ADAPTIVE MULTIRESOLUTION TECHNIQUES FOR I/O, DATA LAYOUT, AND VISUALIZATION OF MASSIVE SIMULATIONS

by

William Usher

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2021

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **William Usher**

has been approved by the following supervisory committee members:

Valerio Pascucci ,	Chair(s)	_____ Date Approved
Christopher R. Johnson ,	Member	_____ Date Approved
Charles Hansen ,	Member	_____ Date Approved
Cem Yuksel ,	Member	_____ Date Approved
Ingo Wald ,	Member	_____ Date Approved

by **Mary Hall** , Chair/Dean of

the Department/College/School of **Computer Science**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

The continuing growth in computational power available on high-performance computing systems has allowed for increasingly higher fidelity simulations. As these simulations grow in scale, the amount of data produced grows correspondingly, challenging existing strategies for I/O and visualization. Moreover, although prior work has sought to achieve high bandwidth I/O at scale or post hoc visualization of massive data sets, treating these two sides of the simulation visualization pipeline independently introduces a bottleneck between them, where data must be converted from the simulation output layout to the layout used for visualization.

The aim of this dissertation is to address key challenges across the simulation visualization pipeline to provide efficient end-to-end support for massive data sets. First, this dissertation proposes an I/O approach for particle data that rebalances the I/O workload on nonuniform distributions by constructing a spatial data structure, improving I/O performance and portability. To eliminate data layout bottlenecks between I/O and visualization, this dissertation proposes a layout for particle data that balances rendering and attribute-query access patterns through a spatial *k*-d tree and fixed size bitmap indices. The layout is constructed quickly when writing the data and requires little additional memory to store. This dissertation proposes a number of approaches to enable visualization of massive data sets at different scales. An asynchronous tile-based processing pipeline is proposed for distributed full-resolution rendering that overlaps compositing and rendering tasks to improve performance. Next, a virtual reality tool for neuron tracing in large connectomics data is proposed. The VR tool employs an intuitive painting metaphor and a real-time page-based data processing system to visualize large data without discomfort. To visualize massive data in compute and memory constrained environments, a GPU parallel isosurface extraction algorithm is proposed for block-compressed data sets. The algorithm is built on a GPU-driven memory management and caching strategy that allows working with compressed data sets entirely on the GPU. Finally, a simulation-oblivious approach to

data transfer for loosely coupled in situ visualization is proposed that minimizes the impact of the visualization by offloading data restructuring from the simulation.

To my wife Angela Yin,

and my parents Anne Mason and Mike Usher.

CONTENTS

LIST OF TABLES

CHAPTER 1

MOTIVATION AND CONTRIBUTIONS

Simulations have fueled a wave of scientific insights and discoveries over the past 30 years. To accurately simulate large and complex systems, these simulations are run in parallel on up to hundreds of thousands of cores on high-performance computing (HPC) systems and output massive amounts of data, on the order of hundreds of gigabytes per time step. One path to gain insight from these simulations is through the use of visualization, which, in a broad sense, involves creating images, plots, statistical summaries, or other representations derived from the output data to study the data, either interactively or through an offline process. This path, which runs from simulation, through I/O and visualization, to arrive at scientific insight, forms the "Simulation Visualization Pipeline" (Fig. 1.1).

With the push to deploy new exascale HPC systems and develop scalable simulations capable of leveraging them, the pace of growth in the scale of these simulations has accelerated rapidly. These upcoming exascale simulations will enable the study of key large-scale problems across energy, molecular dynamics, cosmology, earth science, medicine, and national security [257]; however, they introduce new challenges due to the scale of both the computation and data produced. Although developing simulation codes capable of scaling to such extremes is a significant challenge in its own right, the simulation code is but one piece of the simulation visualization pipeline. To gain new insights from future exascale simulations through visualization, the I/O system, data layout, and visualization must also scale to high levels of parallelism and massive data sets.

The I/O system and data layout face additional challenges, as the bandwidth available on HPC systems has not kept pace with the growth in computational power. Furthermore, as visualization tasks are typically performed on more modest computational resources, such as a laptop or visualization cluster, it is not sufficient to develop only strategies for

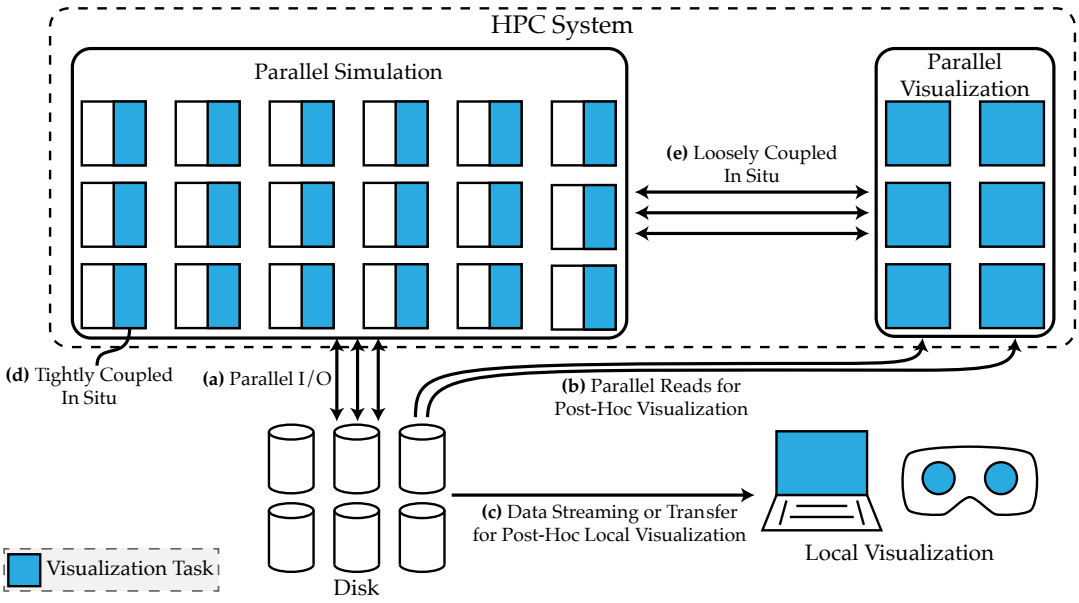


Fig. 1.1: An overview of the simulation visualization pipeline. Data of interest for visualization are produced by a parallel simulation run on an HPC system. The data can be (a) saved to disk for post hoc visualization using a parallel I/O method. Post hoc visualization can be done using (b) distributed parallel visualization algorithms on a visualization cluster or a subset of nodes on the compute cluster, or (c) streamed or downloaded to the scientist's workstation for interactive exploration, e.g., on representative subsets of the data. If the data are too large to save frequently enough for visualization, in situ visualization can be employed. In situ visualization can be used (d) in a tightly coupled configuration, where the visualization tasks are run on each simulation rank, or (e) in a loosely coupled configuration, where data are transferred to a separate process running the visualization.

processing full-resolution data on a large HPC resource. To enable the simulation visualization pipeline, the data layouts and visualization methods must scale from interactive exploratory use cases on representative subsets on a scientist's laptop or workstation to distributed execution on an HPC system. Enabling multiresolution interactive exploration also enables deployment of complex visualizations in low-power environments to improve accessibility; and the use of immersive technologies such as virtual reality (VR), through which scientists can gain a deeper understanding of their data and interact with it in novel ways. However, widely used I/O strategies focus on achieving high bandwidth writes and reads, and do not construct the data layouts needed to enable such multiresolution visualization applications.

In situ visualization has emerged as a key technology to enable visualization at exascale [5]. As simulations grow in size, saving the data at a high enough temporal frequency

to perform the desired visualization becomes infeasible, due to either the time or space required to write out the data. In situ visualization moves the visualization tasks to run inside the simulation process or in parallel alongside it, eliminating the need to save the raw data to disk. The plots, images, and other extracts computed by the visualization are far smaller than the raw data, allowing them to be computed and saved far more frequently. A key concern when adopting in situ visualization is the impact it will have on the simulation; for example, the visualization code may not scale as well as the simulation code and thus may impact overall performance.

The aim of this dissertation is to address issues arising in the simulation visualization pipeline through a holistic approach that adopts adaptive and multiresolution processing throughout the pipeline. Starting at the I/O step of the pipeline, this dissertation proposes a spatially aware adaptive I/O strategy for large particle simulations. The proposed I/O strategy adapts to nonuniform particle distributions to improve performance and outputs the data in a layout readily usable for visualization, without sacrificing write or read throughput compared to standard approaches. In the context of post hoc visualization, this dissertation proposes a range of strategies that enable interactive visualization of massive data sets at full resolution on HPC systems and in VR or the web browser. To render the data interactively at full resolution, an asynchronous tile processing pipeline is proposed that supports scalable image-, data-, and hybrid-parallel rendering. Methods based on adaptive computation and multiresolution data layouts are proposed to enable interactive visualization of massive data sets in VR and the web browser. Finally, to minimize the impact of in situ visualization on the simulation, this dissertation proposes a simulation-oblivious data transfer model for loosely coupled in situ visualization. The proposed data transfer model minimizes the amount of computation performed in the simulation during data transfers, and supports flexible placement of the visualization to reduce its impact further. This dissertation demonstrates that a unified, end-to-end adoption of adaptive and multiresolution data layouts and algorithms is critical to enable science at exascale, and presents algorithms for doing so at key steps in the simulation visualization pipeline.

The remainder of this chapter provides additional background on the simulation visualization pipeline that the methods presented in this dissertation operate within.

Section 1.1 discusses this pipeline; Section 1.2 discusses challenges with I/O, data layout, and post hoc and in situ visualization within this pipeline; and Section 1.3 summarizes the contributions of this dissertation.

1.1 The Simulation Visualization Pipeline

The simulation visualization pipeline is the path through which data produced by a simulation flows to eventually be used to gain scientific insight from the computation (Fig. 1.1). In a typical simulation workflow, a parallel simulation is run distributed across multiple compute nodes on an HPC system. Multiple processes may be run on each node; each such process is referred to as a rank. Each rank is responsible for simulating some subregion of the physical domain, thereby parallelizing the computation over hundreds to hundreds of thousands of cores. The ranks communicate over the network to compute phenomena that cross these subregion boundaries. Large-scale HPC simulations can run for hours or days across hundreds to hundreds of thousands of cores. The raw simulation data are periodically saved to disk using a parallel I/O library (Fig. 1.1a) for post hoc visualization. Post hoc visualization can be performed on a visualization cluster at an HPC center (Fig. 1.1b) or on a scientist's workstation (Fig. 1.1c). Visualization is used to produce images, charts, statistical summaries, segmentations, and other statistical or visual representations, to ask and answer scientific questions about the simulated phenomena. The visualization process can be interactive, where scientists can change the visualization parameters and get immediate real-time feedback through some graphical application, or offline, where they configure a long running task that produces the desired results. The data output by the simulation can become so large that it is infeasible to save it frequently enough to perform the desired visualization. In situ visualization techniques have received growing interest in the simulation and visualization communities to address this issue. In situ techniques run the visualization as part of the simulation (Fig. 1.1d), or in parallel to it (Fig. 1.1e), eliminating the need to save the data to disk. The results of the visualization are much smaller than the raw data, allowing the data to be saved out more frequently and studied post hoc. In situ and post hoc techniques can also be used together in the same workflow, where some frequent batch visualization is performed in situ while interactive post hoc visualization is done on the less frequently saved raw data, augmented with the

outputs of the in situ visualization.

1.1.1 Simulation I/O

To quickly write out hundreds of gigabytes of data, simulations rely on parallel I/O libraries that can take advantage of the HPC system's fast network and parallel file system. Parallel I/O libraries often provide multiple or tunable I/O strategies, as some strategies work better on different networks, file system configurations, or levels of parallelism. Common strategies used by these libraries are file per process, single-shared file, two-phase I/O, and subfiling.

File per process works exactly as it sounds, where each process (i.e., rank) writes out a file containing its local data. This strategy is simple and trivially parallel; however, the massive number of files output by even medium scale simulations can overwhelm parts of the parallel filesystem, leading to poor performance. Post hoc visualization on the output of a file per process strategy is also challenging, as the simulation data are scattered across hundreds to thousands of small files, and must be reassembled for processing each time the visualization is run. Single-shared file also works as one would expect, where the simulation processes work collectively to output their data into a single large file. Although the output of this strategy is convenient to work with, the inherent global nature of the I/O process limits its scalability. A file per process or single-shared file approach can perform better than the other on certain HPC systems, depending on the configuration of the network and parallel filesystem. However, both encounter issues at large scales, either due to the massive number of files produced in file per process strategies, or the global communication required in single shared file strategies.

The latter two strategies, two-phase I/O and subfiling, balance between file per process and single-shared file approaches to provide portable, scalable, and tunable I/O strategies. Two-phase I/O strategies [64,156,173,177,270,284] begin by assigning a configurable number of ranks to be data aggregators. The nonaggregator ranks are assigned a data aggregator to send their data to, forming a subgroup. The ranks then send their data over the network to their assigned aggregator, which writes a single file out after it has received data from the ranks in its subgroup. Subfiling [39,90] works in a similar manner, although it does not necessarily aggregate the data over the network to an aggregator rank before writing it out.

Subfiling strategies group ranks into subgroups, and then perform single-shared file writes within the subgroups, outputting a file per subgroup. In both strategies, the communication required during I/O is restricted to the independent subgroups, avoiding the need for global communication. The number and size of the output files and the number of ranks that must communicate during I/O can be adjusted by tuning the size of the subgroups. By tuning the size of the subgroups, simulations can adjust the I/O strategy to achieve the best performance on a given network, filesystem, and level of parallelism. Two-phase I/O and subfiling strategies can be seen as opening up a spectrum of I/O configurations between file per process and single-shared file approaches, enabling simulations to pick the best for a given environment. At one extreme of this spectrum, each rank can be put in its own subgroup, yielding a file per process strategy; at the other end, a single subgroup can be used, yielding a single-shared file strategy. The ability to tune two-phase and subfiling I/O strategies to adapt to different HPC system configurations has resulted in growing adoption in the I/O community.

1.1.2 Post Hoc Visualization

Post hoc visualization enables scientists to interactively explore the data output by the simulation to ask and answer questions about the simulated phenomena, either directly, through an interactive graphical application, or indirectly, by iteratively re-running a batch computation. The visualization output includes renderings, images, charts, statistics, segmentations, and other statistical or visual representations, depending on the questions the scientist wishes to explore. The ability to interactively adjust the visualization parameters on the fly is key to gaining insight from the data through exploration. The interactivity provided in post hoc visualization makes it a critical aspect of the simulation visualization pipeline even when adopting in situ visualization, which typically does not provide the ability to significantly adjust the visualization parameters after the simulation has started. When processing the full-resolution data or performing expensive computation, these post hoc visualization tasks can be run in parallel on a visualization cluster (Fig. 1.1b), or, when working with smaller data sets or multiresolution representations, can be run on a workstation or laptop (Fig. 1.1c). In situ and post hoc visualization can also be used together, either by combining fixed in situ visualization to generate known visualizations

of interest and interactive post hoc visualization to find unknown ones, or by having the in situ visualization compute a sampling of the visualization parameter space to output a set of "explorable extracts" that can be interactively explored post hoc.

The scientist's laptop or workstation is a convenient platform for visualization, as tasks can be run interactively as desired. Processing data locally also provides lower latency interaction and allows for the use of immersive technologies such as VR. Due to its convenience and interactivity, local visualization is widely used in practice; for example, popular visualization tools such as ParaView [6] and VisIt [46] support it as their default usage mode. However, the compute and memory capacity of a single workstation may not be up to the task of processing the full-resolution output of a large-scale simulation or performing complex calculations.

A visualization cluster can be used to process the full-resolution data or perform more complex computation than can be done on a single workstation. The visualization cluster can be a separate dedicated cluster at the HPC center, or simply a smaller job run on the same cluster that was used to run the simulation. In either case, the visualization cluster provides access to significantly more compute capabilities, with the trade-off of convenience. The visualization tasks must now be submitted through a batch job system and cost compute hours to execute, making interactive tasks more difficult and forcing scientists to balance the amount of visualization that can be done within their compute hour budget. Remote interactive or batch visualization is also supported by standard tools. Both ParaView and VisIt are capable of connecting to a remote parallel visualization process for interactive visualization on a cluster. However, the latency introduced by remote interactive visualization can be a concern for visualization tasks requiring low latency, such as VR. Methods developed for post hoc visualization on a cluster can also be good candidates to deploy in situ, as they are already designed for execution in a distributed parallel environment.

1.1.3 In Situ Visualization

The growth in compute power available on HPC systems has far outpaced the available I/O bandwidth, making it infeasible to save the raw simulation data at a high enough temporal frequency to perform certain analyses or capture certain features. This gap

between FLOPs and I/O is expected to grow on exascale systems. Although these new systems will be capable of an ExaFLOP per second of computation, they will not be able to write out an Exabyte of data per second, nor have enough disk space to store such a massive volume of data. These limitations mean that far more data can be simulated than can possibly be saved for post hoc visualization. This challenge has led to the growth of in situ visualization techniques that run the visualization at the same time as the simulation. The raw simulation data are passed directly to the visualization, eliminating the need to write it to disk. The output of the visualization is usually far smaller than the raw data, allowing it to be computed and saved at a much higher frequency than the raw data could be saved. A wide range of strategies for performing in situ visualization have been proposed; for examples see the survey by Bauer et al. [11] and the In Situ Terminology Project by Childs et al. [45]. In situ strategies can be roughly classified into three categories, based on how they are executed with respect to the simulation: tightly coupled, loosely coupled, or data staging.

Tightly coupled approaches run the visualization tasks in the same process as the simulation, allowing them to access the simulation data directly (Fig. 1.1d). The visualization is integrated into the simulation by linking a visualization-specific library containing the visualization code. The visualization typically receives pointers to the raw simulation data to be processed to work directly on the simulation data. As the simulation and visualization work on the same data, they must take turns executing, i.e., share time. After each time step is computed, the simulation calls into the visualization code, and does not begin the next time step until the visualization has completed. From a practical standpoint, integrating the tasks of simulation and visualization into a single program makes it easier to run in the batch execution mode used on HPC systems. Therefore, this is the approach adopted by widely used libraries for in situ visualization, ParaView Catalyst [78] and VisIt LibSim [302]. However, the approach is not without trade-offs. As the simulation and visualization are now run in the same process on the same nodes, the visualization must be aware of the impact it has on the simulation. For example, if the visualization attempts to allocate a large amount of memory that, when combined with the memory already used by the simulation, exceeds what is available on the node, it will crash the simulation. Visualization tasks are also typically run at smaller scales than simulations, and may not scale to the same level

of parallelism as the simulation. With the visualization integrated into the simulation, any poor scaling or performance properties of the visualization will impact the simulation.

Loosely coupled approaches run the visualization tasks in a different process than the simulation, and access the data by copying it over the network or via a shared memory mechanism (Fig. 1.1e). By separating the visualization and simulation into separate processes, loosely coupled approaches provide some desirable benefits at scale [148], providing flexibility as to when, where, and at what scale the visualization tasks are run. This flexibility makes it possible to reduce the impact of the visualization on the simulation, to provide both processes with more compute and memory capacity, and to run each task at its ideal level of parallelism. For example, the visualization can be run on a separate smaller allocation of nodes and perform its computation as the simulation continues on to the next time step to avoid forcing the simulation to wait for the visualization to complete or to share resources with it. However, loosely coupled approaches bring their own trade-offs. Loosely coupled approaches require making a copy of the data, take additional effort to manage the coordination between the separate processes, and can be more challenging to run in some configurations on current HPC systems that use a batch execution mode.

From the simulation's perspective, the operations of handing off data to an I/O library for output or to an in situ visualization library for processing share many similarities. Thus, one approach to transparently integrate in situ visualization is to integrate it within the I/O library already being used by the simulation. This approach has been used effectively in ADIOS [177] and GLEAN [284]. Such an integration eases the development effort required to adopt in situ visualization, as the simulation does not need to integrate a new visualization-specific library that may have its own data layout requirements or data structures. Beyond its practical convenience, such an integration can provide flexibility as to how the in situ visualization is run [327]. The visualization tasks can be run in a tightly coupled configuration on each simulation rank when it calls in to the I/O library; or, when using a two-phase I/O pipeline, on the aggregators after aggregation in a data staging configuration; or, even in a loosely coupled configuration, by integrating a distributed data query system [63, 68, 326] into the I/O library that the visualization processes can connect to and query data from. Integrating a distributed data query system is also often done to enable data staging configurations, as the visualization tasks may need to fetch

data from other ranks to provide neighbor data. Integrating in situ within the I/O library simplifies integration into the simulation and can be used to realize a range of in situ visualization configurations; however, such approaches rely on the simulation already using the specific I/O library containing integrated in situ visualization. If the simulation relies on a different I/O library, potentially one specifically tuned for its data distribution or data types, migrating to another library for the sole purpose of in situ may be undesirable.

1.2 Challenges in the Simulation Visualization Pipeline

When viewed as a single continuous pipeline through which data flows from simulation to visualization, key challenges can be observed that must be addressed to enable science at exascale. In fact, these challenges already present issues today and will become more pressing as simulation data sizes and execution scales grow at exascale. Addressing these challenges is the focus of the work presented in this dissertation.

1.2.1 Challenges Stemming from Conflicting Data Access Patterns

The data access patterns of the simulation and visualization differ significantly, placing conflicting demands on the I/O library and data layout. Simulations demand throughput-focused parallel I/O libraries that can effectively utilize an HPC system's network and parallel filesystem to provide high bandwidth. These parallel I/O libraries enable simulations to quickly output hundreds of gigabytes of raw data so that they can continue to computing the next time step as quickly as possible. High bandwidth reads are needed when restarting a simulation from a checkpoint to quickly load the hundreds of gigabytes of raw checkpoint data. Similar demands are placed on the data layout by the simulation, which must take little time to construct during writes or to load during reads so as not to impact I/O performance, and require little additional memory to store. On the other hand, visualization tasks demand support for low-latency reads, multiresolution queries, and spatial or attribute-based filtering. Supporting these queries natively in the output data through a more advanced data layout enables visualization applications to quickly load the data needed to compute a visualization, allowing them to compute visualizations faster and provide better interactivity, even for massive data sets. The additional time taken to

construct the data layout has been regarded as less of a concern for visualization tasks, as layout is constructed once ahead of time in a preprocess. High parallel I/O throughput is also less of a concern for post hoc visualization tasks, as they are typically run at much lower levels of parallelism than the simulation, i.e., on single laptops or workstations or visualization clusters.

These different demands placed on the I/O strategy and data layout by the simulation and visualization are in conflict with each other. Simulations demand high-throughput parallel writes at large scales, requiring just a simple data layout; however, visualization tasks demand low-latency multiresolution reads with support for spatial and attribute-based filtering at small to moderate scales, requiring a more complex data layout. Reorganizing the data into an existing visualization layout that supports low-latency multiresolution reads and spatial and attribute-based filtering adds overhead during I/O, and thus is typically not done as it would impact simulation performance. As a result, standard parallel I/O libraries used by simulations output the data as a raw dump with minimal additional metadata. Before post hoc visualization can be done, the raw output data must be converted to the visualization optimized data layout. Visualization can be done on the original output data, at the cost of reduced performance due to the less efficient reads performed and accessibility on lower power devices, as the entire full-resolution data set must typically be read to compute the desired visualization. Both the time spent building the data layout and the additional storage required to store it scale with the size of the data. Postprocess conversion of massive amounts of raw data to a visualization layout is made more challenging as visualization tasks typically have access to more modest computational capabilities than the simulation that produced the data. The cost of this step is already a concern today, and with simulation data sizes expected to grow at exascale, addressing this bottleneck becomes even more pressing.

1.2.2 Challenges in Visualization Across Parallel and Local Environments

A number of challenges exist today in the scientific visualization field, stemming from the need to render massive data sets at full resolution interactively or in situ at large scales, the desire to visualize large data in immersive environments to improve understanding and scientific workflows, and the goal of enhancing the accessibility of large data visualization.

The need for high-performance distributed rendering continues to grow, spurred by trends in increasing data set sizes, the desire for higher fidelity and performance, and the need for in situ visualization. Meeting these demands poses new challenges to existing distributed rendering strategies, requiring scalability across a spectrum of memory and compute capabilities on HPC systems. While the growth in data set sizes demands a large amount of aggregate memory, the desire for more complex shading and interactivity demands additional compute power. A number of applications have needs falling in between these two extrema, requiring a combination of memory and compute, or the need to render more complex data layouts that are not well supported by current compositing strategies. Finally, in situ visualization requires the renderer to scale with the simulation while incurring little overhead. Distributed rendering techniques that scale well for either memory capacity or compute power are well known; however, supporting the entire spectrum in between remains challenging, let alone doing so at large scales.

Prior work has demonstrated that the improved immersion provided by virtual environments such as CAVEs [60] can improve user performance on tasks related to spatial comprehension and search [157, 158, 226]. CAVEs consist of a room with multiple synchronized projectors or monitors driven by a set of powerful workstations to display the virtual environment. The recent growth in consumer VR systems has made high-quality immersive VR affordable and widely accessible. However, using such systems to visualize large data sets is challenging. Consumer VR systems have access to far less compute capabilities than CAVEs as they are driven by single gaming desktops, yet place stricter demands on application performance. Consumer VR applications must maintain consistently high framerates and low-interaction latency to avoid motion sickness. For visualization programs tasked with processing large volumetric data sets, this requirement poses a difficult challenge, requiring the adoption of real-time data processing throughout the application pipeline.

