet and, eventually, the shallow tree. Inner nodes compute their bitmaps by merging those of their children with those of their LOD particles.

In contrast to standard bitmap indexing approaches, where the size of the bitmap can vary as needed at the cost of memory [23]–[25], we fix the bitmaps to be 32 bits. Although restricting the bitmap size reduces their accuracy when filtering queries, it provides two key benefits to reduce memory overhead. First, the bitmaps occupy a fixed and predictable amount of storage. Second, we can build a dictionary of the unique bitmaps in the data set and replace bitmaps in the tree with smaller indices into this dictionary to reduce overhead.

Besides this 32 bit restriction, we use a standard binning strategy to compute the bitmaps [23], where each bit corresponds to a bin covering 1/32 of the attribute's range. To avoid bins becoming so large as to not provide useful filtering, the bitmaps are computed relative to the local attribute range on the aggregator. As simulation attributes are often spatially correlated, the local range is likely a subset of the global range, allowing for finer bin sizes. Bitmaps can be combined using a bitwise `OR` and tested for overlap using a bitwise `AND`, making such operations fast to perform.

3) Tree Compaction: After the build steps have completed, we compact the tree into a single buffer that can be efficiently

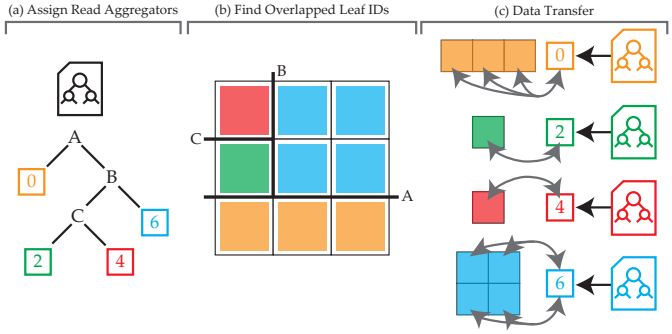


Fig. 3: An overview of our parallel read pipeline. (a) All ranks read the Aggregation Tree metadata, and a subset is selected to act as read aggregators. (b) Each rank determines which leaves it overlaps, and (c) requests data from the aggregator(s) assigned to the leaf file(s).

written to disk and structure data in the buffer to support fast reads via memory mapping (Figure 2). The bitmaps of each treelet are merged into a dictionary of unique bitmaps and replaced with 16-bit IDs into this dictionary to further reduce memory use. Using 16-bit IDs limits the dictionary size to 65k bitmaps; however, we have found this to be more than sufficient in practice.

At the start of the file, we store the shallow tree as a linear *k*-d tree. The bitmap dictionary is stored with the shallow tree, since it will be accessed frequently during traversal. The leaf nodes of the shallow tree are replaced with offsets to the corresponding treelets in the buffer. Treelets are stored at 4KB page boundaries to improve read access and consist of a header specifying the number of nodes and points in the treelet, followed by arrays containing the nodes and particles.

D. Construction of Top-Level Metadata

The final step in the I/O process is the population of a top-level metadata file on rank 0 (Figure 1d). The top-level metadata file allows read access to the entire data set as if it were a single file, transparently supporting spatial and attribute queries and multiresolution reads on the entire data set. We gather the value range and bitmap index of each attribute from each aggregator's root BAT node to rank 0, which populates the corresponding leaf nodes in the Aggregation Tree. Each aggregator's bitmap indices are remapped from its local range to the global range for the attribute. Inner node bitmaps are computed by merging the bitmaps bottom-up from the leaves.

IV. Parallel Reads

An equally important requirement for a parallel I/O library is that it provides high-bandwidth reads for fast checkpoint restart reads. To provide scalable reads on the aggregated data output in Section III, we implement a two-phase parallel read pipeline that mirrors our two-phase write. The read pipeline proceeds as follows: First, all ranks read the Aggregation Tree metadata, and a subset of ranks is assigned to act as "read aggregators" (Figure 3a, Section IV-A). Each rank then determines which leaves it overlaps (Figure 3b) and requests data over the network from the read aggregators for these leaves (Figure 3c, Section IV-B). As for parallel writes, we provide a C API in our parallel I/O library to allow integration into a variety of simulations.

A. Assignment of Read Aggregators

Each rank reads the Aggregation Tree metadata file to determine the number of leaf files and the spatial bounds of each leaf (Figure 3a). Each leaf file is then assigned to a read aggregator responsible for reading the file.