The web browser provides a standardized, cross-platform environment, making it a compelling platform to improve the accessibility of visualization applications. The browser also eases the process by which users access an application, removing the need to install and trust additional packages on their system. The browser has been widely used in the information visualization community to ease deployment for developers and improve accessibility for end users. However, scientific visualization applications rely on native

libraries (e.g.,VTK [249]), fast floating point operations, large amounts of memory, and parallel processing or high-performance rendering on the GPU or CPU. These capabilities were previously not available in the browser, forcing web-based scientific visualization applications that work with large data to rely on backend servers for processing. Using a server gives the application access to significant computational resources, at the cost of latency for users, and financial cost and maintenance effort to the developers. Recent developments in the browser have improved support for parallel GPU computing; however, making large-scale data visualization truly accessible requires algorithms that can work in the computationally constrained environment of the browser and on less capable hardware.

1.2.3 Challenges in Deploying Scalable, Low-Overhead In Situ Visualization

Loosely coupled in situ approaches have found growing interest in recent years, due to the flexibility provided as to how the visualization tasks can be run, which can be used to reduce the visualization's impact on the simulation. However, a number of challenges remain to provide scalable, low-overhead loosely coupled in situ visualization. A key benefit of loosely coupled in situ is the ability to run the simulation and visualization at different levels of parallelism, such that each can achieve the best performance. Visualization processes are typically more memory bound than the simulation, and actually running fewer processes where each has access to more local data will reduce the amount of network communication required, resulting in improved overall performance. For example, the simulation could be run with M processes while the visualization is run with N processes, where $M >> N$. However, to enable such $M : N$ configurations, current strategies rely on complex distributed memory query architectures or restructure the data during transfer. Both techniques provide a convenient single block of data to process to the visualization code, at the cost of data transfer performance. The simulation performance is also impacted, as the simulation cannot proceed to the next time step until the transfer has finished.

Furthermore, although restructuring-based approaches reorganize the data into a single block for each visualization process, they do not construct more advanced data layouts such as those used in post hoc visualization. Instead, current approaches are more similar to existing two-phase and subfiling I/O strategies, providing access only to the aggregated raw simulation data. For example, each group of $\frac{M}{N}$ simulation processes can be seen as

an I/O subgroup with a visualization process as the assigned aggregator. As a result, the in situ visualization tasks do not have access to more advanced data layouts that they may rely on or benefit from in a post hoc visualization scenario. The performance impact of not having access to these data layouts is similar to that discussed above for post hoc visualization, although interactivity and latency are less of a concern in an in situ use case.

1.3 Contributions

The aim of this dissertation is to address the key issues discussed above that arise in the simulation visualization pipeline when tasked with processing massive data, through an end-to-end approach based on adaptive and multiresolution techniques and data layouts. To achieve this end-to-end pipeline, this dissertation makes multiple contributions across I/O, data layout, and post hoc and in situ visualization. Fig. 1.2 illustrates how these contributions fit together in the broader scope of the end-to-end simulation visualization pipeline.

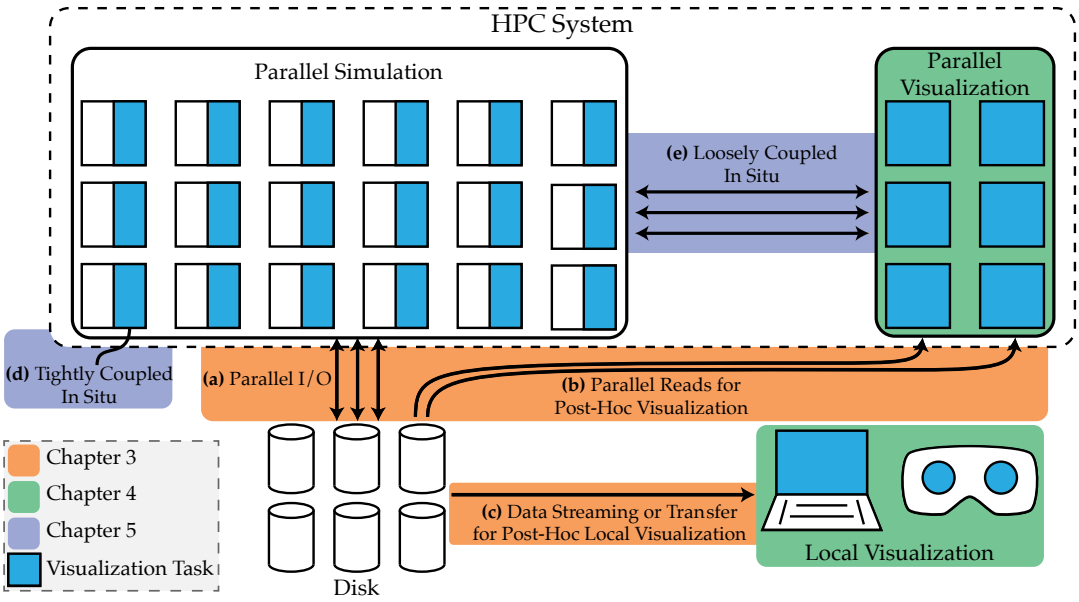


Fig. 1.2: An illustration of how this dissertation's contributions fit together in the broader scope of the end-to-end simulation visualization pipeline. The contributions discussed work together to support massive data sets across the simulation visualization pipeline.

1.3.1 Adaptive I/O and Data Layout

Chapter 3 presents work [154,276] on spatially aware adaptive I/O strategies for particle data sets that are capable of outputting the data directly in a visualization tailored layout, without sacrificing parallel write and read throughput at scale. These strategies propose novel load balancing strategies for I/O of nonuniform particle distributions to improve I/O performance and portability. The proposed strategies construct a spatial data structure over the domain, either a grid or k-d tree, to group ranks together to assign a similar number of particles to each output file. The load balanced grouping computed creates the subgroups used during two-phase I/O, ensuring roughly equivalent data movement costs for each group, and thus for each output file. The amount of aggregation performed is exposed as a tunable parameter for portability. Employing a spatial data structure to create the aggregation groups ensures that each file contains a spatially coherent subregion of data, providing an output layout well suited to efficient post hoc visualization reads. Moreover, this dissertation contributes a new particle data layout, the binned attribute tree (BAT), that balances between rendering and attribute-query focused access patterns. The BAT layout combines a spatial k-d tree with fixed size bitmap indices to support multiresolution spatial and attribute queries, while requiring little additional memory to store. The proposed layout is specifically designed to be constructed quickly in parallel during data writes and is split into a top-level tree and set of treelets. The BAT layout is constructed during the proposed spatially aware two-phase I/O pipeline to output the data directly in a visualization-focused layout, eliminating the data conversion bottleneck without sacrificing parallel I/O performance. The presented I/O approaches are evaluated on uniform and nonuniform particle distributions on four HPC systems to demonstrate their portability and scalability. Finally, an evaluation is conducted to demonstrate the BAT layout's suitability for low-latency multiresolution spatial- and attribute-based queries. The contributions of this chapter are:

- A fast semiadaptive approach for spatially aware I/O based on an adjustable uniform grid that supports a subset of nonuniform particle distributions;
- A fully adaptive approach for spatially aware I/O based on an aggregation tree that supports a wide range of nonuniform particle distributions;

- A novel combination of k-d trees and bitmap indexing to support multiresolution attribute and spatial queries with little overhead on particle data; and
- Parallel construction of the proposed data structure during spatially adaptive I/O with little overhead.

1.3.2 Post Hoc Visualization

Chapter 4 discusses a number of contributions that propose methods to enable visualization of massive data sets at large scales and in computationally constrained environments. The proposed works range from from full-resolution distributed rendering on HPC systems [274, 281] (Section 1.3.2.1), to adaptive and multiresolution methods that enable visualization in virtual reality [189, 277] (Section 1.3.2.2) and the web browser [279] (Section 1.3.2.3). The works are discussed in order of increasing data management complexity and computational constraints.

1.3.2.1 A Distributed FrameBuffer for Scalable Ray Tracing

Section 4.1 proposes the distributed framebuffer (DFB) [281], an asynchronous tile-based pipeline to accelerate image processing tasks in distributed renderers. The image processing tasks can be, for example, depth compositing, sort last compositing, tone mapping, and so on. The DFB decomposes the image processing tasks using a tile-based work decomposition, where the dependencies for each task are constructed per tile at runtime by the renderer. The task dependencies are not necessarily tied to the renderer's work or data distribution, providing the flexibility to implement a wide range of distributed rendering algorithms within a unified framework. The DFB performs all communication and computation in parallel to the renderer, enabling it to reduce overhead by eagerly completing tasks during rendering. Although the DFB is flexible enough to support a wide range of distributed rendering methods, it does not make a performance trade-off to do so. The contributions of this work are:

- An asynchronous tile-based pipeline for image processing tasks in distributed renderers built on flexible per tile task dependencies; and
- A set of distributed rendering algorithms built on this approach, covering standard use cases and more complex configurations.

1.3.2.2 A Virtual Reality Visualization Tool for Neuron Tracing

Section 4.2 presents a new virtual reality tool for neuron tracing in large connectomics data sets, VRNT, developed through a collaborative design study with neuronatomists [189, 277]. VRNT is built on an intuitive painting metaphor developed through this design study, which explored various interaction and rendering modalities to design an effective tool for neuron tracing in VR. Beyond interaction design, a core challenge faced in VRNT to enable immersive visualization of massive data sets is meeting the strict resolution and frame rate demands of VR. To do so, VRNT introduces a real-time page-based data streaming and rendering system, built on top of a multiresolution data layout. VRNT is evaluated through a pilot study conducted with trained neuroanatomists to compare neuron reconstruction accuracy and speed against industry standard tools. This work demonstrates that, given a high enough framerate and appropriate rendering and interaction techniques, the 3D interface provided in VR substantially improves the overall user experience by allowing neuroscientists to directly interact with their data. The contributions of this work are:

- A design study on using consumer-grade VR technology for neuron tracing; and
- A real-time page-based data processing framework that allows neuron tracing in data sets that are orders of magnitude larger than feasible with existing approaches.

1.3.2.3 Interactive Visualization of Terascale Data in the Browser

Section 4.3 presents a new GPU-parallel isosurface extraction algorithm, block-compressed marching cubes (BCMC) [279], for interactive isosurface computation in compute and memory constrained environments, such as the web browser. BCMC builds on a block-compressed volume data layout and novel GPU-driven memory management and caching strategy that allows working with compressed volumes on the GPU without CPU interaction. BCMC decompresses and caches blocks as needed to compute a given isosurface, taking advantage of the sparsity of typical isosurfaces to reduce its memory footprint. To accelerate computation, BCMC runs the entire isosurface extraction pipeline, consisting of vertex computation, data decompression, and cache management, on the GPU. By combining state of the art adaptive precision and resolution trade-offs in the data layout and adaptive processing for computation, BCMC is able to interactively isosurface up to 1B

voxels and 1TB of volume data, entirely in the browser. The contributions of this work are:

- A GPU-driven parallel isosurface extraction algorithm capable of interactive isosurface computation entirely in the browser; and
- A GPU-driven memory management and caching strategy that allows working with compressed data on the GPU without CPU interaction.

1.3.3 In Situ Visualization

Chapter 5 discusses work on adaptive and low-overhead approaches for in situ visualization. Section 5.1 discusses how the DFB ([281], (Section 4.1) can be applied to accelerate in situ rendering of image databases [275] for "explorable extracts" in situ methods such as Cinema [7]. The DFB supports rendering multiple images simultaneously, allowing in situ applications to render multiple camera viewpoints and scene configurations at the same time to more fully utilize the available compute resources. Section 5.2 discusses work in progress to integrate data-staging in situ visualization into the adaptive spatially aware I/O strategies contributed by this dissertation ([154, 276], Sections 3.1 and 3.2). Providing data-staging in situ visualization in the presented I/O pipelines allows the visualization to take advantage of the load balancing already performed for I/O, and to leverage the BAT layout built during I/O to accelerate spatial and attribute-based filtering. Providing the same data layout used post hoc to in situ visualization tasks also eases the process of bringing post hoc visualization tasks to an in situ context.

Finally, Section 5.3 presents a simulation-oblivious approach to data transport for loosely coupled in situ visualization [278, 280, 282], to minimize the impact of the visualization on the simulation. The presented approach adopts a simulation-oblivious data model that explicitly does not perform data restructuring during data transfer to minimize the amount of simulation time spent transferring data to the visualization. The presented approach is well suited to $M : N$ configurations, allowing the simulation and visualization to be run in their optimal configurations to improve overall performance, natively supports asynchronous and on-demand execution, is portable across a range of HPC or ad hoc cluster systems, and allows minimizing the impact of the visualization by executing it on different nodes than the simulation. The contributions of this chapter are:

- A simulation-oblivious data transfer model that minimizes the impact of in situ visualization on the simulation; and
- A data-staging in situ approach that enables the visualization to leverage the same data layouts available post hoc with little overhead.

CHAPTER 2

RELATED WORK

This chapter reviews prior work across the simulation visualization pipeline relevant to the work presented in this dissertation, from I/O and data layout to post hoc and in situ visualization. Section 2.1 discusses prior work on scalable I/O for particle data related to our proposed I/O strategies. Section 2.2 discusses prior work on visualization focused data layouts for massive data sets, covering both rendering and attribute-query optimized layouts. Section 2.3 discusses a range of works on interactive post hoc visualization for massive data, from scalable distributed rendering strategies on a cluster to immersive or web-based single user environments. Finally, Section 2.4 reviews work on in situ visualization in tightly coupled, data staging, and loosely coupled use cases.

2.1 Parallel I/O for Particle Data

Organizing and efficiently accessing large-scale grid-based structured data has always been the aim of several high-level parallel I/O libraries. The most prominent examples are Parallel HDF5 [115], Parallel NetCDF (PnetCDF) [137], ADIOS [177], and GLEAN [284]. To portably achieve high-bandwidth writes, current parallel I/O libraries use two-phase I/O [64,156,173,177,270,284] or subfiling [39,90]. Although the I/O strategies used in these libraries are general purpose, robust, and achieve high bandwidth, the output data layout is not well suited to the read access patterns required for interactive exploratory visualization of massive data sets. Current I/O strategies typically output the data in row-major blocks, and do not reorganize the data or create additional metadata to efficiently support the spatial and attribute queries performed by post hoc visualization tasks. As a result, post hoc visualization tasks may be forced to make inefficient reads on the data, impacting visualization performance. Parallel IDX (PIDX) [156] is a two-phase I/O library that outputs the data in a cache-oblivious, multiresolution layout using a spatially aware two-phase

I/O strategy. The data layout output by PIDX supports low-latency multiresolution reads, making it amenable to interactive post hoc visualization of massive data sets, even on low-power devices. PIDX has been demonstrated to provide portable, scalable I/O performance on structured grids [150,153,155,156] and AMR [151] at high levels of parallelism, while writing the data in a layout that natively supports multiresolution visualization queries.

Although these libraries are effective at dealing with structured data, they provide limited support for particle data, typically treating it as a set of one-dimensional (1D) arrays. For example, see HDF5's H5Part API for particle data [41,123], which provides a convenient wrapper over writing 1D arrays of data. Widely used approaches to I/O for particle data output either a single-shared file [40,41,123,178,225,259] or a file per process [21,191]. As observed previously on I/O for structured grids, such approaches can work well on a single HPC resource or provide reasonable performance at some scales, but struggle with large core counts or portability across different networks and parallel file systems. To address the limitations of single-shared file and file per process I/O, recent work [39,105] on scalable I/O for particle data began to leverage subfiling to improve scalability and portability.

Habib et al. [105] employed a subfiling approach for massive cosmology simulations (HACC) on the IBM Blue Gene/Q. Their approach grouped processes by their I/O node to create the subfiling subgroups and structured writes within the file to avoid file lock contention on the parallel filesystem. The subgroups are structured similar to those described by Vishwanath et al. [284] and Bui et al. [37] for efficient two-phase I/O on the IBM Blue Gene/P and Blue Gene/Q, respectively. Habib et al. demonstrated their approach on up to 1 trillion particles (47TB) on 262k cores, where they were able to write the data at 118GB/s. However, the subgroups created for subfiling are not guaranteed to output a single spatially coherent block of data [117]. As a result, post hoc visualization tasks performed on the data may need to access many individual files to load a spatial subregion of the data for processing. Habib et al. also provided an I/O strategy that outputs a single-shared file using PnetCDF to provide the data in a more widely used format. The single-shared file mode supports outputting metadata about the individual rank's block of data within the file, providing the spatial bounds of each block and its location within the file. However, the strategy does not merge blocks into larger ones or explicitly structure them in the file to improve coherence for spatial queries. Although the metadata can be

used to accelerate spatial queries, reads still need to access multiple disjoint regions of the file to read particles from all blocks overlapped by a query. Scalability is also a concern when outputting this format, as it can be written only in a single shared file mode.

Byna et al. [39] examined tuning HDF5's subfiling strategy on two particle I/O kernels. Data were written through HDF5's particle API, H5Part [123]. Although they found that subfiling I/O improved performance over single-shared file I/O, HDF5's subfiling strategy imposed significant limitations on post hoc visualization tasks, requiring the data to be read with the same number of processes as were used to write the data. As post hoc visualization tasks are typically run at much smaller scales than the simulation, this limitation prevents using the output files directly for visualization on workstations or smaller clusters. Moreover, as H5Part treats the particle data as 1D arrays of data, it is not able to output the particle data in a visualization tailored layout with metadata to accelerate spatial or attribute queries. Without this metadata, post hoc visualization tasks are forced to loop through and filter the particles to perform such queries, leading to poor performance.

The work presented in this dissertation addresses issues with current I/O strategies for particle data by developing adaptive and scalable I/O strategies for nonuniform particle distributions that are capable of outputting the data in a visualization-tailored layout.

2.2 Visualization Optimized Data Layouts

Post hoc visualization tasks are often performed on single workstations or small clusters, with access to far less compute and memory capacity than was used to run the simulation. Data access is driven by user exploration, often involving spatial and attribute subset queries [40,87,122,246,258,306,308,309]. Support for low-latency multiresolution reads and level of detail (LOD) is also desirable [82,87,106,122,246,247,251] to enable visualization on low-power devices, or while additional data are loaded. However, as discussed above, the raw data output by the simulation typically does not include the metadata or structures required to efficiently support such queries. Post hoc visualization tasks must either perform a lengthy postprocess conversion step to transform the data into the desired visualization layout, or tolerate poor read performance on the data. The cost of the conversion, in terms of compute time and additional storage needed, scales with the size of the data, making the process challenging or infeasible for massive data sets.

A large number of rendering-focused (Section 2.2.1) and attribute-query focused (Section 2.2.2) data layouts have been proposed in the visualization and scientific data management communities. Rendering-focused layouts typically provide support for read access patterns involving spatial filtering, low-latency multiresolution reads, LOD, or progressive reads. Supporting some or all of these operations is needed to enable efficient on-demand or progressive loading for interactive applications, and to enable interactive visualization of large data sets on low-power or remote devices. Attribute-query focused layouts, on the other hand, treat the data more akin to entries in a database, applying ideas from the database community to support efficient attribute-based filtering operations. Such layouts support efficiently querying and returning the subset of data satisfying a desired attribute filter, for example, "$x > 5$ and $temperature < 100$", to retrieve data of interest for visualization. Although prior work has focused on optimizing the data layout for either rendering or attribute queries, relatively little work has been proposed to satisfy both domains. The data layout proposed in this dissertation takes a step in this direction, proposing a layout that balances between these two use cases while incurring little memory overhead. Furthermore, the proposed data layout is specifically designed to be built quickly in parallel during I/O, thereby eliminating the need for a postprocess data conversion while incurring little computational impact on the simulation.

2.2.1 Rendering-Focused Layouts

A wide range of rendering-focused layouts have been proposed, providing support for spatial filtering, LOD, and low-latency multiresolution reads. Rendering-focused layouts are typically built on top of spatial hierarchies (Fig. 2.1), for example, *k*-d, octree, and other tree-based hierarchies [57, 58, 76, 87, 93, 111, 112, 121, 122, 147, 233, 247, 251, 296, 306], bricking [82, 85, 106, 176, 227, 246], and other hierarchical layouts [22, 215, 293]. The spatial hierarchy can be used to accelerate spatial queries on the data, or to enable on-demand loading by filtering out data that are outside the region of interest. The hierarchy can also be augmented with various forms of LOD information to provide fast access to coarse representations of the data to enable low-latency multiresolution reads and rendering on low-power devices. The spatial hierarchies constructed can also be used to accelerate rendering algorithms to display the data.

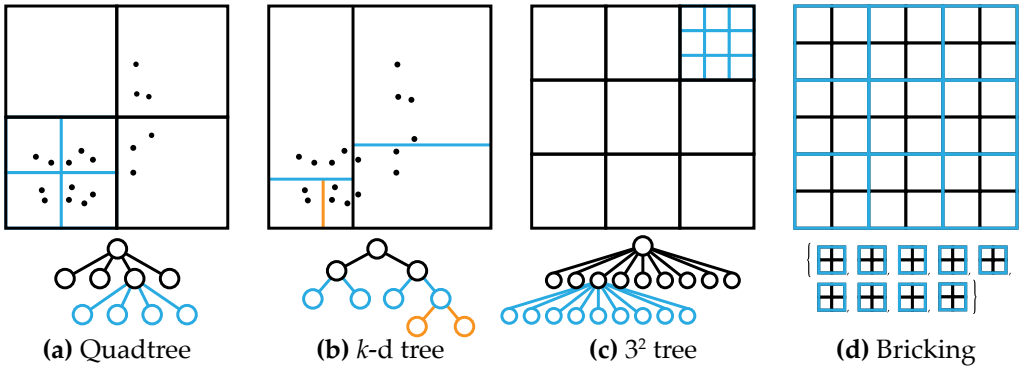


Fig. 2.1: 2D examples of the spatial structures widely used in rendering-focused layouts. (a) The Quadtree [81] employs a fixed subdivision where regions can be recursively subdivided into 2×2 children if needed. The Quadtree extends to an octree [303] in 3D, refining each region into 2^3 children. (b) The k-d tree [18] is a binary tree that partitions data along a line on one axis (in 3D, a plane) at each level. The line (or plane) can be placed at the median or mean of the objects, or a different location, to best partition the data. (c) The N^3 tree [57] (in 2D, N^2) is a generalization of the octree that divides each region into N^3 children, allowing for wider fanout and shallower trees. (d) Bricking is a nonhierarchical layout that divides a grid (e.g., a regular grid volume) into a coarser grid of bricks of some size. The coarser grid can be used to quickly load data from specific subregions. For additional discussion and theoretical results on spatial data structures, see the excellent book by Samet [241].

A concern with approaches that construct and store additional hierarchical or LOD information is the additional memory required to store this information, as this cost typically scales with the size of the original data set. Prior work has adopted compression or quantization [87,122,127,128,147,246,296] to reduce memory overhead, or compensated for this overhead by discarding subsets of the data [306]. Implicit hierarchy approaches [215, 236] have been proposed to eliminate the memory overhead of storing a hierarchy and additional LOD information. Implicit hierarchy approaches reorder the data to form a hierarchical layout, for example, using a Z-order [236] or hierarchical Z-order [215] curve. Implicit approaches can eliminate the memory overhead of storing the hierarchy, but at the cost of losing support for certain rendering queries or restricting the quality of LOD that can be provided.

Data layouts based on octrees [93,121,147], N^3 trees [57,58,76,111,112,168,296], and bricking [82,85,106,176,227] have been used effectively for massive volumetric data sets (also see the survey by Beyer et al. [22]). These layouts are fundamentally based on voxel or grid hierarchies, or a grid-based partitioning, and map well to regular grid, octree, and other grid-based volumetric data sets. Implicit hierarchies have also been proposed for

volumetric data, by reordering the voxels using a hierarchical Z-order curve [215]. The support for spatial filtering, LOD, and low-latency multiresolution reads provided by these layouts has been used extensively to enable interactive rendering of massive volumes that do not fit in CPU or GPU memory.

Octree layouts have been used effectively to build hierarchies over a bricked volume, or to directly represent the volume data as a multiresolution hierarchy. Hong et al. [121] presented an approach to rendering large regular grid volumes by bricking the data and constructing a min-max octree [303] over the bricks. The min-max octree is used to load bricks onto the GPU on-demand for rendering while ignoring those that are not visible, due to either the camera position or the transfer function. Knoll et al. [147] proposed a min-max octree-based approach to render implicit isosurfaces of large volumes that merged voxels with similar or uninteresting values to build the octree and compress the data. The resulting octree required far less memory to store than the original data and could be used to accelerate ray tracing, enabling interactive isosurface rendering on consumer systems that would otherwise be unable to fit the data in memory. Gobbetti et al. [93] proposed an octree based on-demand loading approach for rendering large volumetric data on the GPU similar to that of Hong et al. Gobbetti et al. stored downsampled representations of the bricks in the octree inner nodes to support LOD rendering based on the camera position, whereas Hong et al.'s approach supported rendering the bricks only at full resolution.

N^3 trees represent a superset of octrees (an octree is a 2^3 tree), allowing for shallower trees with more data stored at each level [168]. N^3 trees are especially well suited to representing regular grid volumes, as the shallower hierarchy reduces the memory overhead required to store it. Crassin et al. [57,58] introduced Gigavoxels for out-of-core rendering of massive sparse grids. The Gigavoxels layout consists of an N^3 tree where each node in the tree can reference a 3D brick or constant value. Bricks referenced by inner nodes are created by downsampling those of the children to provide a multiresolution layout. Gigavoxels can be seen as an extension of the octree layout proposed by Gobbetti et al. [93] to support wider branching factors in the tree, require less memory, and more efficient streaming during rendering. Engel [76] presented an extension of Gigavoxels to enable progressive rendering of dense data sets. Harel et al. [111,112] presented the hyper tree grid, which provides a two-level data layout to support a broad range of tree-based adaptive mesh refinement

simulations. The hyper tree grid consists of a top-level grid, where within each cell a hyper tree is stored. A hyper tree is a N^3 where $N = 2$ or $N = 3$. Harel et al. provided methods for query-driven and multiresolution reads of the data using cursors, which were used to enable LOD rendering of large data sets [69]. Wang et al. [296] proposed the brick tree layout, which provided an efficient N^3 tree layout for low-latency multiresolution rendering of massive scientific grid data sets. Wang et al. adopted a similar approach to the hyper tree grid to support noncubic or massive data sets by constructing a grid of brick trees. The brick tree is also able to compress the data by merging bricks with similar values.

Brick-based layouts rearrange the volume into a grid of bricks, removing the need to store a tree or to perform tree traversals when loading the data. Bricking has proven especially useful for managing massive regular grids, as the layout can be constructed by partitioning the volume into subregions and reorganizing the voxels. Prohaska et al. [227] proposed a brick-based layout using HDF5's chunked file mode to enable visualization of large micro-CT scans. For each brick, they constructed and stored multiple coarser resolutions to provide low-latency multiresolution reads. Data are fetched from a remote server by a visualization client and reorganized into an octree for rendering. Prohaska et al. used view-dependent refinement to request higher resolution bricks from the server. Ljung et al. [176] proposed a similar multiresolution brick layout, wherein the bricks are compressed and decompressed during rendering based on the transfer function variance in the brick. The Tuvok architecture presented by Fogal et al. [82] employed a multiresolution brick layout similar to that of Prohaska et al. [227]; however, Tuvok supported on-demand streaming and caching of bricks to render data sets larger than GPU memory. Work by Hadwiger et al. [106] and Fogal et al. [85] proposed ray-guided methods for on-demand streaming of massive multiresolution brick-based layouts. Both approaches are based on a virtual memory technique. As the renderer traverses the volume, it attempts to load data for processing. If the required resolution brick is missing, a cache miss is generated and the brick is uploaded in the next frame. Whereas Hadwiger et al. [106] performed a virtual address lookup for each sample taken by the renderer, Fogal et al. [85] performed this lookup per brick and then computed all needed samples for the brick. The GPU-driven parallel isosurface extraction algorithm proposed in this dissertation is based on a compressed brick layout of the volume data built using ZFP [171]. Bricks are decompressed and cached on

demand as needed to compute the isosurface through a GPU-driven cache.

Pascucci and Frank [215] proposed an implicit hierarchy approach for volume grids by reordering the voxels using a hierarchical Z-Order curve. Voxels along progressively finer Z-Order curves are stored in the file to reorder the data, constructing an implicit spatial and resolution hierarchy. Low-latency multiresolution reads are performed by progressively reading additional data from the file, while the spatial locality of the Z-Order curve can be used for spatial filtering. Their approach is able to reduce memory overhead by not constructing explicit coarser resolutions of the volume and not building explicit tree or other hierarchies over the data. This approach forms the basis of the IDX file format, which is used in the VR Neuron Tracing tool proposed in this dissertation to provide efficient access to large connectomics volumes.

The ability of tree-hierarchies' ability to adapt to nonuniform data distributions has made them an effective data layout for particle data sets. Data layouts based on *k*-d trees [128, 238, 306], BSP trees [217], cluster hierarchies [122], and octrees [8, 62, 75, 86, 87, 127, 141, 208, 209, 246, 247, 251, 252, 304] have been used effectively for massive particle data sets. As was the case with data layouts for volumetric data, the overhead required to store the hierarchy or introduce new data for LOD scales with the data set size, motivating the development of low or zero overhead implicit hierarchy approaches [236], along with the adoption of compression, quantization, and other memory reduction techniques [87, 122, 128, 238, 246, 306].

Particle data layouts based on *k*-d trees [128, 238, 306], BSP trees [217], and cluster hierarchies [122] have been employed as the data structures' flexibility enables them to construct an efficient and balanced hierarchical partitioning, even for highly nonuniform distributions of data. Rusinkiewicz and Levoy's seminal work on QSplat [238] achieved multiresolution point-based surface rendering using a bounding sphere hierarchy. The hierarchy is constructed by first building a midpoint-split *k*-d tree, where inner nodes represent their children by a bounding sphere enclosing them. However, the final data layout used in QSplat is not a binary tree. After constructing the *k*-d tree, inner nodes of the tree are merged to produce a shallower tree with a wider branching factor. Hubo et al. [128] presented a method for constructing a quantized *k*-d tree for ray tracing of large point clouds where split plane and point positions are quantized to the bounding box of the

inner node. Woodring et al. [306] presented an approach for constructing a *k*-d tree with LOD information in situ for cosmology simulations. The *k*-d tree could then be saved to disk and used directly in post hoc visualization to provide multiresolution reads. However, they generated the coarser levels of detail by sampling and duplicating subsets of particles up the tree. As a result, their approach doubled the amount of particles that must be stored, which they alleviated by optionally discarding some levels of the tree. The remaining levels were found to still provide accurate approximations of the data for visualization. Pauly et al. [217] described an approach for constructing a BSP tree over a point cloud based on principal component analysis (PCA) [138]. The plane partitioning the points at each level is placed at the centroid of the set of points and oriented along the largest eigenvector of their covariance matrix. Hopf and Ertl [122] presented a similar PCA-split tree build for cosmology data, although they used the PCA-split process to construct a tree with a wider branching factor. Hopf and Ertl produced a cluster hierarchy consisting of multiple clusters at each level, where each cluster was split into a variable number of children on the subsequent level. Clusters on a level were repeatedly split as described by Pauly et al. [217] until a target minimum distortion level was reached. Subsequent levels were produced by decreasing the minimum distortion level and repeating the splitting process. A concern with BSP tree-based approaches compared to those using more fixed subdivison schemes (i.e., *k*-d trees or octrees) is the additional computational cost required to determine the splitting planes.

Octrees have been widely used [8, 62, 75, 86, 87, 127, 141, 208, 209, 247, 251, 252, 304] for rendering-focused layouts for particle data coming from scientific simulations and LIDAR scans. The wider branching factor provided by an octree can reduce traversal overhead, while the implicit multiresolution gridding provided by the octree can be used to produce voxelized or LOD representations. However, the fixed subdivision structure of octrees may not adapt as well to highly nonuniform distributions, leading to poor quality trees. Pajarola [208] proposed an octree layout and LOD scheme for point-based surface rendering by storing the average of the points at each level in the inner nodes of the octree. Pajarola et al. [209] later extended this approach to support out-of-core rendering of massive data sets by flattening the octree out to a sequential list of points grouped by their LOD level, similar to the sequential point tree layout of Dachsbacher et al. [62]. To support efficient

out-of-core rendering of large data sets, the list of points was grouped into blocks by LOD layer on disk and aligned to page boundaries for efficient access via memory mapping. Schütz et al. [252] proposed a similar approach for rendering out-of-core LIDAR data sets in VR, where a point's LOD level was offset by a psuedorandom value to provide smoother transitions between LOD levels. Fraedrich et al. [86, 87] constructed an octree over a 10B particle Millennium Run cosmology simulation to achieve interactive rendering on GPUs. LOD was provided by merging points up the tree based on their visible size. To reduce memory use, particle attributes in the tree were quantized after a logarithmic remapping to reduce the error introduced by quantization. Fraedrich et al. achieved efficient disk access by grouping nodes in the tree to form a "page tree", where nodes in the original octree that shared a common parent or ancestor were grouped on disk to form larger blocks of memory. Scheiblauer [247] proposed the modifiable nested octree, which created LOD by storing a subsampling of the points at inner node during construction. The points stored in the inner nodes are not duplicates of those at finer levels, thereby reducing memory overhead. Schütz [251] further extended this approach to support out-of-core rendering and streaming to render massive LIDAR data sets in the browser. Schütz split the octree up into chunks that group nodes sharing a common parent and their children down to a configurable number of levels into a file, reducing the amount of data that must be loaded for spatial queries or coarser LOD.

Implicit hierarchy approaches [236] have been proposed to reduce the overhead of storing hierarchies for massive particle data sets. As with implicit approaches for volumetric data [215], implicit approaches for particles are based on resorting the data to produce a new list of points in the desired LOD ordering. Rizzi et al. [236] proposed an approach for particle data similar to that proposed by Pascucci and Frank [215] for volume data. The particles are first sorted in Z-order, after which they are reorganized into multiple levels of detail by sampling particles along the curve. The resulting ordering is similar to a hierarchical Z-order [215]. Their approach provided low-latency multiresolution reads by loading successively larger subsets of the data from disk.

The Binned Attribute Tree layout proposed in this dissertation balances between rendering-focused layouts and attribute-query focused layouts. The tree is based on a spatial k-d tree [18], where subsets of points are sampled during construction at each inner node to

store for LOD without requiring duplication or the creation of representative particles. The tree is built in two parallel stages on each aggregator during I/O, a coarse parallel bottom-up build [144] and multiple treelets built top-down in parallel, and is stored on disk in a layout designed to support efficient access to the treelets via memory mapping. Finally, the nodes of the tree track fixed size bitmap indices, discussed in the following section, to support efficient attribute-based filtering with little memory overhead.

2.2.2 Attribute-Query Focused Layouts

Simulation data follow a write-once read-many usage model, for which bitmap indexing approaches have been demonstrated to be effective for accelerating attribute queries [27, 40, 42, 43, 49, 97, 98, 145, 205, 237, 258, 265, 267, 268, 305, 308–312, 326, 328]. Bitmap indexing approaches treat the individual items in the data set, i.e., the voxels, particles, mesh elements, etc., as individual entries in a database, and construct a bitmap index [205] over each attribute to accelerate queries. A bitmap index consists of multiple bit vectors, where each item in the data set is represented by a bit in each vector. Attribute queries can then be answered by performing bitwise Boolean operations on the bit vectors to quickly find the items satisfying the query. Each bitmap index encodes information about a single attribute in its set of bit vectors, requiring a different index for each attribute. Queries over multiple attributes, for example, "$x > 5$ and $pressure < 10$", are answered by performing the individual queries separately to produce bit vectors marking the items satisfying each query, and then combining these bit vectors with a bitwise `AND`.

The bit vectors that make up the bitmap index can be used to encode different information about each item's attribute value, depending on which queries will be performed on the data or the cardinality of the data attributes. Equality encoding [205] is a straightforward method for encoding the attribute information. Given an attribute with cardinality C and a data set with N items, C bit vectors are created, each with N bits (Table 2.1a). Bit i of bit vector E^j is set if item x_i has the j'th attribute value. Equality encoding is optimal for equality queries, e.g., "$x == 4$", as only one bit vector must be read to answer the query. However, answering range queries, e.g., "$x \geq 3$" or "$2 \leq x \leq 4$", requires reading and `OR`'ing together as many bit vectors as there are attribute values overlapped by the query. Range encoding [305] encodes whether an item's attribute is within a given subrange of

Table 2.1: Different bitmap index encodings for an attribute with cardinality $C = 6$.

(a) Equality bitmap encoding.

x	E^0 $\{0\}$	E^1 $\{1\}$	E^2 $\{2\}$	E^3 $\{3\}$	E^4 $\{4\}$	E^5 $\{5\}$
0	1	0	0	0	0	0
3	0	0	0	1	0	0
1	0	1	0	0	0	0
2	0	0	1	0	0	0
4	0	0	0	0	1	0
2	0	0	1	0	0	0
0	1	0	0	0	0	0
5	0	0	0	0	0	1

(b) Range bitmap encoding.

x	R^0 $[0,0]$	R^1 $[0,1]$	R^2 $[0,2]$	R^3 $[0,3]$	R^4 $[0,4]$
0	1	1	1	1	1
3	0	0	0	1	1
1	0	1	1	1	1
2	0	0	1	1	1
4	0	0	0	0	1
2	0	0	1	1	1
0	1	1	1	1	1
5	0	0	0	0	0

(c) Interval bitmap encoding.

x	I^0 $[0,2]$	I^1 $[1,3]$	I^2 $[2,4]$	I^3 $[3,5]$
0	1	0	0	0
3	0	1	1	1
1	1	1	0	0
2	1	1	1	0
4	0	0	1	1
2	1	1	1	0
0	1	0	0	0
5	0	0	0	1

(d) Binned bitmap equality encoding.

x	B^0 $[0,2)$	B^1 $[2,4)$	B^2 $[4,5]$
0	1	0	0
3	0	1	0
1	1	0	0
2	0	1	0
4	0	0	1
2	0	1	0
0	1	0	0
5	0	0	1

possible values using $C - 1$ bit vectors (Table 2.1b). Bit i of bit vector R^j is set if item $x_i \in [0, j]$. Range encoding can answer equality and range queries by reading at most two bit vectors. Interval encoding [43] encodes whether an item's attribute is within a given subinterval of possible values using $\lceil \frac{C}{2} \rceil$ bit vectors (Table 2.1c). Bit i of bit vector I^j is set if $x_i \in [j, j + m]$ where $m = \lfloor \frac{C}{2} \rfloor - 1$. As with range encoding, interval encoding is able to answer equality and range queries by reading at most two bit vectors; however, interval encoding is able to nearly halve the size of the bitmap index.

When considering attributes with very high cardinality, for example, floating point scientific data, a massive number of bit vectors must be stored to create the bitmap index. To address this, the attribute range can be partitioned into a set of bins, reducing the number of values that must be represented in the bitmap index. The resulting binned encoding [237] can be configured to represent the attributes using as few or as many bit vectors as desired

by adjusting the size of the bins. The bitmap index can then be constructed over these bins using an equality, range, or interval encoding. A binned equality encoded bitmap index is shown in Table 2.1d. However, binning comes at the cost of query performance. If the query ranges do not lie at the bin boundaries, false positives found in the edge bins that are only partially contained in the query must be filtered out. Sinha and Winslett [258] introduced "multiresolution bitmap indices," to reduce the amount of bit vector data that must be read to answer a query. Their approach created multiple levels of binned bitmap indices at different binning granularities. Queries begin by accessing the bitmap index created at the coarsest binning to find the items contained in the query. Items that fall into bins that are entirely inside or outside the query can be kept or discarded without accessing additional bit vectors. Finer levels of bitmap indices are loaded to filter items whose attribute value is contained within an edge bin at the current level. To avoid a terminology conflict with "multiresolution reads," we refer to this approach as "multilevel bitmap indexing." Wu et al. [311] performed an analysis of various bitmap encoding schemes, and proved that specific two-level encoding strategies are optimal for answering range and equality queries, compared to single or n-level indices and other encoding schemes.

Wu et al. [309] demonstrated the effectiveness of bitmap indexing at accelerating queries for feature tracking, region growing, and region tracking tasks on 2D uniform grid autoignitation data sets. Their approach employed a binned range encoded bitmap index, which was compressed to reduce overhead. Compared to tree-based multidimensional indexing methods (e.g., R-Trees [104], B-Trees [14], k-d Trees [18]), bitmap indexing is well suited to data sets with high dimensionality, as the size of the index scales linearly with dimensionality, whereas query cost scales with the number of attributes being queried. Furthermore, the bitmap index can be constructed without reorganizing the underlying data. Wu et al. [312] further demonstrated their approach on multidimensional range queries for high-energy physics data sets. Stockinger et al. [264, 265] leveraged bitmap indexing through an early version of FastBit [308] to accelerate range queries on 3D volumetric data sets with many attributes. They demonstrated the effectiveness of their approach on range queries to select voxels of interest for processing, for example, to accelerate finding the voxels containing an isosurface to accelerate marching cubes [179]. Gosink et al. [97] integrated support for bitmap indexing into HDF5 to provide a new attribute-query-based

API for HDF5, referred to as HDF5-FastQuery, also using an early version of FastBit [308]. They demonstrated significantly improved attribute-query performance over prior work that integrated R*-trees for indexing into HDF5 [200]. Gosink et al. [98] applied multilevel bitmap indexing to accelerate attribute queries on time-varying AMR data sets. They employed a two-level bitmap index that was processed on the GPU to further accelerate query response times.

Although bitmap indices have proven highly effective at accelerating attribute queries on scientific data sets, they are not without limitations. A key concern when using bitmap indices is the size occupied by the index. Although compression methods for bitmap indices have been proposed [308,310], the indices can still require a significant amount of storage. Prior work has reported the bitmap index occupying an additional 30% [268] to 66% [49] of the original data size, and in some cases approaching the original data in size [40,48] or exceeding it [313]. Bitmap indexing also faces difficulties when encoding high cardinality attributes. Whereas binning and multilevel binning can address the need to otherwise store an enormous amount of bit vectors, using binned bitmaps impacts query performance by introducing false positives that must be filtered out. More advanced binning strategies have been proposed [313] to address this "curse of cardinality"; however, the resulting bitmap index requires significantly more storage than a regular binned bitmap index, and can even exceed the original data in size.

As with rendering-focused layouts, the time taken to construct the bitmap index post hoc is another main concern. The construction cost scales with both the number of items and the number of attributes over which the bitmap index is built, posing a challenge for massive data sets. Prior work has proposed to accelerate construction by building the index in parallel on an HPC cluster [27,49,145], or in situ [268,326,328] to eliminate the need for a lengthy post hoc build process. However, the simulation must now write the bitmap index in addition to the raw data. The index can become quite large, which can place a significant additional I/O burden on the simulation.

The binned attribute tree layout proposed in this dissertation provides support for attribute queries through a fixed 32-bit binned bitmap index using equality encoding. Rather than represent each item explicitly in the index, the presented approach indexes only the inner and leaf nodes of the k-d tree to reduce memory use further. Which nodes

contain particles within the query can then be determined efficiently during tree traversal by computing the bitmap representing all bins overlapped by the query. Each node is tested for overlap with the query by computing a bitwise `AND` of its bitmap and the query bitmap to determine if any of its children are contained in the query. Although the presented approach is intentionally limited to 32 bins, reducing query accuracy somewhat, this allows a significant reduction in the size of the bitmap index and to efficiently check queries through a simple bitwise `AND`.

2.3 Post Hoc Visualization of Massive Data Sets

The work in this dissertation touches on a range of areas in post hoc visualization, both on HPC systems and locally on workstations or laptops. Work related to the contributions presented in this dissertation are discussed categorized by the domain in which they operate: scalable distributed rendering on HPC systems (Section 2.3.1) and locally on workstations or laptops (Section 2.3.2).

2.3.1 Scalable Distributed Rendering

A large body of previous work has studied parallel rendering techniques for distributed-memory systems. These works can generally be classified as one of three techniques, first discussed in the context of rasterization by Molnar et al. [193]: sort-first, sort-middle, and sort-last. Sort-middle is tied to rasterization, since the rendering methods contributed in this dissertation are based on ray tracing, the discussion below focuses on sort-first and sort-last strategies. Sort-last is a data-parallel technique with which the workload is distributed by subdividing the data set across multiple ranks (Section 2.3.1.1). Sort-first is an image-parallel technique with which the workload is distributed across multiple ranks by subdividing the image (Section 2.3.1.2). Hybrid approaches have also been proposed that combine sort-first and sort-last techniques (Section 2.3.1.3).

2.3.1.1 Data-Parallel Rendering

Sort-last, or data-parallel, rendering is widely used to render massive data sets on HPC systems. In a data-parallel renderer, the geometric primitives and volumetric data are partitioned in three dimensional (3D) space, with each node assigned a subregion of the overall data to render. Each node then renders its subregion of data to produce a

partial image, which must then be combined with other nodes' partial images to create the final image. Combining these partial images typically requires depth compositing the overlapping partial images to produce a correct final image. It is this second step that becomes the bottleneck at large core counts and high resolutions, and therefore has been the focus of a large body of work (e.g., [72, 100, 101, 125, 183, 222, 320]). Although scalable rendering methods for massive data have been proposed, standard data-parallel renderers impose restrictions on how the data can be distributed among the nodes, are susceptible to load imbalance, and are limited to local illumination effects.

Early work by Hsu [125] and Neumann [202] proposed approaches where each rank rendered the pixels its local data projected to and sent those pixels to another rank that owned the pixel for compositing. The compositing step of this algorithm is referred to as direct send compositing. However, this approach can require a large number of messages to be sent over the network and, in the worst case, all ranks to communicate with each other. Ma et al. [183] proposed binary swap compositing, a bulk synchronous compositing process that ensured all ranks participated in all steps to parallelize the compositing work. The volume is distributed among the ranks using a k-d tree decomposition, allowing the final image to be produced by traversing the tree from bottom to top and compositing the images from a node's children. The compositing work is distributed among all ranks at each level by assigning each a successively smaller subregion of the entire image that it is responsible for compositing. However, a limitation of binary swap is that the number of ranks involved in the compositing process must be power of 2. Yu et al. [320] proposed 2-3 swap compositing, which generalized binary swap to support nonpower of 2 configurations. 2-3 swap employs a compositing tree where each node can have two or three children, allowing the tree to be built with any desired number of leaves. Nodes with three children divide the compositing work into thirds among the ranks, while those with two children proceed as in binary swap. In contrast to binary swap, which ensured that each rank must only communicate with one other rank at each level, the variable sized partitioning employed in 2-3 swap requires ranks to communicate with at most two other ranks at each level. Peterka et al. [222] proposed Radix-k, which further generalized the decomposition employed by 2-3 swap to support variable-sized compositing groups. Radix-k factors the n ranks into r rounds of compositing, $n = \prod_1^r k_i$. Each round consists of $\frac{n}{k_i}$ groups with

k_i ranks each exchanging image data. In each subsequent round, each rank exchanges its image data with more distant neighbors. In the first round ranks are direct neighbors, in the second they are k_1 apart, in the third $k_1 \times k_1$ apart, and so on.

Early work on compositing was developed in a single core or rank per core execution model on HPC systems and achieved parallelism through bulk synchronous strategies that swapped between communication and computation stages. However, current systems have tens to hundreds of cores per rank, or GPUs with thousands of cores. Applications can leverage multiple threads on a node to eliminate the overhead of on-node message passing between processes, and can overlap communication and computation using asynchronous messaging facilities. Howison et al. [124] demonstrated that a hybrid approach, with threads for parallelism on a node and MPI for parallelism across nodes, outperformed traditional rank per-core methods. Grosset et al. [100, 102] presented the task-overlapped direct send tree (TOD tree) compositing method, which was designed specifically for HPC systems with multicore CPUs and GPUs. TOD tree aimed to overlap communication with computation, and to reduce communication overall, to reduce the additional time spent compositing after local rendering had finished on each node. Grosset et al. split the compositing process into three stages. First, ranks are divided into groups of m ranks, within which they perform direct send compositing. At the end of this stage, each rank holds the composited image of its group for a $\frac{1}{m}$ subregion of the image. The second stage composits the partial subregions from each group by creating m groups, each with the $\lfloor \frac{n}{m} \rfloor$ ranks that contain the partial subregion computed in the previous stage. Compositing in the second stage is performed using a k-ary compositing tree, where ranks send their data to a designated receiver at each level that composites the received images. At each stage, communication is performed using asynchronous MPI to overlap communication with other computation performed by the receiver or sender.

Most similar to the work presented in this dissertation on the Distributed FrameBuffer when applied to data-parallel rendering is the dynamically scheduled region-based compositing approach proposed by Grosset et al. [101]. DSRB divides the image into strips and constructed a per-strip blending order, referred to as a chain, based on the data that projected to each strip. Each node renders its local data to the entire image, after which it composites strips of its partial image with the corresponding chain. Partial compositing for

a chain start after receiving strips from successive nodes in the chain, thereby overlapping compositing with rendering on other nodes. Restricting the compositing dependencies to strips of the image has the additional benefit of removing nodes whose data does not touch some strip from the chain. However, DSRB is restricted in the amount of rendering it can overlap with compositing, as each node renders its entire image before starting compositing; is applicable only to data-parallel rendering; and relies on a central scheduler to construct the chains.

The IceT library [194] encompasses several different compositing strategies for sort-last rendering and has been widely deployed across popular parallel scientific visualization tools. Although IceT was initially designed for rasterization, Brownlee et al. [36] used IceT's depth compositing with a ray tracer inside of multiple visualization tools, although they were hindered by the data distribution chosen by the tools. Wu et al. [314] employed a similar approach to integrate OSPRay into VisIt, using OSPRay to render locally on each rank and IceT to composite the image, and encountered similar difficulties.

2.3.1.2 Image-Parallel Rendering

Image-parallel renderers assign subregions of the image to different ranks for rendering. To render large data sets, image-parallel rendering is typically coupled with some form of data streaming or movement into a local cache, and is designed to exploit frame-to-frame coherence. The data movement work can thus be amortized over multiple frames, as the data rendered for a region of the image in one frame will likely be similar to that rendered in the next frame. Early rasterization-based techniques used a sort-middle algorithm, where the image was partitioned between ranks and the geometry sent to the rank rendering the portion of the image it projected to [73].