When reading a data set, we are no longer in control of the number of leaf files, since this is set when writing the data. If we have more ranks than files, we assign read aggregators as done in the write phase, spreading them evenly through the rank space to distribute work over the nodes. If we have fewer ranks than files, we assign the files evenly among the ranks. Providing this flexibility in read aggregator assignment allows us to support scalable reads of data written at much larger or smaller core counts than the reading process. The read aggregator assignments are computed locally on each rank, producing a map of which files will be read by each rank.

B. Fetching Data from Read Aggregators

Each rank queries the Aggregation Tree to get back a list of leaf IDs that overlap its bounds (Figure 3b) and sends its bounds to the read aggregator assigned to each leaf. Upon receiving the bounds, the read aggregator performs a spatial query on its leaf files and returns the particles (Figure 3c).

As was the case for writes, we cannot create aggregator subgroups to transfer the data during reads. Instead, we implement a client-server data query system using nonblocking MPI calls. Each read aggregator acts as a data server and watches for incoming queries to process. Each rank collects the number of particles that will be returned by each query, allocates a single buffer to contain them, and then uses nonblocking receives to write the particles directly into this buffer. After a rank has received its particles, it calls a nonblocking barrier and continues the server loop to handle additional queries. When the barrier is complete, we know that all ranks have received their data and the read is complete. If a rank requires data from itself, it performs these queries locally after exiting the server loop. This query mechanism can also be leveraged to enable distributed data access for in situ analytics.

V. Visualization Reads

Visualization reads on our layout take a desired quality level, bounding box for spatial filtering, and set of attribute filters. The user also provides a callback that is called for each point contained in the query. Our data layout directly accelerates common visualization and analysis tasks involving spatial and attribute subset queries (Section V-A) and supports low-latency progressive multiresolution reads for data streaming (Section V-B). Reads are performed via memory mapping to leverage the operating system's caching mechanisms for frequently accessed regions of the data and for fast access to page aligned regions of the files (i.e., the treelets).

A. Spatial and Attribute Query Acceleration

Supporting spatial and attribute filtering directly in the data layout reduces the amount of data that must be processed to

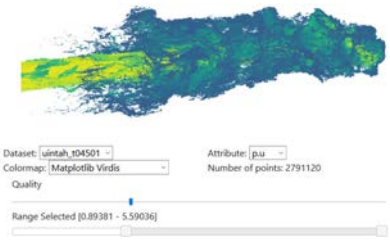


Fig. 4: A prototype web viewer client that progressively streams data from a server. The server uses our BAT layout to progressively load and send data back to clients and apply spatial- and attribute-based filtering.

answer a user query, improving response time and reducing memory use. As our data layout is fundamentally a spatial k-d tree, spatial queries are accelerated by the spatial hierarchy stored in the files.

When performing attribute filtering, we test the user's query bitmaps against those stored at each node during traversal. If the bitwise AND of the node and query bitmaps is zero, the subtree can be ignored. Before returning points to the user's callback, we must check that the query contains no false positives. Although the spatial filtering provided by the k-d tree is exact, the binned bitmaps only ensure that we do not discard false negatives, requiring a final false positive check for attribute queries.

B. Low-Latency Progressive Multiresolution Reads

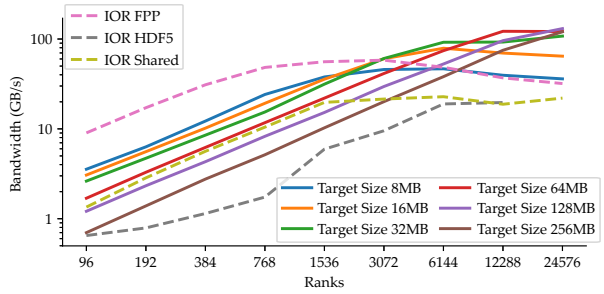
To support fast multiresolution reads, our layout can be queried at different quality levels to return a representative subset of the data (Figure 4). Progressive reads can be performed by providing the previously queried quality level and the desired level, in which case only the new particles for the increment in quality are processed. The quality-level parameter ranges from zero to one. Zero corresponds to loading nothing, and one to the entire data set. We remap this parameter using a log scale to provide a smoother quality progression, since the number of LOD particles stored doubles each tree level. The value is then scaled to a maximum treelet depth to traverse to. When performing progressive reads, the previous quality is mapped to a previously read minimum treelet depth. The tree is then traversed to the maximum treelet depth, processing only points above the minimum depth. We compute a percentage of the points at the maximum level to process to provide a smoother LOD transition. Spatial and attribute filtering can also be performed when using progressive resolution reads.

VI. Results

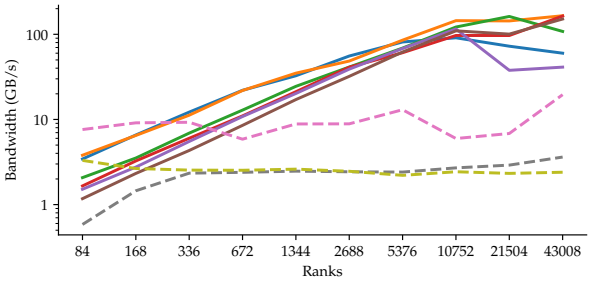
To evaluate our spatially adaptive parallel I/O approach, we perform an extensive set of benchmarks on parallel writes and reads (Section VI-A), using fixed uniform and time-varying nonuniform particle distributions. We report the bandwidth achieved compared to standard approaches and study timing breakdowns of the stages comprising our I/O pipeline. We then demonstrate the effectiveness of our multiresolution data layout for analysis tasks and low memory overhead (Section VI-B).

A. Parallel Writes and Reads

Our parallel I/O benchmarks are run on two HPC systems with different I/O architectures: Stampede2 and Summit.

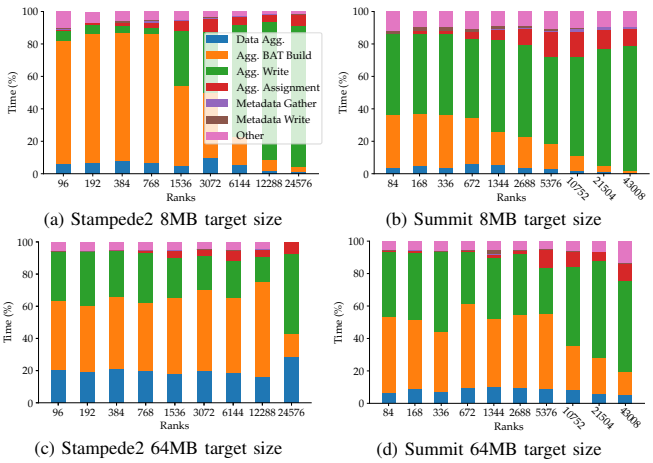


(a) Stampede2 (peak of 786M particles, 99.9GB at 24k ranks)



(b) Summit (peak of 1.37B particles, 174.6GB at 43k ranks)

Fig. 5: Write bandwidth weak scaling on the fixed uniform test data compared to IOR benchmarks. At scale, our two-phase approach outperforms standard file per process and single shared file approaches.



(a) Stampede2 8MB target size

(b) Summit 8MB target size

(c) Stampede2 64MB target size

(d) Summit 64MB target size

Fig. 6: Timing breakdowns on Stampede2 and Summit. In the scaling regime of each target size, the relative time spent in each component remains similar.

Stampede2 uses a Lustre file system and we write to the scratch system, which is capable of 330GB/s peak write bandwidth, where we use a stripe count of 32 and stripe size of 8MB. Summit uses an IBM Spectrum Scale (GPFS) filesystem with a peak write speed of 2.5TB/s. Both systems use a fat-tree topology network, capable of 100Gb/s on Stampede2 and 184Gb/s on Summit. On Stampede2, we use the dual socket Xeon Skylake (SKX) nodes. Each Summit node has two POWER9 CPUs and six Volta V100 GPUs.

1) Weak Scaling on Uniform Particle Distributions: To provide a baseline comparison against standard I/O strategies, we perform a weak scaling study on a generated uniform particle distribution. We compare performance against IOR benchmarks [41] on an equivalent amount of data using IOR's file per process, single shared file (MPI I/O), and HDF5 shared file modes. To represent a moderately sized simulation (e.g., [8]), we generate 32k particles on each rank. Each particle

stores three single precision spatial coordinates and 14 double precision attributes, corresponding to 4.06MB per rank.

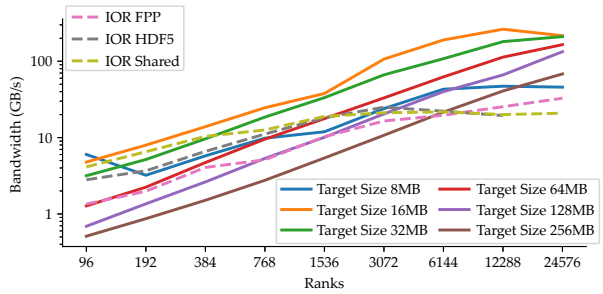
In the benchmarks, we write and read the data 15 times and plot the geometric mean of bandwidth achieved, as done in the IO500 [42]. We are interested in the I/O time observed by a simulation, so we do not perform an explicit fsync. To avoid the impact of OS caching when reading the data, the benchmarks read the data on a different rank than that which wrote it. We also study the effect of the target file size on write and read performance, varying it from 8MB (file per-process) to 256MB (≈63 ranks per file).