Image-parallel rendering lends itself well to ray tracing, and to ray tracing out-of-core data as ray tracers already process the image pixels in parallel and use acceleration structures for ray traversal that can be adapted to stream and cache portions of the scene on-demand as they are traversed. Image-parallel work distributions have been used effectively for distributed rendering of replicated data in a number of ray tracers, for example, Manta [26], OpenRT [288], and OSPRay [291]. Although typically used for data that can be stored in memory on each node, this architecture can be used to render larger data sets by streaming

the data needed for the portion of the image rendered by each worker from disk [293] or over the network [65, 133]. However, methods for image-parallel rendering of large data can suffer from cache thrashing and are tied to specific data types or layouts.

Wald et al. [293] used a commodity cluster for interactive ray tracing of massive models, where a top-level *k*-d tree was replicated across the ranks and lower level subtrees referencing the geometry fetched from disk on-demand. DeMarle et al. [65] used an octree acceleration structure to render large volumes, where missing voxels were fetched from other ranks using a distributed shared memory system. Ize et al. [133] extended the approach of DeMarle et al. to geometric data, using a distributed BVH. When rendering fully replicated data, Ize et al. were able to avoid data movement and compositing, and achieved 100FPS for primary visibility ray casting on 60 ranks. Biedert et al. [24] proposed an image-parallel remote streaming and auto-tuning tiling framework that achieved up to a 2× performance improvement by improving the load balance of the rendering workload among the ranks.

2.3.1.3 Hybrid-Parallel Rendering

Wheras image- and data-parallel rendering methods distribute work solely by partitioning the image or data, hybrid-parallel renderers combine both strategies, aiming to pick the best distribution for the task at hand. Reinhard et al. [234] first proposed a hybrid scheduling algorithm for ray tracing distributed data, where the rays would be sent or the required data fetched depending on the coherence of the rays. Samanta et al. [240] proposed to combine sort-first and sort-last rendering in the context of a rasterizer, by partitioning both the image and data among the nodes. Each node then rendered its local data and sent rendered pixels to other nodes that owned the tiles its data touched. The tiles were then composited on each node and sent to a display node. This approach bears some resemblance to the Distributed FrameBuffer proposed in this dissertation, although it lacks the DFB's extensibility and support for ray tracing specific rendering effects.

Navrátil et al. [201] proposed a scheduler that combined static image and data decompositions for ray tracing, roughly similar to sort-first and sort-last, respectively. However, a key difference of their approach when compared to a sort-last rasterizer is that rays were sent between nodes, similar to Reinhard et al. [234], to compute reflections and shadows. The

static image decomposition scheduler worked similarly to the image-parallel algorithms discussed previously. Abram et al. [1] extended the domain decomposition model to an asynchronous, frameless renderer using a subtractive lighting model for progressive refinement. Rays were sent between nodes to compute distributed illumination effects and reported partial shading results back to the display to provide progressive refinement. Park et al. [212] extended both the image and domain decompositions, and introduced ray speculation to improve system utilization and overall performance. By moving both rays or data as needed, these approaches were able to compute global illumination effects on distributed data, providing high-quality images at additional cost. Biedert et al. [25] employed a task-based model of distributed rendering that was able to combine sort-first and sort-last rendering. Biedert et al. built their approach on an existing runtime system, HPX [143], to balance between these strategies. Although their work used OSPRay for rendering, the renderer was restricted to a single thread per rank and was noninteractive.

2.3.2 Local Visualization

In this dissertation we consider visualization in VR for the development of immersive visualization applications in connectomics, and in the web browser, to improve the accessibility of large data visualization.

2.3.2.1 Immersive Virtual Reality Environments for Connectomics

Interest in using immersive VR environments for neuron tracing, and visualization in general, to overcome the limitations of traditional 2D desktop interaction and rendering modalities has been growing. In contrast to desktop software, immersive VR systems enable users to visualize and interact with their data directly in 3D, providing a more intuitive interface and allowing for better understanding of 3D structures [157, 158, 226]. VR environments such as CAVEs [59, 60] have been demonstrated to be effective at enhancing visualization tasks related to understanding 3D data. Prabhat et al. [226] performed a comparative study on exploration of confocal microscopy data in desktop, fishtank VR, and CAVE VR environments. They evaluated tasks focused on navigation and observation, e.g., locating and counting features or describing some structure. In their study, Prabhat et al. found that users tended to perform better on these tasks in the CAVE environment,

which provided better immersion. Laha et al. [157,158] examined how VR system fidelity affected the performance of common visualization tasks by varying field of view, head tracking capability, and support for stereo display. The tasks studied were similar to those studied by Prabhat et al. [226], involving search and examination of 3D structures. Laha et al. found that more immersive VR environments improved users' understanding of complex structures in volumes [158] and isosurfaces [157].

The original CAVE used a tracked wand device with three buttons to manipulate objects [60]. CAVE2 [79] employed a similar wand controller, using a modified PS3 Move controller. CAVE2 also supported a prototype-tracked sphere controller, the CAVESphere, for moving and interacting with the data, along with a tablet controller showing a web view. Although CAVEs are able to provide high-quality VR, they are large and expensive systems, both to purchase and to maintain, making their incorporation into the routine workflow of scientists in small laboratories unlikely. Direct 3D interaction with the data can also be challenging in a CAVE, as the user's hands will block the displays as a selection is made, thereby occluding the objects of interest. Although well suited for virtual tours of complex data sets with application-centric software (see, for example, [232,287]), using CAVEs for day-to-day tasks involving frequent manipulation of data can be costly and challenging.

Due to the difficulty of providing input feedback when working with free-form 3D controllers, prior work has evaluated the use of haptics to provide better feedback to the user. Ikits and Brederson designed the visual haptic workbench [130], which combined a stereo display table with a probe arm. The arm was used to interact with the data and provide haptic feedback. For example, when tracing a streamline, the probe would be constrained to follow it, providing the feeling of interacting with a physical object. Palmerius et al. [210] described a system of haptic feedback primitives for computing haptics on volumetric data, e.g., for directional or vibration feedback, which can provide a greater sense of touching structures in the environment.

VR systems have also been proposed for visualization of electron and wide-field microscopy data. Agus et al. [4] proposed a model for simulating lactate absorption to compute lactate absorption maps on 3D segmented electron microscopy data sets. The absorption maps could then be visualized in a CAVE or head-mounted VR environment, by rendering the segmented neuron meshes colored by absorption rate. Boges et al. [30,31]

proposed a VR tool for creating, editing, and visualizing skeletonizations of brain cells in electron microscopy data, along with their segmented surface mesh. The neuron skeleton was created by the user with the assistance of a guidance system that used the segmentation to move points placed inside the mesh to the mesh's center. Boorboor et al. [32] proposed a data processing and feature extraction pipeline, the output of which was visualized using an immersive display wall visualization system implemented with Unity. Sicat et al. [256] presented DXR, a Unity-based toolkit to ease development effort for immersive visualization applications. Fulmer et al. [89] presented a web-based immersive neuron visualization system using Unity for exploration of online databases of neuron data using a Hololens.

2.3.2.2 Scientific Visualization in the Browser

Information visualization applications using D3 [33], Vega [245], Vega-Lite [244], and Tableau (formerly Polaris) [266] have become ubiqituous on the web, reaching millions of users to make data more accessible and understandable. The web browser platform is key to enabling this ubiquity, by providing a standardized, cross-platform environment through which applications can be deployed to users. Similarly, the browser eases the process by which users access an application and enhances security, removing the need to install and trust additional packages on their system. Although the browser has been widely used to deploy information visualization applications, this has not been the case for scientific visualization applications.

Scientific visualization applications often rely on native libraries (e.g., VTK [249]), fast floating point operations, large amounts of memory, and parallel processing or high-performance rendering on the GPU or CPU. Porting the required native libraries and rewriting the application to JavaScript is a significant undertaking, and does not address the computational demands of the application. As a result, the majority of web-based scientific visualization applications rely on a backend server to perform data processing and visualization (e.g., [71, 140, 221, 228, 229]), or to stream reduced representations of the data to reduce the demands placed on the browser (e.g., [134, 192, 251, 255]). Using a server gives the application access to significant computational resources, at the cost of latency for users and financial cost to the developer to run the servers. When considering a large-scale

deployment, for example, data visualization on the home page of the *New York Times*, the cost of running sufficient servers to satisfy the large number of clients is a real concern. Fully client-side applications move the computation and data to the client, where users can interact with the data locally without added network latency or expensive backend compute servers; however, client-side applications have access to much less compute and memory capabilities.

Even with the browser's prior limitations on graphics and compute capability, a number of works have explored methods for deploying scientific visualization in the browser, either through a client-server architecture [71, 140, 221, 228, 229] or by leveraging the capabilities provided by WebGL [134, 192, 251, 255]. Mobeen and Feng [192] demonstrated a volume renderer in WebGL that was capable of high-quality interactive volume rendering of small data sets on mobile devices. Sherif et al. [255] presented brain browser, a browser-based tool that enabled interactive visualization of large-scale neuroimaging data sets fetched from a remote server. Potree, proposed by Schütz [251], enabled interactive rendering of large LIDAR data sets in the browser through a rendering-focused octree data layout that was streamed on demand from a remote server. Li and Ma [169, 170] proposed an approach to perform parallel processing in the browser that leveraged the WebGL rendering pipeline to implement a subset of data-parallel primitives. However, WebGL lacks support for compute shaders and storage buffers, making the implementation of general high-performance parallel primitives challenging.

To visualize large data sets, prior work has either streamed subsets of the data to clients for rendering [251,255], or rendered the data remotely on a server and streamed the resulting images [71, 140, 221, 228, 229]. Data streaming approaches are typically built on top of a rendering-focused data layout that enables spatial filtering and multiresolution reads to efficiently send reduced data to the clients from a server. Potree [251] leveraged a rendering-focused multiresolution octree data layout on the server to send LOD representations of large LIDAR data sets to clients that render the data using WebGL. Remote rendering approaches allow the application to access arbitrary compute capabilities to process massive data sets, although they can introduce challenges with latency and cost.

In a recent survey of web-based visualization, Mwalongo et al. [199] specifically mentioned latency and the lack of CUDA for GPU computing as key remaining challenges to

deploying scientific visualization in the browser. By leveraging WebGPU, applications that rely on GPU computing can now be run locally on the client in the browser, eliminating the cost of GPU virtual machines and latency. Access to powerful GPU capabilities can also ease the implementation of more advanced latency-hiding techniques for hybrid client-server applications.

2.3.2.3 Parallel Isosurface Extraction

Isosurface extraction is a classic and widely used scientific visualization algorithm, which has received wide attention since the original marching cubes paper [179]. Livnat et al. [175] and Cignoni et al. [51] proposed to accelerate isosurface computation by skipping subregions that were known to not contain the isosurface based on the range of values in the subregion, using a range k-d tree or interval tree to accelerate the search. Approaches leveraging similar ideas for skipping regions, based on macrocell grids [213] and k-d trees [99, 290], have been proposed for implicit isosurface ray tracing, allowing the application to render the isosurface directly without storing additional vertex data.

A number of GPU parallel isosurface extraction algorithms have been proposed [52, 61, 70, 136, 172, 187, 248, 250] that accelerate computation by parallelizing the work over the many cores of the GPU. GPU parallel algorithms typically process each voxel in parallel in a thread, using data-parallel prefix sums [28] and stream compaction operations to output the computed vertices into a single buffer. Early work achieved the compaction step using HistoPyramids [70] and performed processing in the geometry shader; however, recent works [61, 172] in CUDA have been able to leverage compute kernels for processing and Thrust [15] for fast prefix sum and stream compaction kernels. However, existing algorithms have been built on the assumption that the entire data set is able to fit in the memory of a single GPU [52, 61, 70, 136, 172, 187, 248, 250], or be distributed among multiple GPUs on a cluster [187], preventing their use on massive data sets on single workstations or the web browser.

The block-based work decomposition proposed by Liu et al. [172] for isosurface extraction on the GPU is similar to the algorithm proposed in this dissertation for interactive isosurface extraction in the browser. Liu et al. computed the value range of each block and used this information to filter out blocks that did not contain the isovalue. They then

computed the number of vertices to be output by each block in parallel and performed a prefix sum to compute the output offsets for each block. A second pass computed and output the vertices for each voxel. Per voxel output offsets were computed using a prefix sum within the thread group processing a block, eliminating the need to store per voxel offsets or perform a global prefix sum. Within each block, vertices are computed using Flying Edges [250]. However, Liu et al. assumed that the entire data set is stored in a single 3D texture and only used the block decomposition to accelerate computation. In contrast, the approach presented in this dissertation does not store the full volume data. Instead, the blocks required to compute the isosurface are decompressed and cached as needed using a GPU-driven LRU cache.

2.4 In Situ Visualization

Early work on in situ visualization began over 25 years ago [108, 182, 214], although it was referred to as "co-processing" [108], "computational steering" [214], or "runtime visualization" [182]. In recent years, the increasing gap between FLOPs and I/O has motivated research and adoption of in situ visualization in the simulation and visualization communities [11]. A wide range of in situ visualization approaches have been proposed, from simulation-tailored methods [17,103,235,306,307,319] to general purpose frameworks, such as ParaView Catalyst [12,78], VisIt LibSim [47,302], ADIOS [177], and GLEAN [284,285]. These works have explored a number of strategies to integrate the visualization into the simulation, what manner of visualizations to produce, and what support for human interaction should be provided, if any. Childs et al. [45] presented a unified terminology to describe in situ visualization, to address the growing and conflicting terminology used to describe various methods. This section first summarizes the terminology presented by Childs et al. [45] (Section 2.4.1), which is then used to describe in situ work throughout the dissertation and discuss illustrative examples from prior work. Based on this terminology, I adopt three coarser terms that can be used to describe key aspects of how the visualization interacts with the simulation: tightly coupled, data staging, and loosely coupled. Section 2.4.2 discusses recent work on Tightly Coupled in situ methods, Section 2.4.3 discusses related work on Data Staging in situ, and Section 2.4.4 discusses work on low overhead Loosely Coupled in situ strategies.

2.4.1 The Terminology of In Situ Visualization

The In Situ Terminology Project [45] was a collective effort led by Hank Childs of over 50 visualization experts to develop a consistent and unified terminology for in situ visualization. The terminology project defined six axes along which an in situ visualization method can be categorized. The axes described different aspects of how a technique is integrated into the simulation, how it is operated, and what manner of outputs it produces.

2.4.1.1 Integration with the Simulation

The major axes along which in situ visualizations have been categorized in previous work describe how the visualization is integrated into the simulation. Integration type describes how the visualization methods are called by the simulation or otherwise integrated into it. Proximity describes how close the visualization tasks are to the simulation data. Access describes how the simulation makes data available to the visualization. Division of execution describes how compute resources are shared by the visualization and simulation.

2.4.1.1.1 Integration type. The integration type axis is split into two main categories: application aware and application unaware, each with a set of more specific subcategories. Application aware approaches consist of typical methods for integrating in situ visualization. An application aware integration can be bespoke, where a custom visualization method tailored for the specific simulation is used [272, 306, 307, 319]; or through a dedicated application programming interface (API) expressly for in situ visualization, such as Catalyst [78], LibSim [302], or SENSEI [10]; or through a multipurpose API, where the visualization is integrated into a data management library already used by the simulation, for example, ADIOS [177] or GLEAN [284]. Bespoke integrations require little integration effort on the part of the simulation, but can require significant effort from the visualization effort as each new visualization method must be custom tailored to the simulation. Bespoke integrations can potentially provide better performance, as the visualization is tailored specifically to the simulation's data structures. Dedicated APIs require that the simulation map its data to a visualization layout used by the API (e.g., VTK). However, once this is done the simulation has access to the wide range of visualization methods provided by the APIs. To reduce the mapping cost, VTK provides zero-copy strided arrays to support a wide range of simulation data layouts. Multipurpose API integrations require no integration effort for simulations

already using the repurposed data management API, and can provide additional flexibility as to the proximity of the visualization.

Application unaware approaches integrate the visualization without modifying the simulation's code or that of any libraries it uses. An application unaware integration can be done through interposition, where a shared library used by the simulation for data management is replaced at runtime by another containing the visualization code [84], or through inspection, where the same facilities used to implement debuggers are leveraged to inspect the simulation memory and insert the visualization [83].

2.4.1.1.2 Proximity. The proximity axis describes how close the visualization is to the simulation data, and is directly tied to how expensive it is for the visualization to gain access to the simulation data. Proximity is best seen as a spectrum, where data access costs increase as the visualization gets further from the simulation data. The closest configuration is a same-process proximity, achieved by integrating the visualization directly into the simulation code. A desirable benefit of same-process proximity is that the simulation and visualization can operate directly on the same data. Same-process proximity is the proximity employed in standard dedicated APIs [78, 302]. A slightly further configuration consists of running the visualization in a separate process on the same nodes as the simulation, and accessing the data by copying it over a shared memory mechanism [325]. To reduce the impact of the visualization on the simulation, the visualization can be moved further away and run on a separate set of nodes on the same HPC cluster [17], or even to an entirely different HPC cluster [74]. In these more distant configurations, the data must be transferred over the network from the simulation to the visualization, increasing transfer costs.

2.4.1.1.3 Access. The access axis describes how the simulation passes data to the visualization, and is split into two main categories: direct and indirect. Direct access is made when the simulation and visualization operate in the same logical memory space and can share pointers with each other. Direct access can be further divided into shallow copy or deep copy access, depending on whether the visualization just shares the simulation's pointers, or makes a separate copy of the data to work on. Indirect access is made when the simulation and visualization do not operate in the same logical memory space, and pass data by sending them over the network.

Access is closely related to proximity, a desirable aspect of running the visualization

in the same process as the simulation is that the data can be shared directly using direct shallow copy access to avoid data copies. However, it is possible to combine off node proximity with direct access through remote direct memory access via MPI or partitioned global address spaces. Similarly, indirect access can be used with on node proximity to decouple the simulation and visualization tasks.

2.4.1.1.4 Division of execution. The division of execution axis describes how compute resources are shared between the simulation and visualization using two categories: time and space. Time division runs the simulation and visualization tasks on the same compute resources, where they must take turns executing. Time division is commonly employed in same-process proximity configurations, after the time step is completed the simulation calls into the visualization code, and does not continue until the visualization completes. Space division divides the compute resources between the simulation and visualization, dedicating a set of resources to each. Space division allows the visualization to run in parallel to the simulation, for example, on a separate thread, process [235, 325], or node [74]. Space division can also be used to reduce the impact the visualization has on the simulation, by allowing the simulation to begin computing the next time step while the visualization continues processing, or to move the visualization tasks to an entirely different node to reduce memory pressure. Time division is typically employed by in situ applications using Catalyst [78] or LibSim [302], as it does not require synchronization or incur possible memory hazards between parallel tasks.

2.4.1.2 Operation Controls

The operation controls axis describes how the in situ visualization tasks can be controlled or modified while the simulation is running. The axis contains two categories: automatic and human in the loop, each with two subcategories. Automatic controls can be either nonadaptive, where the visualization configuration is fixed when the simulation is started and cannot be changed, or adaptive, where the visualization can be adjusted using, for example, in situ triggers [161]. Automatic controls are well suited to the batch execution mode employed on HPC clusters, and are typically the default mode for in situ visualization tasks. Human in the loop controls allow the scientist to connect an application to the running simulation to view or modify the visualization configuration. These controls can be

blocking, where the simulation must wait until the application is disconnected to continue, or nonblocking, where the visualization runs in parallel to the simulation. Blocking methods are typically employed in visualization configurations using same-process proximity, direct access, and space division. For example, although they are typically used in an automatic nonadaptive mode, both ParaView Catalyst [78] and VisIt LibSim [302] support attaching a local client to the running simulation to inspect or modify the visualization. Human in the loop systems can also be used to interactively adjust the simulation parameters to support computational steering, as done in SCIRun [214].

2.4.1.3 Output Type

The output type axis describes the types of visualization products output by the in situ visualization for use post hoc. The output type axis is defined by three categories of possible outputs, subset, transform, or derived. Subset outputs save out some reduced form of the raw data, by selecting a subregion of interest or performing a subsampling of the data [306]. Transform outputs apply some form of transformation to the data before outputting it, although they do not necessarily reduce the data in size. Derived outputs create and output different representations of the raw simulation data, for example, images, statistics, or particle traces. Derived outputs can be further divided into subcategories based on whether the output size is fixed or proportional relative to the input. For example, image databases [7, 142] are fixed in size relative to the input data size, as the desired visualization parameters are set up front by the scientist. The size of a proportional output is tied to the size or complexity of the input data, for example, isosurfaces or Merge Trees [17].

2.4.2 Tightly Coupled In Situ Visualization

Tightly coupled in situ visualization is characterized by same-process on-node proximity, direct access and a time division of execution. Tightly coupled in situ allows the visualization to directly share memory with the simulation to eliminate data copy costs, and has been widely used in practice. From a practical standpoint, tightly coupled in situ visualization is well suited to the batch execution model of current HPC systems, as the visualization is performed as an additional fixed task within the simulation code. Tightly coupled in situ visualization has been demonstrated as an effective approach for a range of in situ visualization applications [7, 23, 77, 80, 116, 129, 142, 159–161, 182, 214, 218, 219, 223, 271,

272, 298, 306, 307, 316, 317, 319, 321], and is the in situ modality provided by the standard dedicated in situ APIs Catalyst [78] and LibSim [302]. The SENSEI [10] project was started to address the portability and reusability challenges introduced by this growth of in situ frameworks and visualization methods. SENSEI supports a range of execution modes depending on the backend selected, although it is frequently used in a tightly coupled mode.

Early work by Parker and Johnson [214] leveraged tightly coupled in situ visualization to develop a fully integrated computational steering and visualization system, SCIRun. SCIRun employed a dataflow based computational model and in situ visualization to provide an intuitive and interactive interface that scientists could use to structure their computation, inspect its progress, visualize results, and adjust the computational parameters as needed. Ma [182] integrated a volume renderer into a computational fluid dynamics (CFD) simulation to provide real-time simulation monitoring. The rendered images were either saved to disk or sent to a remote client for display. However, the remote client did not provide interactive controls to modify the visualization parameters.

Tu et al. [272] proposed a fully integrated meshing to visualization pipeline for large finite element method (FEM) simulations. Tu et al. integrated the various stages of an FEM simulation visualization pipeline (meshing, partitioning, simulation, and visualization), into a single application, thereby eliminating the I/O bottlenecks that would otherwise have occurred between these stages to improve overall performance. Yu et al. [319] demonstrated the effectiveness of tightly coupled in situ visualization for rendering large combustion data sets from S3D [114]. Yu et al. integrated a data-parallel renderer based on a 2-3 swap [320] compositor into S3D. Yu et al. evaluated their approach on up to 15,360 cores and demonstrated that in situ rendering could be up to hundreds of times faster than saving the data for post hoc visualization. Woodring et al. [306] presented an in situ data reduction strategy for large cosmology simulations, based on constructing a multiresolution k-d tree over the particle data. Woodring et al. built a k-d tree over the particles, and duplicated a subsampling of the particles up the tree to provide LOD. To reduce the amount of data saved, their approach discarded levels of the tree, using the coarser levels to provide a representative subset of the data for post hoc visualization. Woodring et al. [307] proposed an in situ eddy tracking approach that enabled high spatial and temporal

resolution tracking of eddies in large ocean simulations. Their approach detected eddies by computing and thresholding the Okubo-Weiss field on each rank, after which eddies were found by computing the connected components of the Okubo-Weiss field. Finally, a global list of eddies was constructed and output for post hoc visualization by aggregating the partial eddy statistics from each rank. Thompson et al. [271] presented an in situ framework based on VTK-m [196] and SENSEI [10] to leverage the parallel computing capabilities available on CPUs and GPUs. Larsen et al. presented Strawman [160] and its successor ALPINE [159]. Both frameworks were designed to explore options for lightweight and low-overhead tightly coupled in situ integrations that were capable of leveraging parallel processing on CPUs and GPUs. Larsen et al. adopted a flexible JSON-based data description model for low-overhead data description and VTK-m [196] for parallel computation on CPUs and GPUs. Ibrahim et al. [129] presented an approach that leveraged Jupyter notebooks for interactive in situ visualization through ALPINE's Ascent run time.

A number of simulations have begun moving to an asynchronous many task (AMT) runtime model for portability and performance [16]. This migration can pose a challenge for tightly coupled in situ visualization methods that do not use the same runtime system, as it can lead to scheduling or data management issues. Pébaÿ et al. [219] performed an initial assessment of using Legion [13] as a runtime system for in situ computation of statistics within a Legion-based simulation mini-app. Pébaÿ et al. [218] continued this effort further and developed a SPMD model for computing statistics within the Legion runtime. Pébaÿ et al. integrated their approach into a full-scale simulation, S3D [114]. Heirich et al. [116] presented an approach to task-based in situ visualization using Legion [13] that allowed the visualization to use the same runtime system as the simulation, for Legion-based simulations. Petruzza et al. [223] proposed an embedded domain-specific language for in situ visualziation, BabelFlow, that could integrate with different AMT runtimes for portability.