On both Stampede2 and Summit (Figure 5), we find that our two-phase I/O approach outperforms standard ones at higher core-counts. Although file per-process initially performs well on both systems, the overhead of creating and writing the large number of files begins to degrade performance at 672 ranks on Summit and 1536 ranks on Stampede2. We observe similar degradation in our method when using small target sizes; however, by increasing the target size to perform more aggregation and write fewer, larger files, these issues can be avoided. The shared file approaches encounter scaling issues due to the global communication required during I/O.

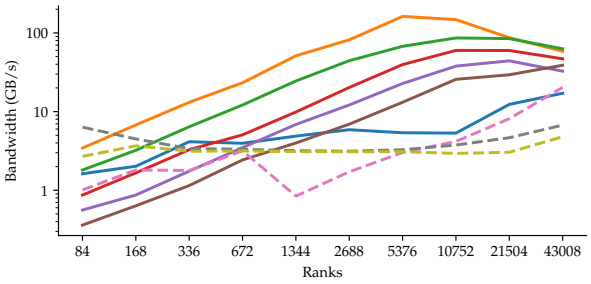
The bulk of time in our I/O pipeline is spent writing the aggregator files to disk, constructing the BATs on each aggregator, and transferring data to aggregators (Figure 6). Comparing the 8MB and 64MB target size runs, we find that the 64MB configuration spends a relatively consistent amount of time in each component as we scale up, whereas the 8MB configuration spends a greater percentage of time in writes at higher core counts where the corresponding scaling trend flattens off. On Stampede2, we find a larger percentage of time is spent constructing the BAT layout. The build is compute and memory bandwidth heavy, and likely benefits from the larger L3 cache of the POWER9 CPUs on Summit.

On parallel reads, we find that our approach outperforms file per process and single shared file strategies beyond moderate core counts on both systems (Figure 7). We observe similar performance trends as were seen on writes, where the overhead of many small files in file per process and small target size configurations impacts performance, whereas the global communication of single shared file approaches limits scalability. Our two-phase approach allows selecting the target size to avoid both issues and achieve high bandwidth reads. On Summit, we observe the scaling trend for small to medium size aggregation settings flattening off or decreasing by 43k cores, although the 256MB aggregation size does not flatten off as rapidly. Selecting an even larger aggregation size could help continue scaling read bandwidth past 43k cores.

2) Comparison vs. Prior Work: We use data sets from two simulations with time-varying nonuniform particle distributions (Figure 8) to study the effect of our adaptive aggregation strategy on I/O performance. We compare our approach against the adjustable uniform grid (AUG) approach of Kumar et al. [27] implemented within our library to provide a direct algorithmic comparison. The grid is built based on the target size to assign ranks to aggregators and discards empty regions



(a) Stampede2 (peak of 786M particles, 99.9GB at 24k ranks)



(b) Summit (peak of 1.37B particles, 174.6GB at 43k ranks)

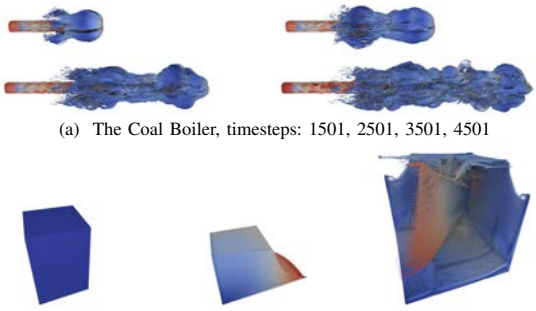
Fig. 7: Read bandwidth weak scaling on the fixed uniform test data, compared to IOR benchmarks. Our two-phase parallel read strategy outperforms standard file per process and single shared file reads at scale.

of the grid, as described by Kumar et al. Our adaptive approach is configured to allow the creation of overfull leaves up to 1.5× the target file size, if the best split to partition the ranks has a cost of four or higher. The benchmarks are performed on the SKX nodes on Stampede2.

The Coal Boiler (Figure 8a) is a real-world simulation performed using Uintah [29], simulating the injection of coal particles into a boiler. Uintah is a computational framework that has been used to simulate large-scale nonuniform particle distributions scaling up to 512k cores of Mira (65% of the machine) [8]. The domain is partitioned using a 3D grid and resized to fit the data bounds as they change over time. At timestep 501, the simulation contains 4.6M particles and reaches 41.5M at timestep 4501. We perform the I/O benchmark using 1536 ranks. Each particle saves three floating point coordinates and seven double precision attributes.

The Dam Break (Figure 8b) is a 3D free surface water column collapse simulation, containing a fixed number of particles that move through the domain. The Dam Break was simulated with ExaMPM, a mini-app developed using the Exascale Computing Project (ECP) Cabana particle toolkit [11], that accurately represents the I/O workload of production applications. The domain is partitioned among the ranks using a 2D grid along x and y (the floor) to achieve better compute load balance. We use two versions of the Dam Break to compare scalability, one with 2M particles on 1536 ranks and one with 8M particles on 6144 ranks. Each particle saves three floating point coordinates and four double precision attributes.

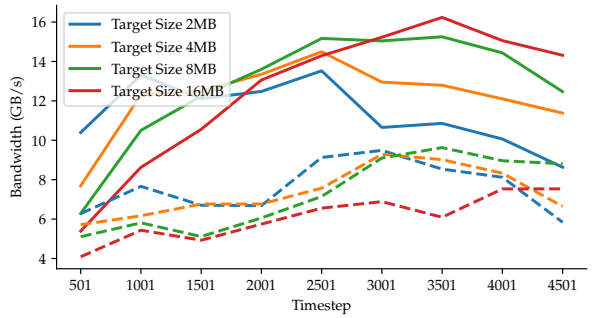
On the Coal Boiler, we find that our adaptive I/O strategy can improve write performance by up to 2.5× compared to AUG aggregation (Figure 9a). Parallel reads on the adaptively aggregated data can be up to 3× faster (Figure 9b). As the number of particles in the simulation increases, we observe decreasing performance at lower target sizes, whereas larger



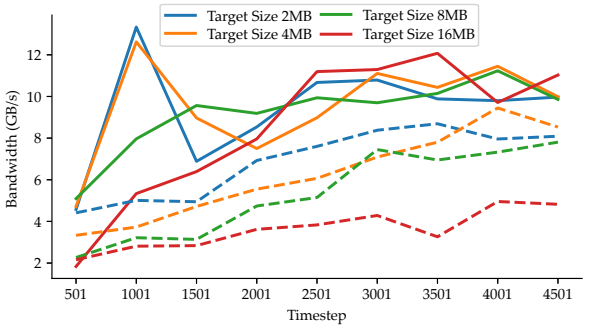
(a) The Coal Boiler, timesteps: 1501, 2501, 3501, 4501

(b) The Cabana Dam Break, timesteps: 0, 1001, 4001.

Fig. 8: The time-varying nonuniform simulation data used in our benchmarks.



(a) Write bandwidth.



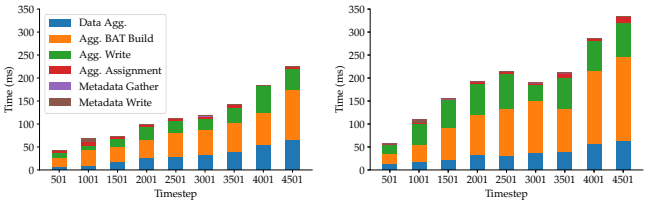
(b) Read bandwidth.

Fig. 9: Adaptive vs. AUG I/O on the Coal Boiler time series on 1536 ranks. Dashed lines indicate AUG aggregation [27]. Our adaptive approach is able to improve write and read performance for imbalanced I/O workloads.

target sizes surpass them. Similar trends are observed for reads. When comparing timing breakdowns at 8MB (Figure 10), we find that our adaptive strategy spends less time in the major steps of our I/O pipeline.

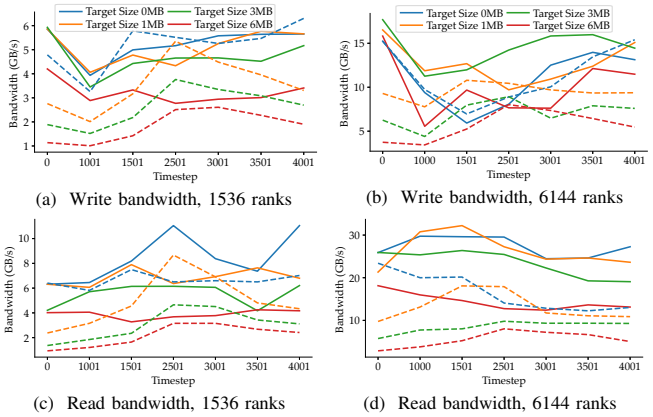
These performance improvements are the result of our adaptive aggregation ensuring a more balanced I/O workload. For example, on the 8MB target size runs at timestep 4501, AUG aggregation outputs 296 files, with an average size of 10.2MB and standard deviation of 13.9MB. Our adaptive aggregation outputs 327 files, with an average size of 9.2MB and standard deviation of 8.4MB. The largest file written by the AUG aggregation was 72.9MB, whereas the largest written by our adaptive aggregation was 36.6MB.