A key limitation of in situ visualization is the loss of interactivity when viewing the outputs post hoc. The visualization parameters used to produce the images are fixed when the simulation is launched, and cannot be adjusted afterwards when viewing, e.g., the output images. To recapture some of the interactive and exploratory capabilities provided by post hoc visualization, work on in situ visualization has explored the creation of "explorable

extracts" [7, 23, 80, 142, 298, 316, 317, 321]. Such approaches output a set of images or other proxy representations of the data that can be used to provide various levels of interactive manipulation post hoc. Although such approaches are not specifically tied to a tightly coupled configuration, they are typically employed in such a configuration. Yucong et al. [321] proposed an approach that created layered images using a data-parallel renderer, which could then be recombined interactively post hoc. Their approach took advantage of the distributed nature of data-parallel rendering by saving the partial images rendered on each rank containing color, depth, normals, and other information instead of compositing them to a single final image. The partial images could then be reconstructed post hoc to perform spatial clipping, transfer-function-based culling, and modify the camera or light position. Kageyama et al. [142] and Ahrens et al. [7] proposed strategies based on rendering a sweep of the desired visualization parameter space. The parameter sweep could cover different isovalues, transfer functions, camera positions, and so on, to output a database of images. The images could then be blended and combined post hoc in a viewer to interactively adjust the various visualization parameters. Works by Fernandes et al. [80], Biedert and Garth [23], Ye et al. [317], and Wang et al. [298] have leveraged various forms of layered depth images to output explorable extracts. Rather than rendering a single image containing just the objects nearest to the camera, multiple images are created, containing objects successively further from the camera [23, 317]. In the case of volume rendering, Fernandes et al. [80, 298] proposed to split the ray into segments to produce a layered volume image. The layered depth images could then be explored interactively post hoc by combining the multiple layers to create new views or filter out objects.

2.4.3 Data Staging In Situ

Data staging in situ is typically performed during the simulation's I/O pipeline, and is characterized by its use of a multipurpose API, a same-process subset-of-ranks proximity, indirect access, and time division of execution. From the simulation's perspective, I/O and data transfer for in situ visualization share many similarities, making integrating in situ visualization through the I/O library a valuable approach to integrate it without modifying the simulation code. Although it is also possible to implement a tightly coupled system through a multipurpose API by running the visualization immediately after the data are

passed to the I/O library on each rank, data staging in situ first aggregates the data to a subset of ranks that then run the visualization. By decoupling the level of parallelism and data distribution of the simulation and visualization, data staging can provide benefits over a tightly coupled approach. For example, the visualization may need access to more data than are available locally on each rank. By aggregating the data to form larger subregions on fewer ranks for processing, the total amount of network traffic required can be reduced. Similarly, the visualization code may not scale as well as the simulation, and running it on a subset of ranks can actually improve overall performance [17].

Data staging in situ has been integrated into ADIOS [94, 177] and GLEAN [284], by running the visualization on the aggregator ranks during the two-phase I/O pipeline. Data staging in situ integrations within an I/O library is typically done by integrating a distributed data management or query strategy [63, 68, 326, 327] to support visualization tasks that require data from other ranks. These distributed data management libraries work by leveraging the data and metadata provided to the I/O library to aggregate and restructure the data to match the parallelism of the visualization process or to implement an on-demand distributed data query mechanism.

DataSpaces [68] is a generic distributed data sharing library that has been used to enable data staging in situ, among other methods. DataSpaces provides a shared virtual memory space abstraction over the distributed simulation data. The distributed data are indexed using a distributed hash table, which a query engine uses to answer spatial subregion queries from clients. Client queries can request large regions to redistribute the data on to fewer ranks, or small ghost regions as needed for processing. Bennett et al. [17] demonstrated a combined tightly coupled and data staging in situ pipeline for in situ Merge Tree computation in S3D [114] that used DataSpaces to stage data for the Merge Tree computation. Zheng et al. [326] presented PreData, a data staging and aggregation middleware within ADIOS for in transit visualization, that leveraged ADIOS's I/O metadata to describe the data. PreData uses a map-reduce stream processing pipeline on the staging nodes to reduce memory overhead. PreData also integrates DataSpaces to provide distributed data access for visualization tasks that required neighbor data. FlexIO [327] repurposed the ADIOS file I/O API to enable data staging in situ, and to allow flexible placement of the in situ visualization tasks. Simulations using FlexIO "write"

a stream file that is "read" by the visualization tasks. Reads can specify a different data distribution among the subset of ranks running the visualization task to restructure the data using FlexIO.

2.4.4 Loosely Coupled In Situ

Loosely coupled in situ is characterized by different process proximity, indirect access, and a space division of execution. Loosely coupled approaches can run the simulation and visualization on distinct nodes, reducing the impact of the visualization on the simulation [74,148,325]. This separation is highly desirable when scalability or resource contention is of concern, as is often the case in large-scale simulations. Furthermore, the visualization component is free to run at a different level of parallelism than the simulation, or in parallel to the simulation to perform additional computation [74,195,323,325] or enable interactive visualization [235]. The visualization can also be run on demand, starting and stopping as desired in a separate process or job, based on, e.g., simulation triggers [161].

Early work by Haimes [108] presented pV3, which enabled interactive loosely coupled visualization of large-scale CFD simulations. Haimes specifically selected loosely coupled visualization, referred to as "co-processing," to enable interactive visualization, asynchronous execution of the simulation and visualization, and on-demand execution of the visualization. pV3 could be run as desired by the scientist to connect and disconnect from the simulation. When connected, new time steps were queried from the simulation and rendered interactively on a distinct set of nodes while the simulation progressed. Ellsworth et al. [74] presented a loosely coupled in situ visualization approach to render movies of weather simulations in a sensitive environment for hurricane forecasting. Due to the time sensitive nature of the application, it was critical to avoid slowing down the simulation or introducing any chance of it crashing due to the visualization. Thus, Ellsworth et al. adopted a loosely coupled approach and ran the visualization on a separate cluster to render and distribute the videos. Zhang et al. [325] described a loosely coupled approach for in situ feature tracking that ran the feature tracking tasks in parallel to the simulation on the same nodes. Running the two processes on the same nodes allowed them to reduce data transfer costs by directly sharing memory with the simulation, whereas keeping the processes separate still reduced the computation's impact on the simulation and allowed

it to be performed in parallel while the simulation continued on to the next time step. Rizzi et al. [235] coupled LAMMPS [225] to vl3 to enable interactive in situ visualization of molecular simulations. The time division of execution employed in loosely coupled visualization allowed the scientist to interactively explore the data set while the simulation progressed. Zanúz et al. [323] proposed a stream-processing-based approach that leveraged tools from the big data community to construct the pipeline. Their approach built on ZeroMQ for communication, Apache Flink for stream processing, and Apache HBASE for storage. Loring et al. [180] presented an approach to reduce the amount of data that must be transferred for loosely coupled in situ visualization by taking into account metadata from the simulation. Their approach leveraged information about the value range of data on each rank or its spatial position to skip sending data not needed for the visualization being performed. For example, when computing an isosurface, their approach could skip transferring simulation data that did not contain the isovalue.

Loosely coupled in situ visualization can also be realized using a multipurpose API to ease the process of moving post hoc visualization tasks to run in situ [63,327]. Rather than merging the tasks into the simulation, which could require extensive code modifications to both or degrade simulation performance due to scaling or memory use issues, the visualization tasks can be run in a separate process and open a "file" to establish a connection with the running simulation. Reads performed on this file are then mapped to data requests over the network from the running simulation. FlexIO [327] was designed to transparently provide flexibility as to where the visualization tasks were run, and can also be used to enable loosely coupled visualization through the same stream file approach discussed above for data staging. Rather than run the visualization tasks on a subset of the simulation ranks, they are run in a separate process. The information in the stream file is then used to establish network connections between the two processes to transfer data. Dayal et al. [63] adopted a similar strategy in FlexPath, and repurposed the ADIOS file I/O API for both the simulation and visualization to establish network connections between the two. FlexPath used a publisher/subscriber model that allowed visualization tasks to subscribe to data updates from the simulation or other visualization tasks to build loosely coupled visualization pipelines.

CHAPTER 3

ADAPTIVE I/O AND DATA LAYOUT

Particles are widely used in the simulation community to model dynamic and nonuniformly distributed media (see, for example, [21, 105, 225, 259, 260]), due to their ability to handle large dynamic ranges and contact discontinuities often found in cosmology, molecular dynamics, or free-surface and disperse multiphase fluid dynamics. Particle populations in these simulations often span large ranges of space, where localized groups of particles may represent clustered galactic masses, atomic features, or fluid droplets. Although using particles improves the simulation's ability to model these phenomena, the unstructured nature and nonuniform distribution of the particles pose challenges to the I/O system. Post hoc visualization tasks are also made more challenging when forced to work on the raw particle data output by standard I/O strategies, which do not include the spatial or attribute metadata needed to accelerate visualization queries.

This chapter presents work on two-phase I/O techniques for spatially aware adaptive aggregation of particle data [154, 276]. The presented techniques provide different levels of adaptivity to the particle distribution, enabling them to adjust the I/O strategy to balance the workload on different nonuniform particle distributions. This load balancing is performed during data aggregation in a two-phase I/O pipeline. The amount of data aggregated to a single file can also be adjusted by the simulation to tune the I/O strategy to achieve high bandwidth writes across different HPC systems. Finally, to enable particle simulations to output a data layout that is directly usable for visualization, a low-overhead data layout is proposed that can be constructed quickly during the I/O pipeline. The I/O strategies proposed are also capable of high bandwidth reads on the proposed data layout to enable fast checkpoint restart reads or parallel visualization tasks.

Section 3.1 discusses a new adjustable uniform grid (AUG) aggregation strategy. The AUG strategy incurs little computational overhead during aggregation by employing a

uniform aggregation pattern, although it can adjust to a subset of nonuniform particle distributions encountered in simulations. Section 3.2 discusses a new adaptive aggregation strategy using an aggregation tree. The aggregation tree strategy can adjust the aggregation subgroup assignment to adjust to a wide range of nonuniform particle distributions to achieve balanced I/O workloads, at the cost of some additional computation. Section 3.3 discusses a new data layout that allows simulations to output the particle data in a layout that is directly usable for accelerating spatial and attribute queries in post hoc visualization. The proposed data layout is built on a combination of *k*-d trees and bitmap indexing to support such queries with low memory overhead, and can be constructed quickly in distributed parallel environment during I/O. Finally, Section 3.4 demonstrates the scalability and portability of the proposed I/O strategies on uniform and nonuniform particle distributions, and Section 3.5 demonstrates the performance of the proposed data layout on visualization queries.

3.1 Spatially Aware Two-Phase I/O Using an Adjustable Uniform Grid

There are two main challenges when designing a spatially aware I/O strategy: how to create a mapping between the location of a particle in the domain and its location in the file, and how to use this mapping to design an efficient data aggregation phase. For regular grid data there exists a direct one-to-one mapping between a voxel's 3D position in the domain and its location in the file. Such mappings can be produced with various ordering schemes. Common ones used in practice for grid-based data are row-major, Morton order [197] (also referred to as Z-order), and HZ-order [216]. However, the unstructured and unbalanced nature of particle data makes it challenging to construct a similar mapping in general. The aggregation phase is also made more challenging for particle data. In contrast to regular grids, where a fixed number of voxels occupy each spatial region, particles move dynamically through the domain and do not have a fixed spatial distribution that can be determined a priori.

This section presents one approach to tackling these problems, using an adjustable uniform grid (AUG). The approach begins by imposing a 3D rectilinear grid over the simulation domain, referred to as the adjustable uniform grid. The AUG divides the domain into disjoint aggregation partitions where each particle falls into a unique partition.

Each partition is then assigned to a rank that acts as the aggregator for the particles within the partition. Each rank sends its particles to the aggregator rank that owns the partition it falls within, after which the aggregators output a file containing the received data. The AUG can be adjusted to tune the number of ranks that fall in to a single partition, and repositioned or scaled to filter out subregions of the domain that do not contain particles.

3.1.1 Parallel Writes

An overview of the proposed parallel write pipeline on a 2D uniform grid simulation is shown in Fig. 3.1. The simulation partitions the compute workload into patches using a grid, where some number of particles exist in each patch. In the aggregation step, a uniform grid is imposed over this domain to produce four aggregation partitions. One rank is assigned to act as the aggregator for each partition, and receives the particle data that fall within its region. After receiving the particle data, the aggregator can reorganize the particles to construct visualization tailored data layouts if desired. The resulting data layout, or the raw particle data, is then written to disk independently by each aggregator. Finally, a metadata file containing an index of each aggregator's output file and its spatial location is created and output by rank 0. The I/O pipeline is broken down into the following steps, discussed

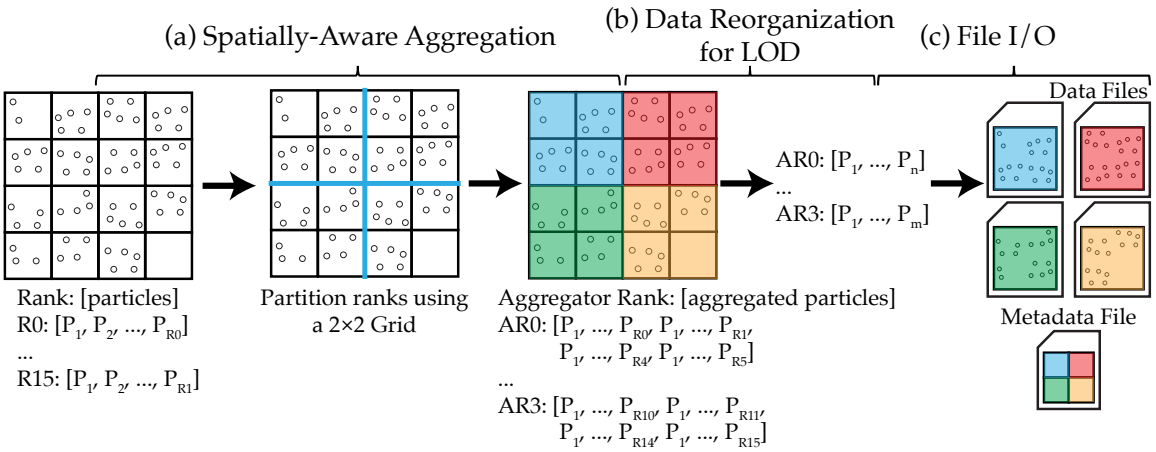


Fig. 3.1: An overview of the adjustable uniform grid two-phase I/O strategy. (a) An adjustable uniform grid is imposed over the simulation domain to partition the domain into the aggregation subgroups. An aggregator rank is assigned to collect the data for each subgroup, and those ranks whose data fall into each partition send their data to the assigned aggregator. After receiving the data, (b) the aggregator reorganizes the data into a layout better suited to progressive level-of-detail reads and (c) writes the data to disk. Finally, a spatial index of the individual files is created on rank 0 and written to disk, containing the bounds of each file and its location on disk.

in the sections below:

- Adjustable uniform grid aggregation (Section 3.1.1.1)
- Data reorganization for LOD reads (Section 3.1.1.2)
- Aggregator data output (Section 3.1.1.2)
- Construction of top-level spatial metadata (Section 3.1.1.3)

3.1.1.1 Adjustable Uniform Grid Aggregation

The first step in the I/O pipeline is to compute and impose the adjustable uniform grid over the simulation domain to create the aggregation partitions. The AUG is constructed based on an aggregation partition factor (APF), (P_x, P_y, P_z). The APF defines the integer size ratio of a cell in the AUG to the simulation patch size in x, y, and z, respectively. For example, in a 2D simulation with a grid of 1×1 patches distributed among $n_x \times n_y$ ranks, each aggregation partition would cover $P_x \times P_y$ ranks (Fig. 3.2). The APF adjusts the number of ranks that are grouped together during aggregation to form each aggregation subgroup, and thus is used to tune the amount of network communication performed and the size of the output files. Given a simulation with grid of $n_x \times n_y \times n_z$ ranks, the total number of files output would be $f = (n_x/P_x) \times (n_y/P_y) \times (n_z/P_z)$. For example, selecting $(1, 1, 1)$ would result in a file per process I/O strategy, whereas selecting (n_x, n_y, n_z) would yield a single-shared file. APFs in between these extremes result in various levels of aggregation and files output. The best partition factor depends on the HPC system's network and filesystem, and the level of parallelism used to run the simulation, and is left exposed as a tuneable parameter for portability.

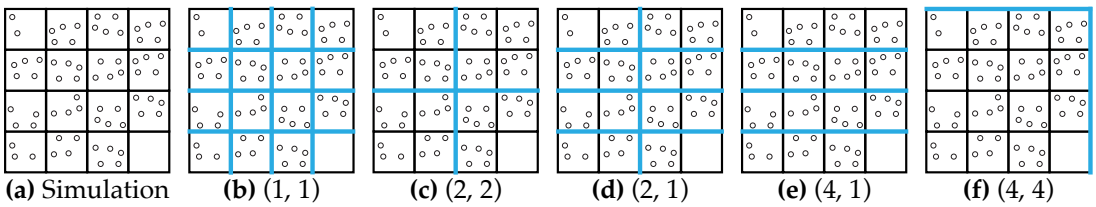


Fig. 3.2: The aggregation partition factor can be tuned to adjust the amount of communication that takes place during aggregation and the number and size of the output files. (a) The simulation is partitioned using different APFs: (b) $(1, 1)$, (c) $(2, 2)$, (d) $(2, 1)$, (e) $(4, 2)$, and (f) $(4, 4)$.

After selecting the APF, the aggregation grid is computed and imposed on the domain. In the case of grid-based simulations, the use of an integer size ratio respective to the simulation's patch size allows the grid to avoid splitting a rank's data between different partitions. Avoiding splitting the data on a rank is highly desirable to improve data transfer performance. If a rank's data are split into multiple partitions, the rank must divide its particles into separate buffers for each partition before the data can be transferred. The cost of this step is linear in the number of particles on the rank as it requires a loop over them to separate them. In the case of non-grid-based simulations, avoiding partitioning the data on a rank between multiple aggregators may not be possible.

Once the aggregation grid is constructed, the next step is to select a rank to act as the aggregator for each partition. Each aggregator rank is responsible for receiving the particle data contained within the assigned aggregation partition. The aggregator ranks are chosen uniformly from the simulation ranks to ensure even utilization of the network. Although this does mean a rank may be assigned as the aggregator for a partition that does not contain its own data, this approach ensures a more even distribution of work over the network [152].

Each aggregation group then transfers its data to the assigned aggregator. However, in contrast to grid based simulations, where the number of voxels on each rank is fixed and known, each aggregator does not know how many particles to expect from each rank in its partition. Thus a metadata exchange must be performed within each aggregation group to compute the number of particles the aggregator will receive from each rank. If the APF is an integer scale of the underlying simulation grid size, each rank will send its data to a single aggregator. Thus the number of particles is just the number of particles on that rank. In the case that a rank's data are split among different aggregation partitions, it must first locally partition the data and send the number of particles in each partition, after which it can transfer the particles. After receiving the number of particles to expect from each rank, the aggregator allocates a buffer large enough to hold the entire set of particles and receives the data from each rank into this buffer. Nonblocking point-to-point MPI communication is used for both the metadata and data exchange.

Although the aggregation partitions are based on a uniform grid, it is possible to adjust the grid somewhat to achieve better load balance on some nonuniform particle distributions

(Fig. 3.3). For example, in material point method simulations in engineering, the particles represent solid objects, and do not scatter around freely in the domain as they do in cosmology or molecular dynamics. As a result, large portions of the domain do not contain any particles. By inspecting the number of particles on each rank before constructing the aggregation grid, the grid can be adjusted to cover just the subdomain occupied by the ranks with particles (Fig. 3.3a). Ranks that fall outside the aggregation grid do not participate in the rest of the I/O pipeline. However, as the partitioning is based on a dense grid, it cannot discard empty ranks (Fig. 3.3b) or even empty partitions (Fig. 3.3c) within the covered domain. Although these empty regions are still placed in an aggregation group, the aggregator can easily determine that no particles are contained within its partition and skip the subsequent aggregation and data output steps.

3.1.1.2 Data Reorganization for LOD Reads and Output

After the particles have been collected on their respective aggregators they can be reorganized to construct a visualization tailored data layout. To reduce the compute requirements of the proposed I/O pipeline and avoid introducing memory overhead, a

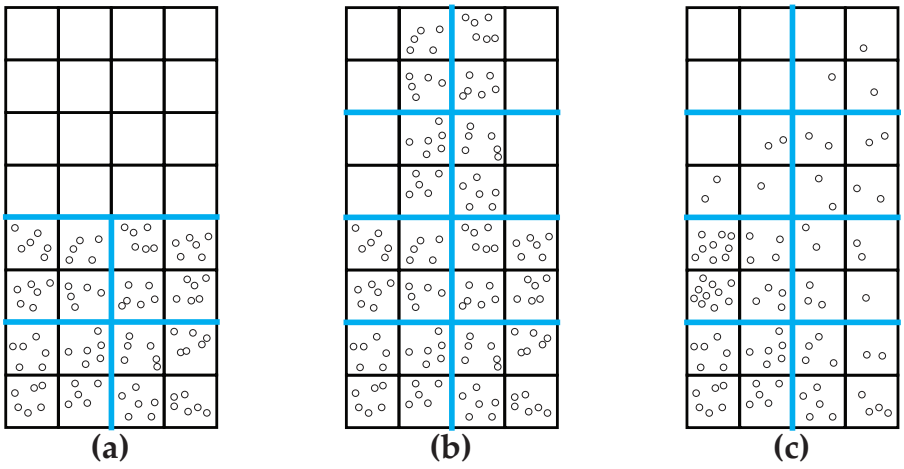


Fig. 3.3: The AUG can be adjusted to adapt to a subset of nonuniform particle distributions encountered in practice. (a) If only a subset of ranks has particles in the domain, the grid can be adjusted to partition only this subset of ranks. (b, c) However, the grid cannot adapt to other distributions of particles that require partitioning the entire domain, even if some ranks do not contain particles or contain fewer particles than others. Although data transfers from ranks without particles are skipped in (b, c), the I/O workload across the aggregators is unbalanced. The empty aggregation group in the top-left of (c) is able to skip the data aggregation step and does not output a file.

minimal layout is built that provides support for only low-latency multiresolution reads. The layout used is similar to that proposed by Rizzi et al. [236], and is constructed by randomly reordering the particles on each aggregator. The particles can then be read incrementally from the start of each file. Each LOD l corresponds to a set of $P \cdot s^l$ particles, where P is the number of particles to treat as the first LOD, and s is a resolution scale factor that can be used to adjust how quickly the number of particles in each level grows (s defaults to 2). As the visualization layout is a simple reordering of the data, it requires no additional memory to store and can be constructed quickly and independently on each aggregator. It is also possible to construct more complex data layouts at this stage if desired, as discussed in Section 3.2. Once the data layout has been constructed, each aggregator outputs its particles to an independent file.

3.1.1.3 Construction of Top-Level Spatial Metadata

To allow the independent files output by each aggregator to be treated as a single file by visualization applications, the final step of the I/O pipeline outputs a spatial metadata file. The metadata file stores a list of bounding boxes corresponding to the bounds of each aggregator's partition and a reference to the file file containing the data for the partition (Fig. 3.4). Rank 0 collects the bounds of each aggregator's partition and writes the spatial metadata file. Post hoc visualization tasks then open this metadata file and use the index to find the other files containing the required data. Visualization tasks can use the spatial metadata to skip reading files that do not contain data required for the task being performed, for example, when processing just a subregion of the domain.

3.1.2 Parallel Reads for Visualization

The presented I/O strategy and data layout provide two key properties that enable scalable reads for visualization queries on smaller clusters and workstations. First, the two-phase I/O strategy outputs far fewer files than typical file per process approaches, and can be configured to output the ideal number of files for the target system. Fewer output files also improve read performance for post hoc visualization tasks, which must access far fewer files to load the data for processing, thereby reducing the overhead that would be incurred by accessing many small files. Second, the spatially coherent data layout output by the I/O pipeline allows visualization tasks that operate on spatial subregions

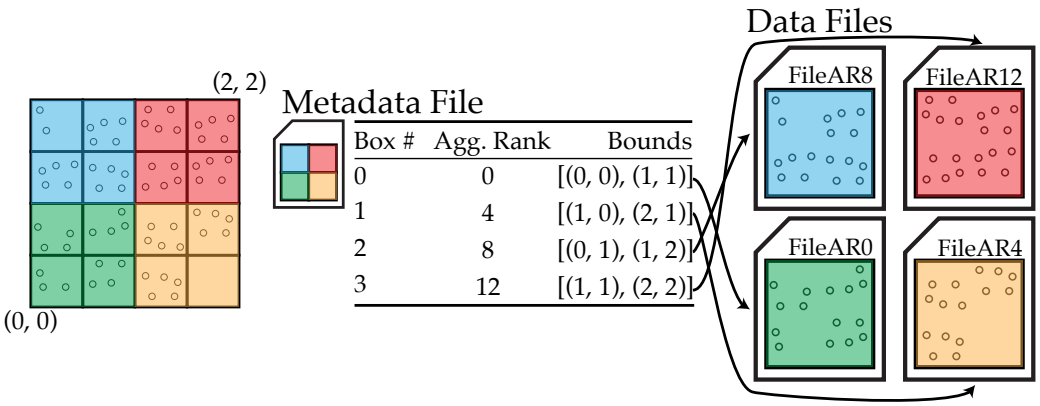


Fig. 3.4: The top-level spatial metadata file output by rank 0 at the end of the I/O pipeline file allows accessing the set of files output as a single file. The metadata provides the bounds of each subregion and a reference to the corresponding output file. The spatial information can be used to accelerate spatial queries by skipping files not contained in the query.

to perform more efficient reads on the data. When accessing a spatially coherent data set, the visualization process can use the spatial metadata to find and access just those files containing the data required for the subregion of interest. Moreover, as the files contain convex subregions, the number of files that must be accessed scales predictably with the size of the subdomain being queried. However, as each rank must independently read the files containing its data, this approach may encounter issues when used for checkpoint restart reads at large levels of parallelism.