On the 2M Dam Break, we find that the file per process mode of both strategies achieves the best (and similar) write performance (Figure 11a); however, the adaptively written data provides slightly faster reads (Figure 11c). On the 8M Dam Break, the 3MB target size using adaptive aggregation achieves the best write performance overall (Figure 11b), at a $1.5-2\times$



(a) Adaptive aggregation (b) AUG aggregation

Fig. 10: Breakdowns of adaptive vs. AUG I/O on the Coal Boiler, 8MB target size. The improved load balance achieved by our approach reduces time spent in the major components of the pipeline, improving write performance.



(a) Write bandwidth, 1536 ranks (b) Write bandwidth, 6144 ranks

(c) Read bandwidth, 1536 ranks (d) Read bandwidth, 6144 ranks

Fig. 11: Adaptive vs. AUG aggregation on the Dam Break time series. Dashed lines indicate AUG aggregation [27]. The performance improvement provided by adaptive aggregation increases at larger scales for imbalanced simulations.

speed-up over AUG aggregation at the same target size, with up to $3\times$ speed-ups observed for reads. The performance gap between adaptive and AUG aggregation grows with the particle and core count, with the exception of file per process writes.

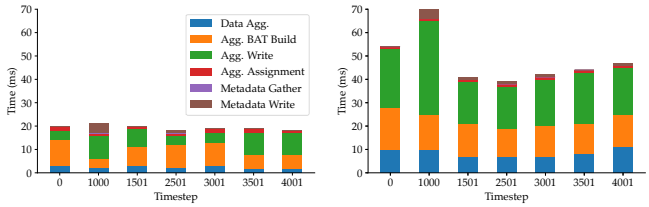
The Dam Break contains a fixed number of particles, and an ideal I/O strategy would be expected to achieve constant write times over the time series. In a timing breakdown of the 3MB run on the 8M Dam Break (Figure 12), we find that the AUG approach is strongly affected by the simulation's particle distribution, whereas our adaptive approach achieves nearly constant write times.

Based on our experiments, we recommend using smaller target sizes at lower core or particle counts, corresponding to roughly $1:1$ to $4:1$ aggregation factors. At larger scales, the target size should be increased to $16:1$ or higher to avoid creating a large number of files. If particles are added during the simulation, as in the Coal Boiler, the target size should be increased correspondingly to maintain high-performance I/O.

B. Visualization Reads

To examine the suitability of our Binned Attribute Tree for visualization and analysis, we evaluate its support for low-latency progressive multiresolution reads and report the memory overhead required to store it. The evaluation is performed using a single threaded process on a desktop with an i9–9920X CPU, 128GB of RAM, and a 1TB Samsung 970 NVMe drive.

1) Low-Latency Progressive Multiresolution Reads: We report average read performance on the Coal Boiler (Table I) and Dam Break (Table II) for a typical progressive multiresolution use case. Starting from a coarse quality level of 0.1 (~10% of



(a) Adaptive aggregation (b) AUG aggregation

Fig. 12: Breakdowns of adaptive vs. AUG I/O on the 8M Dam Break, 3MB target size. Our adaptive approach maintains nearly constant I/O times, whereas the AUG approach is influenced by the changing distribution of the particles.

TABLE I: PROGRESSIVE SINGLE-THREAD READ TIMES AND THROUGHPUT ON THE COAL BOILER TIMESERIES WRITTEN USING 1536 RANKS.

Target Size	Avg. Read (ms)	Avg. Throughput (points/ms)
2MB	72.5ms	54968 pts/ms
4MB	69.1ms	55663 pts/ms
8MB	71.8ms	54148 pts/ms
16MB	70.2ms	52501 pts/ms

the data), successively higher quality levels are requested in increments of 0.1 until the entire data set is loaded. We record the time spent to traverse the tree and process each requested point. The BATs are built with eight LOD particles per treelet inner node and up to 128 particles per treelet leaf.

We achieve similar performance when reading data aggregated at different target sizes on both data sets. On the Coal Boiler, we find slightly better performance at 4MB, and on the Dam Break at 3MB. The largest factor determining performance is the number of points queried. The Coal Boiler sees higher query times since each query returns far more points than on the Dam Break. When comparing the data sets in terms of points per millisecond read throughput, we find similar throughput on the Coal Boiler and 8M Dam Break. The 2M Dam Break achieves higher throughput, likely due to its small size allowing more data to remain cached by the OS.