3.2 Spatially Aware Two-Phase I/O Using an Aggregation Tree

The key limitation of the AUG strategy is that it cannot provide the adaptivity needed to load balance the variety of nonuniform particle distributions encountered in practice. In simulations where the particles occupy the entire domain, but with varying density, the AUG approach is unable to adapt (Fig. 3.3). Similarly, although the AUG approach can determine that an aggregation group does not contain any particles and skip the aggregation and output steps of the pipeline, the existence of empty or variable size aggregation groups leads to an imbalanced I/O workload and underutilization of the I/O system (see cases Fig. 3.3b,c). Such nonuniform distributions of particles can occur in simulations where the particles move freely throughout the domain, for example, disperse multiphase Lagrangian-Eulerian techniques [91] and partitioned multiphysics methods [324].

This section presents work on a portable technique for scalable, spatially aware adaptive aggregation that is capable of adapting to a wide range of nonuniform particle distributions. The presented technique employs a *k*-d tree [18] partitioning of the domain to produce aggregation groups with similar numbers of particles. Compared to other widely used adaptive spatial data structures, e.g., octrees, BSP-trees, the *k*-d tree provides a good balance between spatial adaptivity and construction cost [241]. The *k*-d tree used to load balance the I/O workload, referred to as the aggregation tree, is built over the ranks' spatial bounds and constructed to produce leaves that contain similar numbers of particles. The ranks within each leaf of the tree form an aggregation group for I/O. The aggregation tree partitioning is able to achieve better I/O load balance across the aggregation groups compared to the AUG approach discussed previously. On nonuniform and time-varying particle distributions, the aggregation tree can achieve up to $2 - 2.5\times$ faster writes and $3\times$ faster reads compared to the AUG (see Section 3.4.2).

3.2.1 Parallel Writes

An overview of the proposed two-phase I/O pipeline is shown in Fig. 3.5. Each write proceeds as follows: All ranks send their particle counts and domain bounds to rank 0, which constructs the aggregation tree (Fig. 3.5a, Section 3.2.1.1). 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

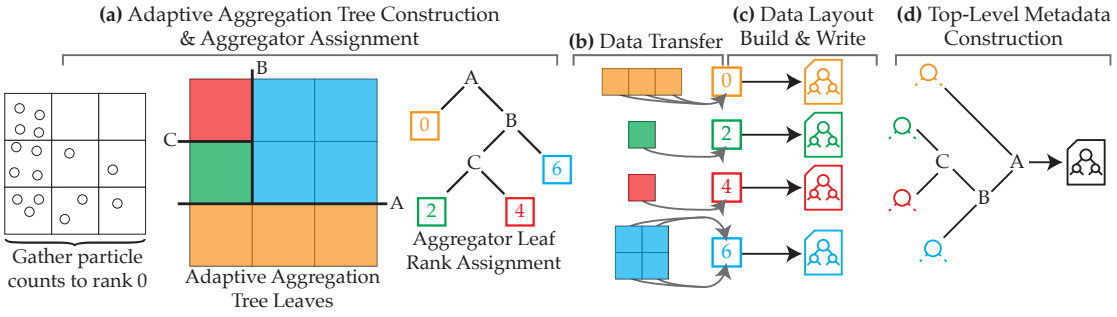


Fig. 3.5: An overview of the adaptive two-phase I/O pipeline using an aggregation tree. (a) Given the number of particles on each rank, rank 0 constructs the aggregation tree to create leaves with similar numbers of particles. Each leaf is assigned to a rank responsible for aggregating the data and writing them to disk. (b) Each rank sends its data to its aggregator. (c) Each aggregator constructs a multiresolution data layout and writes it to disk. (d) The aggregators send their local attribute ranges and their root node bitmaps to rank 0, which populates the aggregation tree with the bitmaps and writes it out.

its local data to the aggregator for the leaf containing it (Fig. 3.5b). After receiving the particles for the leaf, the aggregator constructs and writes out a multiresolution data layout (Fig. 3.5c, see Section 3.2.1.2). Next, a top-level metadata file is populated rank 0 that contains the aggregation tree and references to the leaf files written by the aggregators (Fig. 3.5d, Section 3.2.1.3). The I/O library implementing the proposed approach provides a C API to allow integration into simulations written in a range of programming languages. The API follows an array-based attribute storage model similar to HDF5 [115], ADIOS [177], and Silo [163].

3.2.1.1 Adaptive Aggregation with an Aggregation Tree

The aggregation tree is constructed by first gathering each rank's bounds in the simulation domain and the number of particles it owns on rank 0 (Fig. 3.5a). Rank 0 then performs a modified k-d tree build to construct the aggregation tree. The tree is built so that each leaf of the aggregation tree contains a similar number of particles. As the actual particle distributions within each rank are unknown, the tree build is restricted to select split positions that lie on rank boundaries. Not splitting a rank's data between multiple aggregators also reduces the data processing and transfers that take place during aggregation.

The tree is built with the goal of producing subtrees, and thus leaves, with roughly equal numbers of particles. To do so, a split position must be found at each level that partitions the ranks such that a similar number of particles fall into each subtree. To do so, the builder selects the longest axis of the aggregate bounds of the current set of ranks that have particles and finds 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, the builder determines 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, $c = |0.5 - n_l/(n_l + n_r)|$. Each candidate is tested to find the one with the minimum cost which is selected as the splitting plane. After partitioning the ranks, this process is repeated 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 plane across all spatial axes. In some cases the best split is not along the longest spatial axis, for example, in Fig. 3.5a the split plane B is chosen based on the longest axis, but does not produce the most even partitioning possible.

A leaf node is created when the amount of 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. The improved load balancing achieved by the aggregation tree is illustrated in Fig. 3.6.

To avoid forcing the creation of extremely imbalanced leaves to satisfy the target file

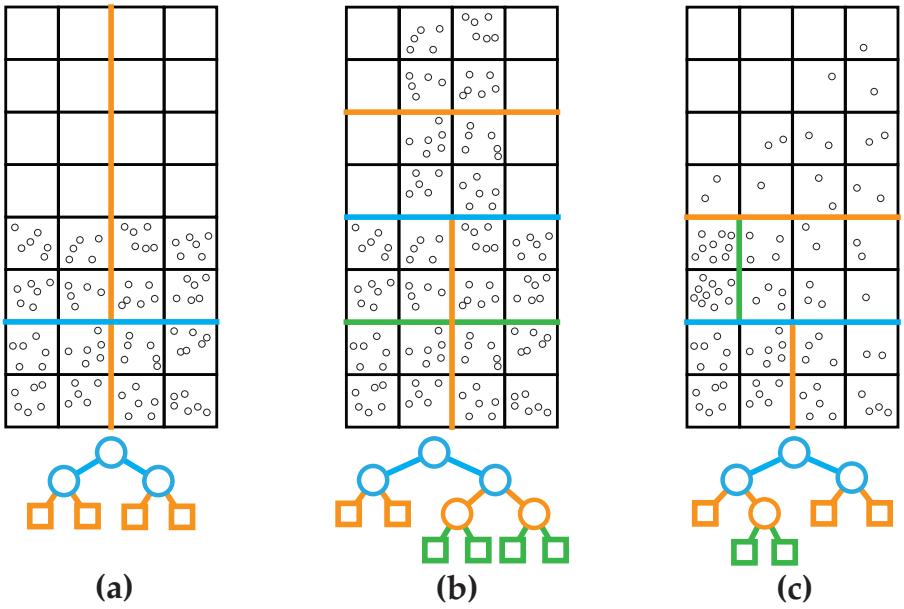


Fig. 3.6: The aggregation tree is able to adapt to nonuniform particle distributions to achieve a balanced I/O workload. Each tree is constructed with a target leaf size of ≈22 particles. The square leaf nodes correspond to the aggregation subgroups. (a) The aggregation tree computes a similar set of aggregation groups as the aggregation grid on uniform subregion cases (compare to Fig. 3.3a). As with the aggregation grid, ranks without particles do not send data during aggregation. Ranks without particles are also not included when selecting splitting positions. (b, c) The aggregation tree build places a similar number of particles in each subtree to produce aggregation groups with similar numbers of particles, and thus a balanced I/O workload. The aggregation grid is unable to achieve a well-balanced I/O workload in these cases (compare to Fig. 3.3b, c).

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 are not partitioned.

To ensure even utilization of the network, leaves are assigned aggregators evenly across the rank space [152]. 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 with ranks in less populated regions would be underutilized.

Rank 0 then 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.

Finally, 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 (Fig. 3.5b). If a rank does not have particles, the data transfer is skipped. Subcommunicators are not created for leaf aggregation groups as 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.

3.2.1.2 Construction of a Multiresolution Data Layout on the Aggregators and Output

After receiving the particles on each aggregator, the multiresolution data layout described in Section 3.3 is constructed on each aggregator (Fig. 3.5c). This data layout, the binned attribute tree (BAT), is then readily available for in transit visualization and analysis executed on the aggregators, providing support for accelerated spatial and attribute queries. Each aggregator then writes its BAT to an independent file.

3.2.1.3 Construction of Top-Level Metadata

The final step in the I/O pipeline is the population of a top-level metadata file on rank 0 (Fig. 3.5d). Each leaf in the tree stores its aggregator rank ID, which is used to locate the corresponding file written by the aggregator containing the leaf data. 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 attribute ranges and bitmap indices of 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 attribute range. The bitmaps of the aggregation tree's inner nodes are computed by merging the bitmaps bottom-up from the leaves.

3.2.2 Parallel Reads

An equally important requirement for a parallel I/O library is that it provide high-bandwidth reads for fast checkpoint restart reads. A two-phase parallel read pipeline that mirrors the parallel write pipeline is used to provide scalable reads on the aggregated data output in Section 3.2.1. The read pipeline proceeds as follows: first, all ranks read the

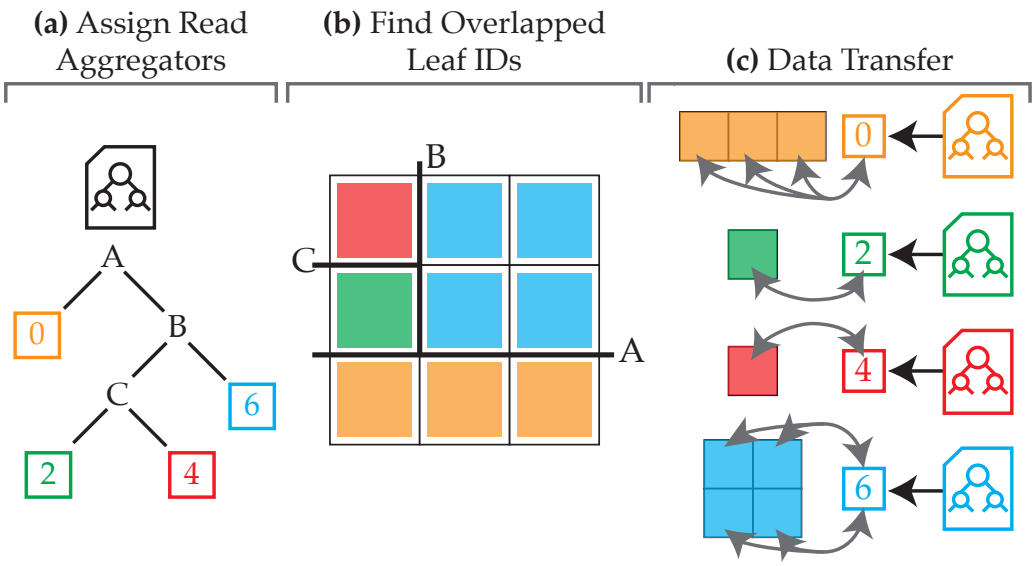


Fig. 3.7: An overview of the parallel read pipeline. (a) All ranks read the aggregation tree metadata, and a subset is selected to act as read aggregators. (b) Each rank determines which leaves it overlaps, and (c) requests data from the aggregator(s) assigned to the leaf file(s).

aggregation tree metadata, and a subset of ranks is assigned to act as read aggregators (Fig. 3.7a, Section 3.2.2.1). Each rank then determines which leaves it overlaps (Fig. 3.7b) and requests data over the network from the read aggregators for these leaves (Fig. 3.7c, Section 3.2.2.2). The read API provided by the library implementing the proposed approach mirrors the write API, with corresponding getters instead of setters to retrieve the data.

3.2.2.1 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 (Fig. 3.7a). Each leaf file is then assigned to a read aggregator responsible for reading the file. The read pipeline is not in control of the number of leaf files, as this is set when writing the data. If there are more ranks than files, read aggregators are assigned as in the write phase, spreading them evenly through the rank space to distribute work over the nodes. If there are fewer ranks than files, the files are assigned evenly among the ranks, with each rank a read aggregator for some set of files. Providing this flexibility in read aggregator assignment allows for 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.

3.2.2.2 Fetching Data from Read Aggregators

Each rank queries the aggregation tree to get back a list of leaf IDs that overlap its bounds (Fig. 3.7b), and uses the read aggregator assignment map to send its bounds to the read aggregator assigned to each leaf. Upon receiving a bounds query, the read aggregator performs a spatial query on its leaf files and returns the particles (Fig. 3.7c).

As was the case for writes, it is not possible to create aggregator subcommunicators to transfer the data during reads. For example, the rank responsible for reading a leaf file may not require data from that leaf, or when reading with fewer ranks than files, each rank may need data from multiple other ranks. Instead, the read pipeline uses a client-server data query system implemented with nonblocking MPI calls. Each read aggregator acts as a data server and watches for incoming queries to process. To receive its data, each rank collects the number of particles that will be returned by each query and 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 completed, 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 visualization.

3.3 The Binned Attribute Tree Particle Data Layout

This section discusses the proposed visualization-focused particle data layout, the binned attribute tree (BAT). The BAT layout builds on ideas from both rendering- and attribute-query focused data layouts, with the aim of providing a layout that can reasonably satisfy the demands of both tasks. Moreover, the layout is specifically designed to be constructed quickly in parallel during the I/O pipeline to be output instead of the raw data, eliminating the need for post hoc data conversion. The binned attribute tree consists of a spatial *k*-d tree with additional metadata stored at each node (Fig. 3.8). The tree nodes are augmented with bitmap indices [309] to accelerate attribute subset queries, and the tree is stored in a memory layout optimized for fast progressive multiresolution reads. To support progressive multiresolution reads, each inner node stores a fixed number of LOD particles to provide a coarse representation of the subtree. The low memory overhead of the *k*-d tree [18, 241], combined with its support for flexible refinement, fast parallel construction [144], and its ability to be augmented with additional metadata for attribute-based filtering and LOD, make it a useful foundation for the data layout when compared to grid-, cluster- or octree-based hierarchies.

3.3.1 Spatial and Attribute Query Filtering

As the BAT layout is fundamentally a spatial *k*-d tree built over the particles, it inherently supports efficient spatial subregion queries. To accelerate attribute subset queries, each node tracks a set of bitmap indices [309] representing the values contained in its subtree and LOD particles (if any). Bitmaps representing user attribute queries are tested against the node bitmaps during traversal to determine if the subtree contains the desired values. In contrast to traditional bitmap indexing approaches [258, 308, 309], where the size of the bitmap can vary as desired at the cost of memory (e.g., to bin real values more accurately), the bitmaps stored in the BAT are restricted to be 32 bits each. Besides this size restriction,

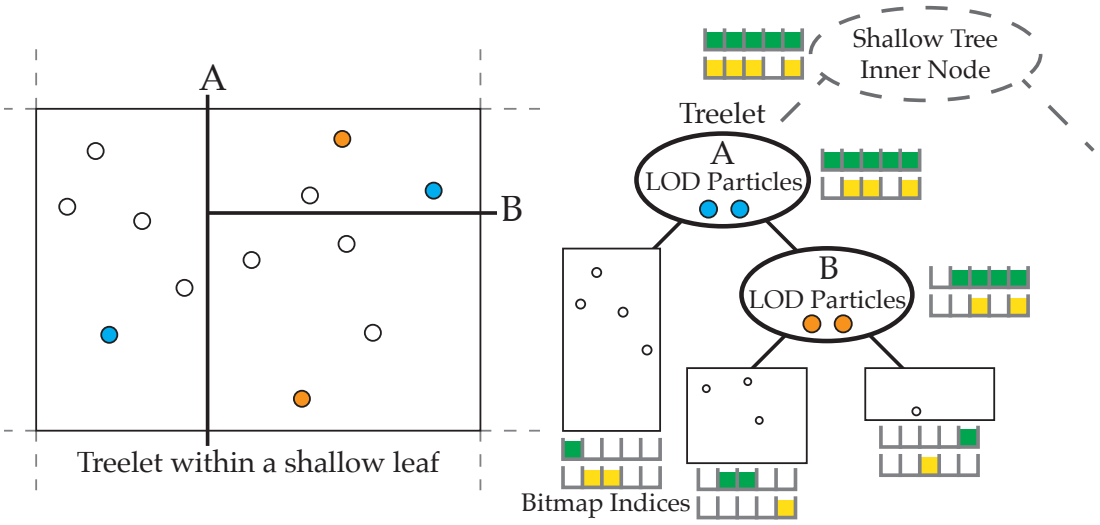


Fig. 3.8: The binned attribute tree layout. Treelets are built within the leaves of a shallow *k*-d tree to parallelize tree construction. Nodes track a bitmap index for each attribute (green and yellow bitmaps) to accelerate attribute queries. Treelet inner nodes store LOD particles sampled during construction for multiresolution reads.

the layout uses a standard binned equality encoding to represent the bitmaps [309]. Integer valued attributes are binned in the same manner. Each bit in a bitmap corresponds to a bin covering 1/32 of the attribute's value range. A bin is set if a particle's value is contained in the bin. Bitmaps can be combined using a bitwise `OR` and tested for overlap by checking if their bitwise `AND` is nonzero, making such operations fast to perform.

Although restricting the size of the bitmaps reduces the resolution of the binning, it provides two key benefits for reducing memory overhead. First, the bitmaps occupy a predictable and small amount of memory compared to unrestricted approaches. The bitmaps computed in unrestricted size approaches can require 30% [268] to 66% [49] of additional storage for the bitmap index; in some cases the index can approach the original data in size [40,48] or exceed it [313]. Second, it is now possible to build a bitmap dictionary over the entire set of bitmaps in the tree, significantly reducing the number of unique bitmaps that must be stored. Although the value ranges of the attributes may differ widely, the corresponding bitmaps are frequently shared. As the BAT is constructed independently on the aggregator ranks during I/O, the bitmaps are computed relative to the local attribute range of particles on the aggregator, instead of the global attribute range. The attributes of interest stored on the particles are often spatially correlated (e.g., pressure, temperature, velocity, strain). As a result, the local attribute range is likely to be a subset of the global

range, allowing for finer bin sizes and in turn better filtering accuracy.

3.3.2 Parallel Construction

The tree will be constructed on a potentially large number of points on each aggregator during I/O, and its performance is critical to the overall I/O pipeline. The tree is built in two parallel steps: a data-parallel bottom-up build that constructs a shallow *k*-d tree (Section 3.3.2.1) and top-down treelet builds within the leaves of the shallow tree (Section 3.3.2.2). These steps are executed in parallel to each other and are parallelized internally. After construction the tree is compacted to build the shared bitmap dictionary and written in a layout optimized for fast multiresolution reads (Section 3.3.2.3).

3.3.2.1 Bottom-Up Shallow Tree Construction

The shallow tree is built bottom-up using Karras's parallel *k*-d tree construction algorithm [144], which builds the entire 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 [197] 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 then 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 resulting tree stores a single particle per-leaf node, which is not suited to the goal of multiresolution reads and low memory overhead.

The shallow tree is built by using a prefix of each particles' Morton code and merging shared prefixes 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 with the same Morton code suffix. A default 12 bit prefix is used, which was found to provide satisfactory results with respect to the number of leaves produced and particles falling into each. For small inputs, the prefix is shortened to coarsen the tree and build fewer treelets with more particles in each one. A treelet is constructed with each leaf of the shallow tree, described in the following section. To support attribute filtering in the shallow tree, the bitmap indices of the inner nodes are populated bottom-up in parallel after the treelets have been built, using the bottom-up traversal described by

Karras. The bitmaps of each shallow leaf are set by the root node of its treelet.

3.3.2.2 Parallel Treelet Construction

The treelet builds for each leaf in the shallow tree are independent and run in parallel. For each leaf, the set of particles with the leaf's prefix are found using a binary search over the sorted set of Morton codes, after which a median split k-d tree is built over them. The median split k-d construction is a standard top-down serial builder that splits the set particles at their median coordinate to form the left and right children. A leaf node is created when the number current particles falls below a configurable size threshold.

A fixed number of LOD particles are stored at each treelet inner node to support multiresolution reads. The LOD particles are selected when creating an inner node by performing a stratified sampling of the current set of particles to select a representative subset. The particles are repeatedly sorted along the splitting axes during construction, allowing a stratified sampling to provide a reasonable spatial distribution of the chosen particles. By taking subsets of particles for LOD, the layout avoids duplicating particles or introducing new representative ones, thereby reducing memory overhead.

The attribute bitmaps are propagated bottom-up during construction. After creating a leaf node, its bitmap indices are computed by taking the bitwise `OR` of the bitmaps of its contained particles. Each inner node computes its bitmap indices by computing the bitwise `OR` of its children's bitmaps with those of its own LOD particles. To reduce memory overhead, bitmaps are stored only for the inner and leaf nodes, not for individual particles.

3.3.2.3 Tree Compaction

After the build steps have completed, the tree is compacted into a single buffer that can be efficiently written to disk. Data within this buffer are structured to support fast reads via memory mapping (Fig. 3.9). During compaction, the bitmaps of each node 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, this has been found to be more than sufficient in practice.

The shallow tree and bitmap dictionary are stored at the start of the file, as they will be accessed frequently during traversal. The shallow tree is stored as a linear k-d tree. The leaf nodes of the shallow tree are replaced with offsets to the corresponding treelets in the file.

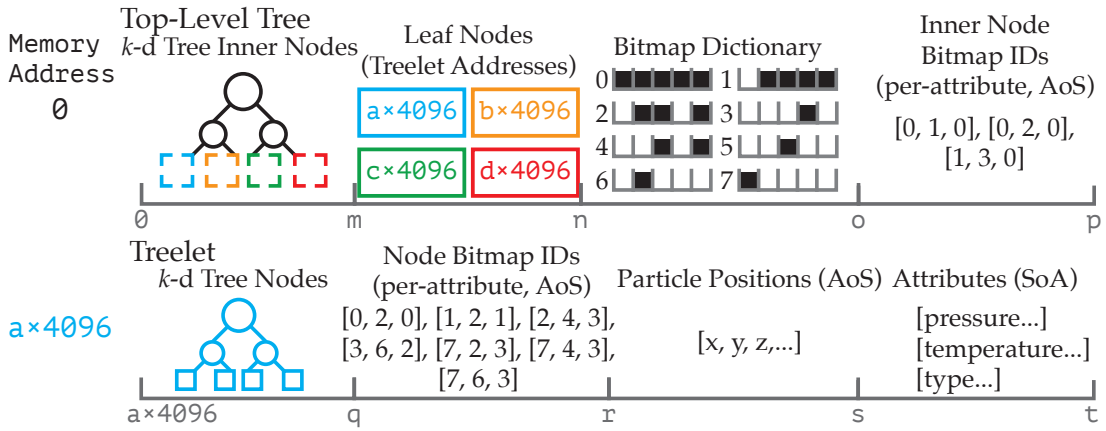


Fig. 3.9: The BAT layout is organized on disk for fast spatial and attribute query traversal and LOD via memory mapping. The top-level tree is stored at the start of the file in two parts. The first section contains the *k*-d tree inner nodes, followed by the leaf nodes. The leaf nodes are addresses 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.

Treelets are stored at 4KB page boundaries to improve read access performance. Each treelet consists of a header specifying the number of nodes and points in the treelet, followed by arrays containing the nodes and particles. Storing the BAT in this separate layout provides a $2-3\times$ improvement in read performance over merging the shallow tree and treelets into a single tree, and removes the slow serial merge step that would otherwise be required to merge the shallow tree and treelets into a single tree.

3.3.3 Support for Visualization Reads

Postprocess visualization tasks are typically performed on laptops, workstations, or small clusters, at far lower levels of parallelism than the simulation was run at, and with access to far less memory and compute power. In the case of large-scale simulations, the user's system may not even have sufficient disk space to store the data, requiring some form of remote streaming or rendering.

Visualization reads on the BAT 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. The BAT layout directly accelerates common visualization and analysis tasks involving spatial and attribute subset queries (Section 3.3.3.1), and

supports progressive low-latency multiresolution reads for data streaming (Section 3.3.3.2). 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).

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

When performing attribute filtering, the user's query bitmaps are tested against those stored at each node during traversal. If the overlap of the two is empty, the subtree can be ignored. The user's query bitmap must be remapped when entering a treelet, as the range of the treelet's bitmaps is a subset of the global range for each attribute. A final false positive check is required before returning queried points to the user's callback. Although the spatial filtering provided by the *k*-d tree is exact, the binned bitmaps only ensure that false negatives are not discarded, requiring a final false positive check for attribute queries.

3.3.3.2 Low-Latency and Progressive Resolution Reads

Low-latency and progressive resolution reads work together to allow users to quickly load coarse representations of their data to begin performing their analysis while additional data are loaded in the background, and to analyze large data sets on low-power devices (Fig. 3.10). Without this capability, it may be possible to work with the data only on a large memory workstation or small cluster, impacting the scientists' ability to perform their analysis.

To support fast multiresolution reads, the BAT layout can be queried at different quality levels to return a representative subset of the data using the LOD particles stored at the treelet inner nodes. 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, where zero corresponds to loading nothing and one corresponds to loading the entire data set. Internally, this parameter is

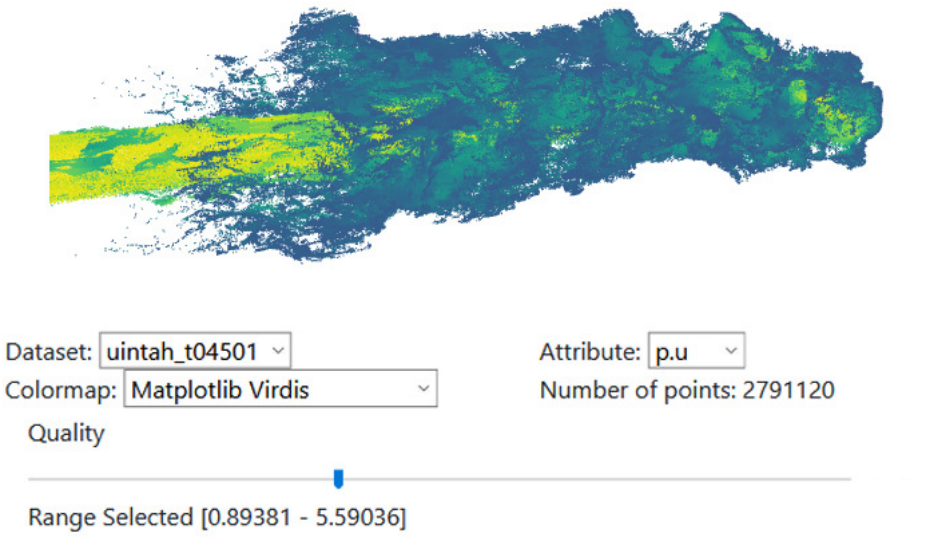


Fig. 3.10: A prototype web viewer client that progressively streams data from a server. The server uses the BAT layout to progressively load and send data back to clients and apply spatial- and attribute-based filtering.

remapped using a log scale to provide a smoother quality progression as the number of LOD particles stored grows exponentially with the tree level. The quality value is then scaled to a maximum treelet depth to traverse to, based on the total tree depth. When performing progressive reads, the previous quality is mapped to a previously processed minimum treelet depth. The tree is then traversed to the maximum treelet depth, processing only points above the minimum depth. To provide a smoother LOD progression, the quality parameter is also used to compute a percentage of the points at the maximum level to process, reducing issues with LOD pop-in. Spatial and attribute filtering can also be performed when using progressive resolution reads.

3.4 Parallel I/O Scaling Study

This section demonstrates the scalability and portability of the AUG and aggregation tree approaches for spatially aware two-phase I/O are through an extensive set of benchmarks on parallel writes and reads. The benchmarks are run on both fixed uniform and time-varying nonuniform particle distributions. The bandwidth achieved by the presented I/O strategies is compared against standard file per process and single-shared file approaches, and studied in depth through timing breakdowns of the stages comprising the I/O pipelines.

The parallel I/O benchmarks are run on four HPC systems with different I/O archi-

tectures: Mira, Theta, Stampede2, and Summit. Mira uses a 5D Torus network topology and GPFS filesystem, which is capable of 240GB/s peak bandwidth [44]. Theta uses a Cray Dragonfly topology network and a Lustre filesystem, which is capable of 172GB/s peak bandwidth [113]. On Theta, the Lustre system to use 48 stripes (48 OSTs), with the stripe size set to 8MB, as recommended by the ALCF guidelines for I/O performance on Theta [53]. Stampede2 uses a Lustre file system and the benchmarks write to the scratch system, which is capable of 330GB/s peak write bandwidth [263]. On Stampede2, the Lustre output is configured to use a stripe count of 32 and stripe size of 8MB. Summit uses a GPFS filesystem with a peak write speed of 2.5TB/s [206]. Both systems use a fat-tree topology network, capable of 100Gbps on Stampede2 and 184Gbps on Summit. Each Mira compute node has a 16-core IBM PowerPC A2 CPU; each Theta node has a Intel Xeon Phi Knight's Landing CPU; Stampede2 has multiple compute partitions, the benchmarks are run on the dual socket Xeon Skylake (SKX) nodes; each Summit node has two POWER9 CPUs and six Volta V100 GPUs.

3.4.1 Weak Scaling on Uniform Particle Distributions

To provide a baseline comparison of the presented I/O strategies against standard ones, they are evaluated using a weak scaling study on a generated uniform particle distribution. The generated uniform distribution provides a workload representative of particle-based simulations performed using the Uintah computational framework [21]. Uintah is a general purpose computational framework, and has been used for numerical modeling of fluid-structure interactions, computational fluid dynamics, solid mechanics, and multiphysics simulations. As part of its framework, Uintah includes support for a range of particle-based models, for example the material point method [21] and Lagrangian particle transport [239]. The Uintah computational framework has been demonstrated at core counts approaching 768k [21]. Workloads with 32,768 (32k) and 65,536 (64k) particles per core are used in the experiments to provide data volumes similar to large-scale simulations. Each particle stores 124 bytes of data representing its position and physical other attributes, corresponding to 4MB and 8MB of data per core in the tested workloads respectively. IOR [254] is used to evaluate the performance of standard approaches by writing an equivalent amount of data using IOR's file per process, single-shared file (MPI I/O), and HDF5 shared file modes. To

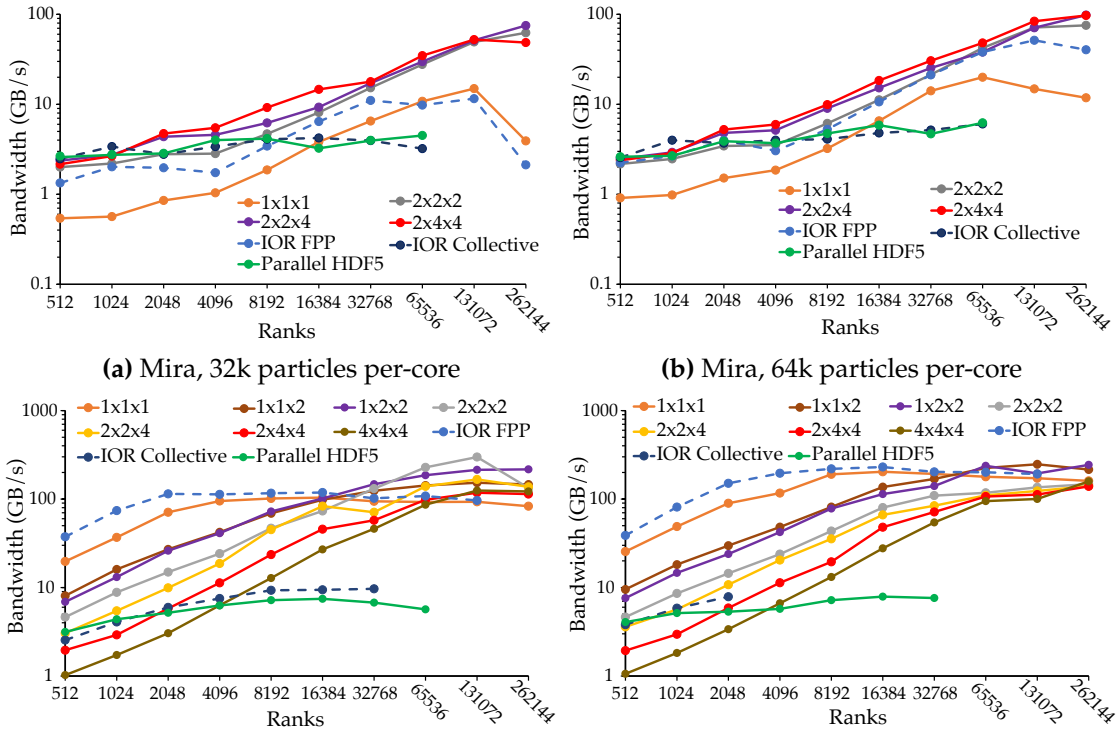
measure the I/O time that would be observed by a real-world simulation, the benchmarks do not perform an explicit fsync. To avoid the impact of OS caching when in the read benchmarks, files are read on different ranks than those that wrote them.

3.4.1.1 Aggregation Using the Adjustable Uniform Grid

The AUG aggregation strategy is evaluated on the Mira and Theta HPC systems using the 32k and 64k particle per core workloads. The benchmarks are run on both machines scaling from 512 to 262,144 ranks, thus varying the amount of data generated in the benchmark from 2GB to 1TB at 32k particles per core, and 4GB to 2TB at 64k particles per core. The benchmarks are run at multiple APFs to evaluate the impact of the amount of aggregation performed on I/O performance; testing APFs of $(1,1,1)$, $(1,1,2)$, $(1,2,2)$, $(2,2,2)$, $(2,2,4)$, $(2,4,4)$ and $(4,4,4)$. In preliminary scaling tests, Mira was found to perform better with larger APFs compared to Theta, which performed better at smaller ones. Based on this observation, the APF configurations $(1,1,2)$ and $(1,2,2)$ were not run on Mira.

The results of the scaling studies on Mira are shown in Figs. 3.11a and 3.11b, along with the IOR and Parallel HDF5 results. Both IOR's file per process and the proposed AUG approach with an APF of $(1,1,1)$ (i.e., file per process), begin to encounter scaling issues at high levels of parallelism (131,072 ranks with 32k particles and 65,536 with 64k particles). At these scales, file per process I/O encounters issues with the parallel file-system failing to handle the massive number of small output files efficiently. The single-shared file approaches, evaluated using IOR's collective mode and Parallel HDF5, are also observed to not scale beyond medium core counts. Collective I/O to output a single-shared file requires global synchronization and communication between the ranks, and it is this global communication that does not scale well. On Mira, the proposed AUG approach using APFs of $(2,2,4)$ and $(2,4,4)$ provides the best performance, and scales well up to the largest configurations tested at 262,144 ranks. At this scale, AUG achieves a peak throughput of 75GB/s on the 32k particle workload using an APF of $(2,2,4)$ and 98GB/s on the 64k particle workload using an APF of $(2,4,4)$, The two-phase I/O approach employed in AUG is able to avoid the bottlenecks that impact file per process and single-shared file at scale by adjusting the amount of aggregation performed to achieve high-performance I/O.

On Theta (Figs. 3.11c and 3.11d), exhibits very different I/O performance characteristics compared to Mira. Theta uses a Lustre filesystem, which is known to perform well with file per process I/O, even at large scales. As would then be expected, IOR's file per process mode and AUG with an APF of $(1,1,1)$ (i.e., file per process) perform well on Theta, and scale much better than the same methods did on Mira. However, even on the Lustre filesystem file per process I/O is outperformed by higher APFs at large core counts where the time spent creating the massive number of files becomes a bottleneck. Although the $(1,2,2)$ configuration is outperformed by the $(1,1,1)$ configuration, it overtakes it at higher core counts and achieves the best performance at 262,144 cores. The $(1,2,2)$ configuration achieves a peak throughput of 216GB/s on the 32k particle workload and 243GB/s on the



(a) Mira, 32k particles per-core

(b) Mira, 64k particles per-core

(c) Theta, 32k particles per-core

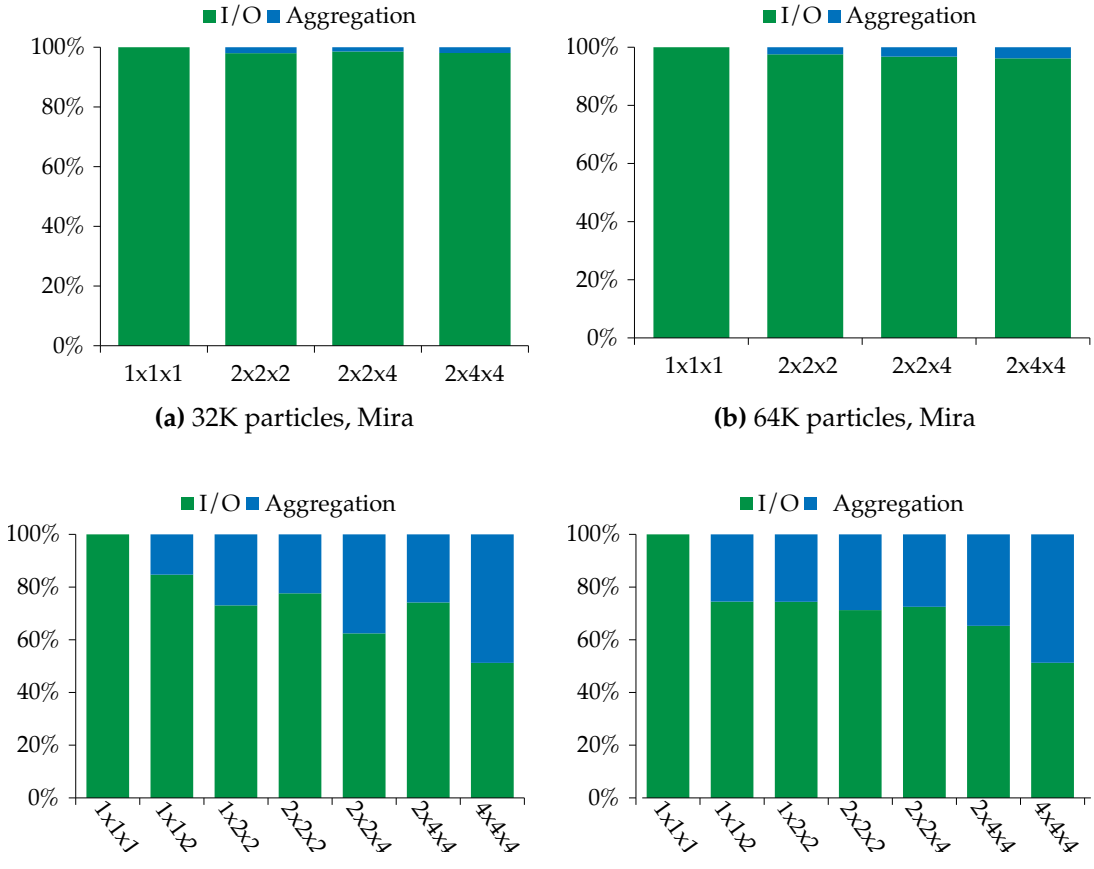
(d) Theta, 64k particles per-core

Fig. 3.11: Parallel write weak scaling for different configurations of two-phase I/O using Adjustable Uniform Grid aggregation on Mira (first row) and Theta (second row). The best Aggregation Partition Factor is machine and workload dependent, and is left exposed to simulations as a tuneable parameter. IOR and Parellel HDF5 experiments are provided as a reference for file per process and collective I/O performance.

64k particle workload[1]. Single-shared file I/O provides poor performance overall on Theta, attributable to both the expensive global communication step and the Lustre filesystem's better support for file per process style I/O.

Timing breakdowns of the time spent on communication for aggregation and I/O in the pipeline are shown in Fig. 3.12 to study how different APFs affect performance on Mira and Theta. A key difference between Theta and Mira is the cost of data aggregation, with data aggregation occupying a greater percentage of time on Theta an Mira at equivalent APFs. The higher cost of aggregation, combined with the Lustre filesystem's better support

[1]Note that this exceeds the 170GB/s peak bandwidth advertised for Theta. The filesystem can return back to the caller before the data are fully flushed to disk, providing the appearance of higher performance.



(a) 32K particles, Mira

(b) 64K particles, Mira

(c) 32K particles, Theta

(d) 64K particles, Theta

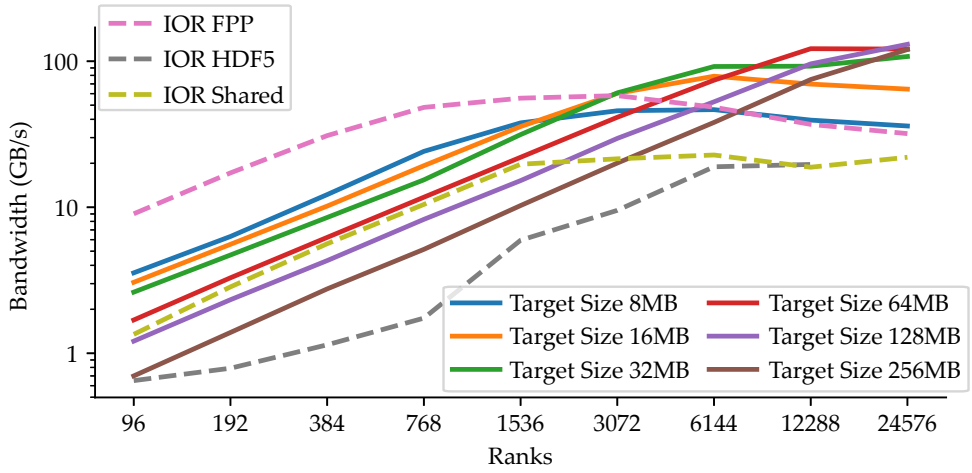
Fig. 3.12: Time profiles for different aggregation configurations on Mira (a, b) and Theta (c, d). More time is spent in aggregation on Theta compared to Mira for the same APF configurations, which indicates that smaller APFs, and thus less aggregation, should be preferred on Theta, while the opposite is true on Mira.

for writing many files, explains why lower APFs perform better on Theta than higher ones, and vice versa on Mira. For example, when comparing the performance of a $(2, 2, 4)$ APF on Theta and Mira, less than 10% of the total time is spent on aggregation on Mira, while on Theta it accounts for 30-40% of the total time. This observation, combined with the performance results discussed previously, indicate that data aggregation tends to be more expensive on Theta, and smaller APFs should be preferred, while the opposite is true on Mira. By exposing the APF as a configurable parameter, simulations using the AUG two-phase I/O approach are able to adjust the I/O strategy to best fit the target HPC system to portably achieve high bandwidth writes.

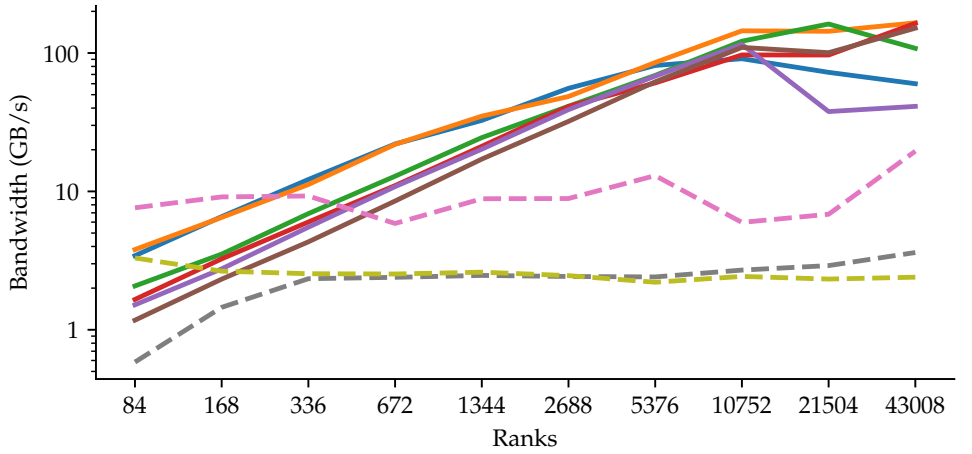
3.4.1.2 Aggregation Using the Aggregation Tree

The aggregation tree approach is evaluated on the Stampede2 and Summit HPC systems using the 32k particle workload. In each benchmark, the data are written and read 15 times and the geometric mean of bandwidth achieved recorded, as done in the IO500 [132]. The benchmarks also study the effect of the target file size on write and read performance by varying the target file size from 8MB (file per process) to 256MB ($\approx$63 ranks per file).

On both Stampede2 and Summit (Fig. 3.13), the proposed aggregation tree two-phase I/O approach outperforms file per process and single-shared file I/O at higher core counts. The results observed are similar to those seen for AUG on Theta and Mira. Stampede2 (Fig. 3.13a) uses a Lustre filesystem, and thus sees better performance in IOR's file per process mode and in the aggregation tree with an 8MB target size than is observed on Summit, which uses a GPFS file system. However, as observed in the previous section, the file per process approaches do not scale to higher core counts compared to configurations that perform more aggregation, due to the overhead of creating the large number of files. Single-shared file I/O achieves relatively better performance on Stampede2 than was observed on Theta, although it does not scale to large core counts. On Stampede2, the aggregation tree achieves a peak bandwidth of 145GB/s at 24,576 cores in the 128MB target file size configuration. Summit (Fig. 3.13b) uses a GPFS file system, and sees poor scalability for both file per process and single-shared file I/O, trends that are in line with observations on Mira in the previous section, which also uses a GPFS filesystem. On Summit, the aggregation tree achieve a peak bandwidth of 165GB/s at 43,008 cores in the



(a) Stampede2

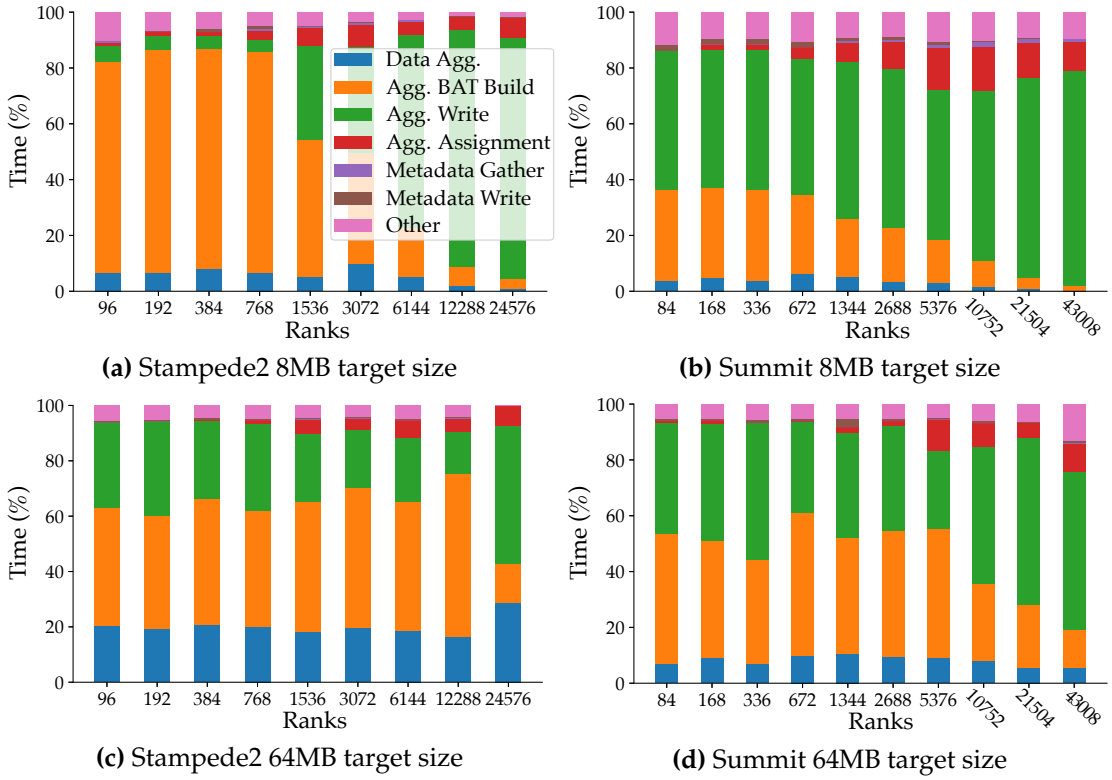


(b) Summit

Fig. 3.13: Write bandwidth weak scaling on the fixed uniform test data compared to IOR benchmarks. At scale, the proposed aggregation tree approach outperforms standard file per process and single-shared file approaches.

16MB target file size configuration, with the 64MB configuration only slightly slower at 163GB/s. Overall, the performance difference between smaller and larger target file sizes is smaller on Summit than on Stampede2, with both systems performing better with larger target sizes at higher core counts.