Our BAT layout does not impose a specific visual representation on users since the best choice is often domain specific. To provide an example of the quality progression provided by our layout, we implement an LOD approach where coarser representations are displayed with increased particle radii to fill holes and preserve the overall shape of the object, shown on the Coal Boiler in Figure 13. By restricting the bitmap index sizes and avoiding duplication for LOD particles, we achieve low memory overhead for our layout, requiring just 0.9% additional memory to store.

VII. CONCLUSION

We have presented a spatially adaptive approach for parallel I/O of multiresolution particle data layouts. Our approach outperforms standard methods on uniform distributions, and achieves up to 2.5× faster writes and 3× faster reads on nonuniform data compared to the prior state of the art [27]. Our low overhead data layout built when writing the data is directly available for in situ and postprocess visualization tasks, eliminating the need for data conversion or duplication. Our code and reproducibility information can be found on GitHub[1].

[1]https://github.com/Twinklebear/libbat

TABLE II: PROGRESSIVE SINGLE-THREAD READ TIMES AND THROUGHPUT ON THE 2M AND 8M PARTICLE DAM BREAK TIME SERIES.

Target Size	Avg. Read (ms)	Avg. Throughput (points/ms)
2M total points, written on 1536 ranks		
0MB	2.7ms	71441 pts/ms
1MB	2.7ms	70012 pts/ms
3MB	2.6ms	72997 pts/ms
8M total points, written on 6144 ranks		
0MB	12.7ms	57659 pts/ms
1MB	12.8ms	57703 pts/ms
3MB	12.4ms	58926 pts/ms

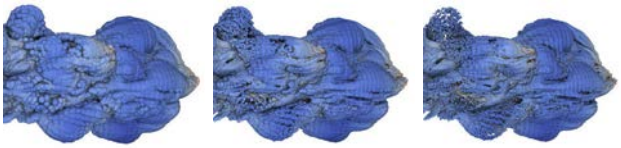


Fig. 13: The visual quality progression on the Coal Boiler. Quality, from left to right: 0.2, 0.4, 0.8.

A. Limitations

Although our results are compelling, there remain limitations to address and exciting avenues for future work. Our attribute filtering approach assumes spatial coherence in the attributes; if this is not the case, the bitmaps in the tree will be less useful, degrading query filtering performance. This limitation could be addressed by adopting more advanced binning schemes [43] or additional hierarchies on these attributes. Moreover, the effectiveness of limiting bitmaps to just 32 bits warrants further evaluation. Our approach is also somewhat limited in how much it can rebalance the I/O load, since we do not divide a single rank's data into separate leaves. It would also be valuable to support automatically selecting the target size based on the particle count and size using the results of our evaluation. Finally, our BAT layout does not make use of compression or quantization, which would reduce memory use further.

Allowing users to build their own data layout would ease adoption of our method for simulation-analysis pipelines that already use a specific layout. The layout would also be available in situ, easing the process of bringing postprocess analytics reliant on a specific data layout in situ. The output structure size is likely related to the input data size, allowing our approach to still provide effective load balancing.

ACKNOWLEDGMENTS

This work was funded in part by NSF OAC awards 1842042, 1941085, NSF CMMI awards 1629660, LLNL LDRD project SI-20-001, DoE award DE-FE0031880, and the Intel Graphics and Visualization Institute of XeLLENCE. This material is based in part upon work supported by the DoE NNSA under award DE-NA0002375. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DoE and the NNSA. This work was performed in part under the auspices of the DoE by LLNL under contract DE-AC52-07NA27344, and UT-Battelle, LLC under contract DE-AC05-00OR22725. The authors thank the Texas Advanced Computing Center for access to Stampede2. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DoE User Facility.

REFERENCES

[1] "HDF5 Home Page," http://www.hdfgroup.org/HDF5/.

[2] 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 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[3] Jianwei Li, Wei-keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.

[4] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, Metadata Rich IO Methods for Portable High Performance IO," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium On*, 2009.

[5] S. Kumar, J. Edwards, P.-T. Bremer, A. Knoll, C. Christensen, V. Vishwanath, P. Carns, J. A. Schmidt, and V. Pascucci, "Efficient I/O and storage of adaptive-resolution data," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[6] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference On*, 2011.

[7] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *Journal of Computational Physics*, 1995.

[8] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices," *SIAM Journal on Scientific Computing*, 2016.

[9] S. W. Skillman, M. S. Warren, M. J. Turk, R. H. Wechsler, D. E. Holz, and P. M. Sutter, "Dark Sky Simulations: Early Data Release," *arXiv:1407.2600*, 2014.

[10] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "HACC: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, 2016.

[11] S. Slattery, C. Junghans, D. Lebrun-Grandie, R. Halver, G. Chen, S. Reeve, A. Scheinberg, C. Smith, and R. Bird, "ECP-copa/Cabana: Cabana Version 0.2.0," Mar. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2591488

[12] W. Ge, R. Sankaran, and J. H. Chen, "Development of a CPU/GPU portable software library for Lagrangian-Eulerian simulations of liquid sprays," *International Journal of Multiphase Flow*, 2020.

[13] D. Z. Zhang, Q. Zou, W. B. VanderHeyden, and X. Ma, "Material point method applied to multiphase flows," *Journal of Computational Physics*, 2008.

[14] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Seventh Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[15] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *SIGARCH Comput. Archit. News*, 1993.

[16] M. Howison, A. Adelmann, E. W. Bethel, A. Gsell, B. Oswald, and others, "H5hut: A high-performance I/O library for particle-based simulations," in *Cluster Computing Workshops and Posters, 2010 IEEE International Conference On*, 2010.

[17] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference For*, 2012.

[18] S. Byna, M. Chaarawi, Q. Koziol, J. Mainzer, and F. Willmore, "Tuning HDF5 subfiling performance on parallel file systems," in *Cray User Group*, 2017.

[19] M. Hopf and T. Ertl, "Hierarchical splatting of scattered data," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003.

[20] R. Fraedrich, J. Schneider, and R. Westermann, "Exploring the Millennium Run–Scalable Rendering of Large-Scale Cosmological Datasets," *IEEE Transactions on Visualization and Computer Graphics*, 2009.

[21] K. Schatz, C. Muller, M. Krone, J. Schneider, G. Reina, and T. Ertl, "Interactive Visual Exploration of a Trillion Particles," in *LDAV*, 2016.

[22] J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, S. Habib, and K. Heitmann, "In-situ Sampling of a Large-Scale Particle Simulation for Interactive Visualization and Analysis," *Computer Graphics Forum*, 2011.

[23] K. Wu, W. Koegler, J. Chen, and A. Shoshani, "Using bitmap index for interactive exploration of large datasets," in *Scientific and Statistical Database Management, 2003. 15th International Conference On*, 2003.

[24] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Transactions on Database Systems*, 2007.

[25] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang, "FastBit: Interactively searching massive data," *Journal of Physics: Conference Series*, 2009.

[26] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Supercomputing, ACM/IEEE 2001 Conference*, 2001.

[27] S. Kumar, S. Petruzza, W. Usher, and V. Pascucci, "Spatially-aware Parallel I/O for Particle Data," in *Proceedings of the 48th International Conference on Parallel Processing - ICPP 2019*, 2019.

[28] S. Byna, A. Uselton, and D. Knaak, "Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper," 2013.

[29] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: A unified heterogeneous task scheduling and runtime system," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012.

[30] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 1975.

[31] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Liu, P. Sadowski, E. Racah, S. Byna, C. Tull, W. Bhimji, Prabhat, and P. Dubey, "PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures," *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[32] M. M. A. Patwary, P. Dubey, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, and Prabhat, "BD-CATS: Big Data Clustering at Trillion Particle Scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC'15*, 2015.

[33] J. Chou, K. Wu, O. Rubel, M. Howison, J. Qiang, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani, "Parallel index and query for large scale data analysis," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference For*, 2011.

[34] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu, "Parallel in situ indexing for data-intensive computing," in *Large Data Analysis and Visualization (LDAV)*, 2011.

[35] Y. Su, Y. Wang, and G. Agrawal, "In-Situ Bitmaps Generation and Efficient Data Analysis based on Bitmaps," 2015.

[36] J. Chou, K. Wu, and Prabhat, "FastQuery: A Parallel Indexing System for Scientific Data," in *2011 IEEE International Conference on Cluster Computing*, 2011.

[37] 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 *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010.

[38] Lawrence Livermore National Laboratory, "Silo: A Mesh and Field I/O Library and Scientific Database." [Online]. Available: https://wci.llnl.gov/simulation/computer-codes/silo

[39] S. Kumar, D. Hoang, S. Petruzza, J. Edwards, and V. Pascucci, "Reducing Network Congestion and Synchronization Overhead During Aggregation of Hierarchical Data," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017.

[40] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*, 2012.

[41] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov 2008.

[42] "Virtual Institute for I/O." [Online]. Available: https://www.vi4io.org

[43] K. Wu, K. Stockinger, and A. Shoshani, "Breaking the Curse of Cardinality on Bitmap Indexes," in *Scientific and Statistical Database Management*, B. Ludäscher and N. Mamoulis, Eds., 2008.