Timing breakdowns of two target file sizes, 8MB and 64MB, across the scaling runs on Stampede2 and Summit are shown in Fig. 3.14, to study where time is spent in the aggregation tree I/O pipeline. Within the aggregation tree pipeline, the bulk of time is spent writing the aggregator files to disk (Agg. Write), constructing the BATs on each aggregator (Agg. BAT Build), and transferring data to aggregators (Data Agg.). The total time spent



(a) Stampede2 8MB target size

(b) Summit 8MB target size

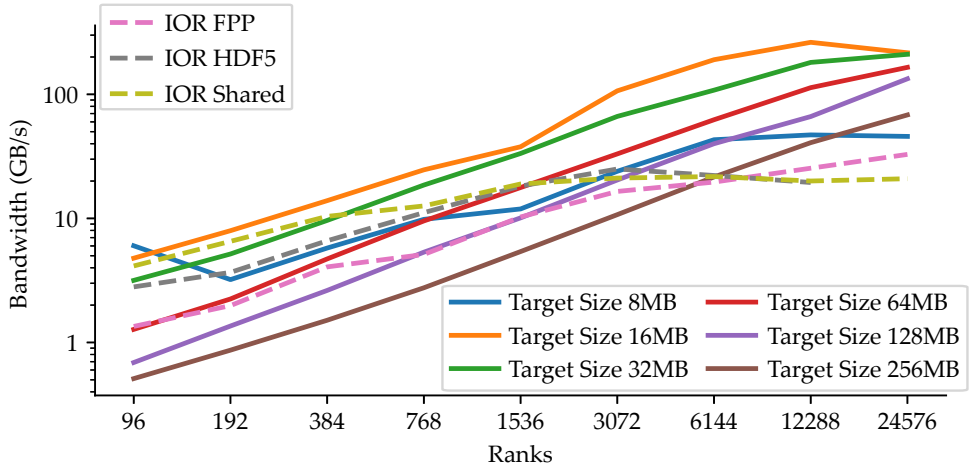
(c) Stampede2 64MB target size

(d) Summit 64MB target size

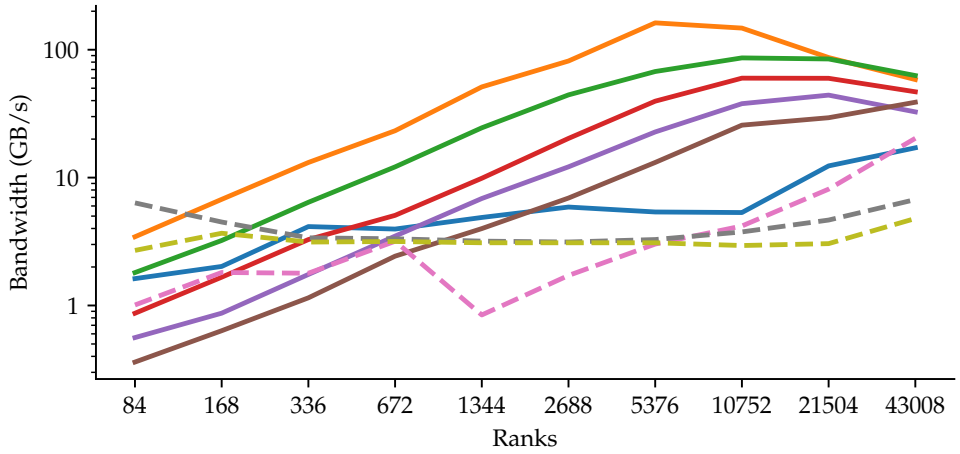
Fig. 3.14: Timing breakdowns on Stampede2 and Summit. In the scaling regime of each target size, the relative time spent in each component remains similar.

constructing the Aggregation Tree and aggregator assignment (Agg. Assignment) varies with core count and tree depth. The depth of the tree is related to the target file size; smaller target sizes will have to partition the ranks further to reach the smaller desired leaves, whereas larger ones can terminate the build earlier. Similarly, the cost of communicating the aggregator assignment information to each aggregator during Agg. Assignment also grows with the number of assigned aggregators, as each must be sent a list of the ranks in its aggregation group. When comparing the major costs of the 8MB and 64MB target size runs, the 64MB configuration is found to spend a relatively consistent amount of time in each component as the benchmark is scaled 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, a larger percentage of time is spent constructing the BAT layout than on Summit. 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, the aggregation tree approach outperforms file per process and



(a) Stampede2



(b) Summit

Fig. 3.15: Read bandwidth weak scaling on the fixed uniform test data, compared to IOR benchmarks. The aggregation tree's two-phase parallel read strategy outperforms standard file per process and single-shared file reads at scale.

single-shared file strategies beyond moderate core counts on both systems (Fig. 3.15). Performance trends similar to those seen on writes are observed, 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. The aggregation tree's two-phase I/O approach allows selecting the target size to avoid both issues and achieve high bandwidth reads. On Summit, the scaling trend on small to medium size aggregation settings flattens off or begins decreasing by 43k cores; however, 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.

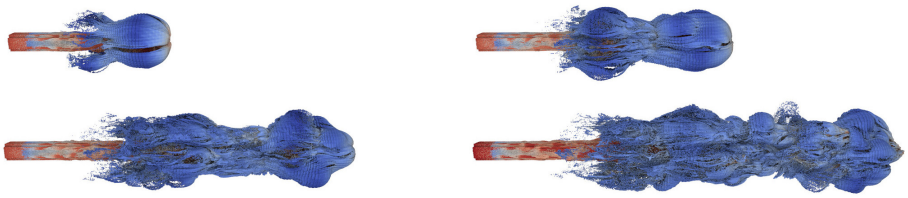
3.4.2 Adjustable Uniform Grid vs. Aggregation Tree

A set of benchmarks on time-varying nonuniform particle distributions from two simulations (Fig. 3.16) is used to compare the performance impact of the improved adaptivity provided by the aggregation tree over the AUG. For ease of comparison, the AUG approach is adjusted to take a target file size as input instead of an APF. The AUG builder will then assume a uniform particle distribution over the ranks that have particles, based on the average number of particles per rank. The aggregation tree 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 Skylake Xeon nodes on Stampede2. The two data sets used are representative of different time-varying nonuniform particle distributions used in simulations: the coal boiler and the dam break.

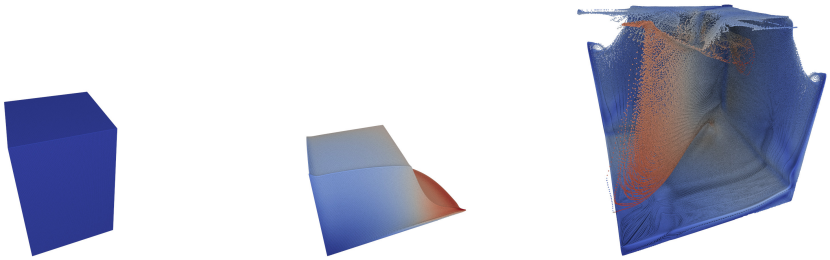
The coal boiler (Fig. 3.16a) is a real-world simulation performed using Uintah [191], simulating the injection of coal particles into a boiler. Uintah is a computational framework that has been used to simulate large-scale nonuniform particle distributions scaling up to 512k cores of Mira (65% of the machine) [21]. The domain is partitioned using a 3D grid and resized to fit the data bounds as they change over time. At time step 501, the simulation contains 4.6M particles and reaches 41.5M at time step 4501. The I/O benchmark is run using 1536 ranks. Each particle saves three floating point coordinates and seven double precision attributes.

The dam break (Fig. 3.16b) 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 [260], which accurately represents the I/O workload of production applications. Migrating the mini-app's original output required minimal changes, replacing Silo [163] write calls with corresponding API calls from to the library implementing the presented I/O strategies. The domain is partitioned among the ranks using a 2D grid along x and y (the floor) to achieve better compute load balance. Two versions of the dam break are used 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, the aggregation tree strategy improves write performance by up to



(a) The coal boiler, time steps: 1501, 2501, 3501, 4501



(b) The dam break, time steps: 0, 1001, 4001.

Fig. 3.16: The time-varying nonuniform simulation data used in the benchmarks.

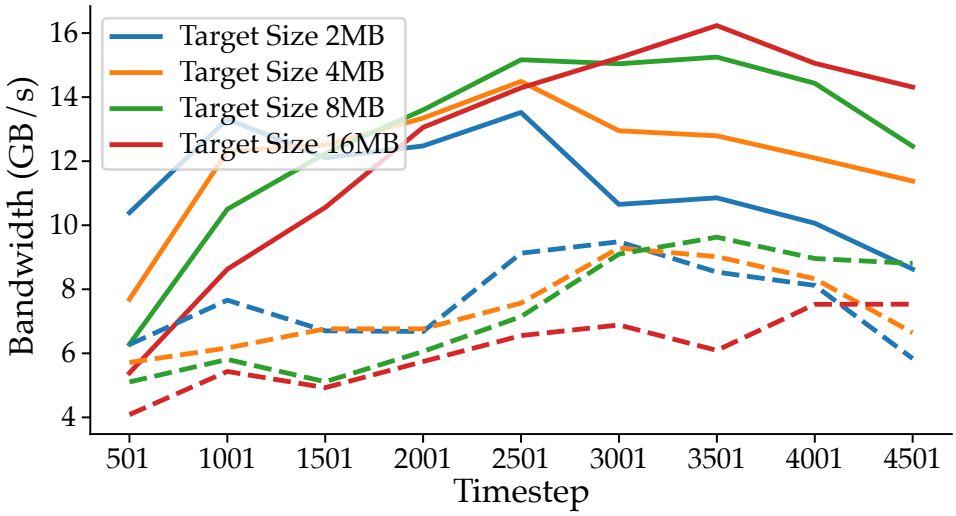
2.5× compared to AUG aggregation (Fig. 3.17a). Parallel reads on the adaptively aggregated data can be up to 3× faster (Fig. 3.17b). As the number of particles in the simulation increases, performance decreases at lower target sizes, whereas larger target sizes surpass them. Similar trends are observed for reads. When comparing timing breakdowns of both approaches at 8MB (Fig. 3.18), the aggregation tree strategy is found to spend less time in the major steps of the I/O pipeline.

These performance improvements are the result of the aggregation tree's adaptive aggregation ensuring a more balanced I/O workload. For example, on the 8MB target size runs at time step 4501, AUG aggregation outputs 296 files, with an average size of 10.2MB and standard deviation of 13.9MB. The aggregation tree approach writes 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 file written by the aggregation tree approach was 36.6MB. If a rank's local data exceed the target file size, both methods can output files larger than the target size, as the local data on a rank will not be subdivided.

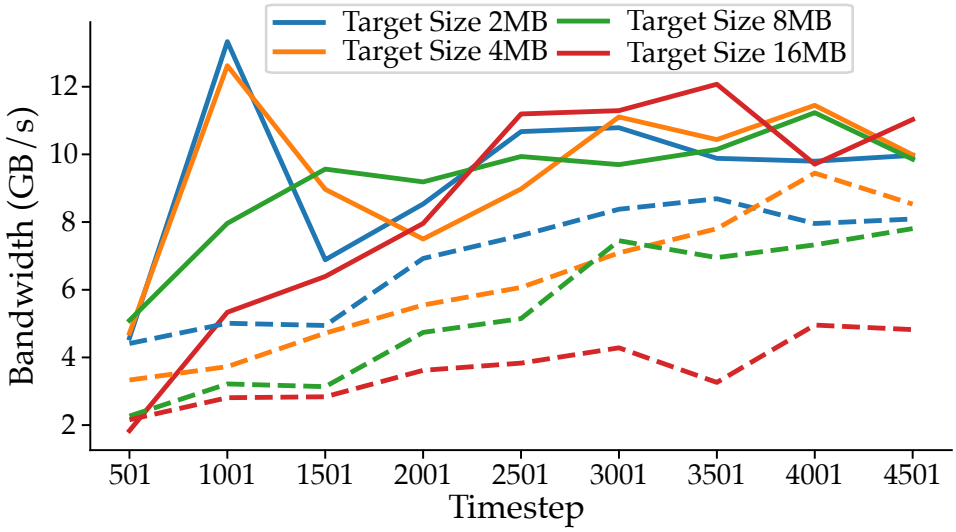
On the 2M dam break, the file per process mode of both strategies achieves the best (and similar) write performance (Fig. 3.19a); however, the adaptively written data provide slightly faster reads (Fig. 3.19c). On the 8M dam break, the 3MB target size using adaptive aggregation achieves the best write performance overall (Fig. 3.19b), at a $1.5 - 2\times$ speed-up over AUG aggregation at the same target size, with up to 3× 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 smaller performance difference is also partially attributable to the smaller size (in bytes) of each particle in the dam break compared to the coal boiler.

The dam break contains a fixed number of particles that move through the domain, and an ideal I/O strategy would be expected to achieve constant write times over the time series. A timing breakdown of the 3MB run on the 8M dam break (Fig. 3.20), demonstrates that the

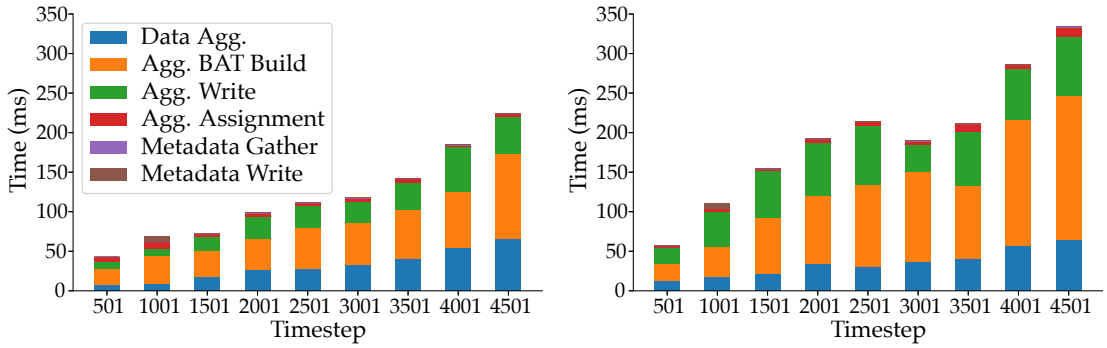


(a) Write bandwidth.



(b) Read bandwidth.

Fig. 3.17: Aggregation tree vs. AUG I/O on the coal boiler time series on 1536 ranks. Dashed lines indicate AUG aggregation. The aggregation tree is able to improve write and read performance for imbalanced I/O workloads.



(a) Adaptive aggregation **(b)** AUG aggregation

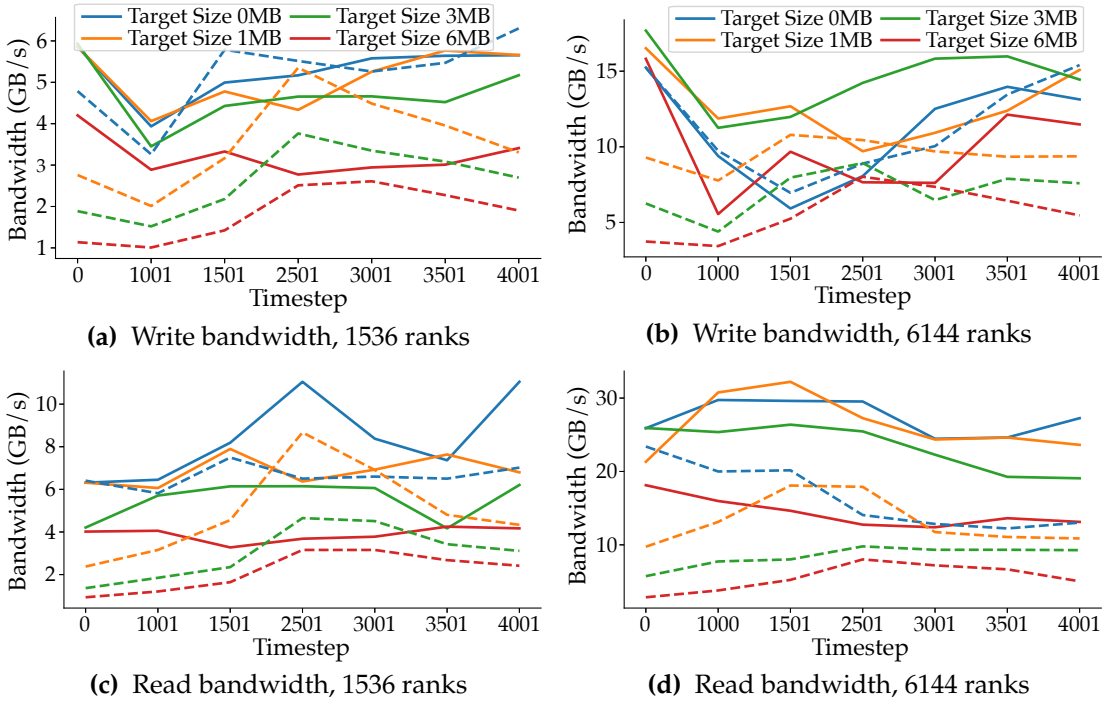
Fig. 3.18: Breakdowns of adaptive vs. AUG I/O on the coal boiler, 8MB target size. The improved load balance achieved by the aggregation tree approach reduces time spent in the major components of the pipeline, improving write performance.

AUG is strongly affected by the simulation's particle distribution, whereas the aggregation tree achieves nearly constant write times.

Based on these experiments, the aggregation tree should be configured to use 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 roughly $16:1$ to $64:1$ or higher, to avoid overloading the file system with large numbers of files. In simulations where particles are added over time, such as the coal boiler, the target size should be increased over time to maintain high-performance I/O. These recommendations could be used to automatically pick a default target size.

3.5 Evaluation on Post Hoc Visualization Reads

To examine the suitability of the BAT layout for visualization and analysis, an evaluation is conducted of its support for low-latency progressive multiresolution reads, accelerating attribute-based queries, reporting the memory overhead required to store the structure. The following evaluation is performed using a single threaded process on a desktop with an i9–9920X CPU, 128GB of RAM, and 1TB Samsung 970 NVMe drive. The benchmarks are run reading data output by the aggregation tree approach, which constructs the BAT layout during the I/O pipeline.



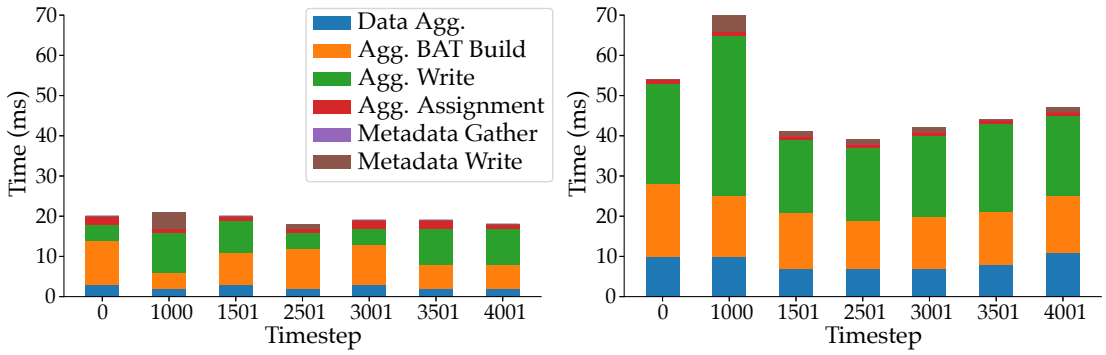
(a) Write bandwidth, 1536 ranks

(b) Write bandwidth, 6144 ranks

(c) Read bandwidth, 1536 ranks

(d) Read bandwidth, 6144 ranks

Fig. 3.19: Aggregation tree vs. AUG aggregation on the dam break time series. Dashed lines indicate AUG aggregation. The performance improvement provided by the aggregation tree increases at larger scales for imbalanced simulations.



(a) Adaptive aggregation

(b) AUG aggregation

Fig. 3.20: Breakdowns of adaptive vs. AUG I/O on the 8M dam break, 3MB target size. The aggregation tree approach maintains nearly constant I/O times, whereas the AUG approach is influenced by the changing distribution of the particles.

3.5.1 Low-Latency Progressive Multiresolution Reads

The BAT layout's suitability for low-latency progressive multiresolution reads is evaluated through a benchmark representative of a typical progressive read use case. The benchmark starts at a coarse quality level of 0.1 (approximately 10% of the data), and successively requests higher quality levels in increments of 0.1 until the entire data set is loaded. The time spent to traverse the tree and process each requested point is recorded; however, the time spent in the user callback is not, as its cost will inherently vary by the desired application (e.g., copying data, performing computation, etc.). The BATs were built in the previous section, storing eight LOD particles per treelet inner node and up to 128 particles per treelet leaf. The average read performance achieved on the Coal Boiler is shown in Table 3.1, and on the Dam Break in Table 3.2.

Similar performance is achieved in both data sets when reading data aggregated at different target sizes. On the Coal Boiler, multiresolution reads perform slightly better on data written with a 4MB target size, and on the Dam Break on data written with a 3MB target size. The largest factor determining read performance is the number of points queried. The quality level represents a percentage of the total data to query, and thus level 0.1 corresponds to more points on the Coal Boiler than on the Dam Break. When comparing the data sets in terms of points per millisecond read throughput, both the Coal Boiler and 8M Dam Break achieve similar throughputs. The 2M Dam Break achieves higher throughput, likely due to its small size, which allows more data to remain cached by the OS.

Although not encountered in the benchmarks presented, it is possible for small target size or file per process writes to produce more files than can be open at once. In this case, a caching mechanism can be used to close the oldest files when new ones must be opened; however, the added overhead of opening and closing files would have a large impact on read performance.

The BAT layout does not impose a specific visual representation on visualization tasks, as the best choice is often domain specific. To provide an example of the quality progression provided by the layout, a simple LOD approach is used, where coarser representations are displayed with increased particle radii to fill holes and preserve the overall shape of the object [164], shown on the Coal Boiler in Fig. 3.21.

Table 3.1: Progressive single-thread read times and throughput on time step 4501 of the Coal Boiler, 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

Table 3.2: 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)
1536 Ranks, 2M total points		
0MB	2.7ms	71441 pts/ms
1MB	2.7ms	70012 pts/ms
3MB	2.6ms	72997 pts/ms
6144 Ranks, 8M total points		
0MB	12.7ms	57659 pts/ms
1MB	12.8ms	57703 pts/ms
3MB	12.4ms	58926 pts/ms

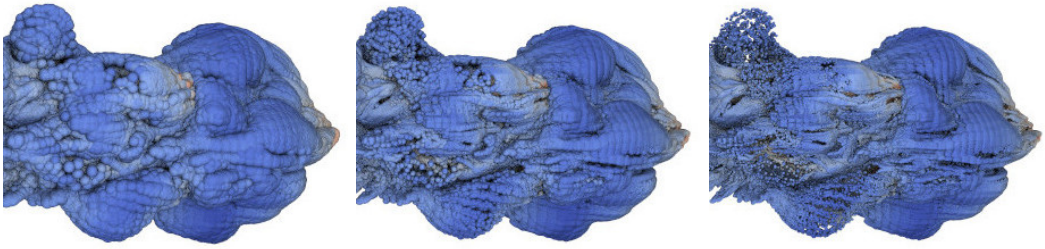


Fig. 3.21: The visual quality progression on the Coal Boiler, time step 4501. Quality increases from left to right: 0.2, 0.4, 0.8.

The error introduced when computing statistical analyses on coarser representations is also of interest, as this indicates how useful the coarser representations can be expected to be for visualization tasks beyond rendering. The error of various statistical measures computed on the data at different quality levels is shown in Fig. 3.22. For most attributes, the naive strategy employed for selecting the LOD particles provides improvement in error as the quality is increased, and a tolerable maximum error on a coarse representation (0.1 quality). However, the LOD selection is based on sampling the spatial distribution of the

particles, leading to less consistent error improvement for those attributes not well sampled by this approach. For example, the error actually increases for some attributes on the Coal Boiler (Fig. 3.22a) and Dam Break (Fig. 3.22c) before decreasing again, indicating that these were poorly represented by coarser levels of the tree.

3.5.2 Query Acceleration and Memory Overhead

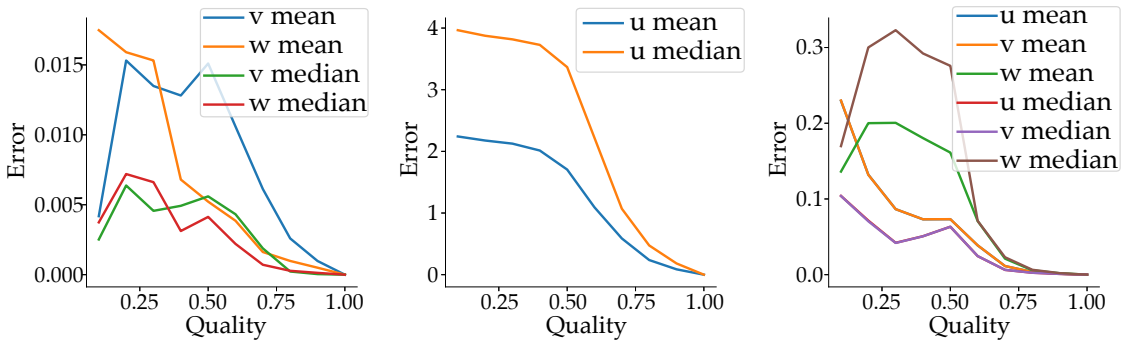
The core of the BAT layout is a k-d tree, which provides built-in support for fast spatial subregion, nearest neighbor, and radius queries. The 32-bit bitmap indices tracked for each node are used to accelerate attribute subset queries. Although the accuracy of these bitmaps is intentionally restricted compared to conventional bitmap indexing approaches, the filtering selectivity is on par with explicitly storing exact value ranges at each node, while requiring less memory.

Within each BAT, the bitmaps are stored relative to the local value range on the aggregator to improve the bin granularity. When using the global range, each bin would correspond to $\approx$3.1% of the range. However, when using the aggregator range, the bin sizes correspond to 1.2% of the range on average across attributes in the Coal Boiler and Dam Break data sets. A set of generated attribute subset queries is performed that query up to four disjoing ranges on each attribute to evaluate the cost of querying the data compared to storing just a value range at each node. Overall, there is a negligible performance difference compared to storing the value ranges at each node, with the 32-bit bitmaps processing and discarding an extra 1.4% of false positives.

The memory overhead of storing the bitmap dictionaries is extremely low. Compared to storing just the raw set of particle positions and attributes, the entire BAT structure (the tree nodes, bitmap dictionaries, and indices) accounts for an additional 0.9% of memory on average. The same layout configured to store ranges at each inner node would require an extra 2.1% of additional memory. The 0.9% overhead is also significantly lower than the space required for standard bitmap indexing approaches and multiresolution layouts that rely on adding or duplicating particles.

3.6 Summary

General purpose I/O libraries typically do not take into account the requirements of post hoc visualization tasks, instead focusing solely on raw I/O throughput. This



(a) Coal Boiler **(b)** Coal Boiler **(c)** Dam Break

Fig. 3.22: The error in basic attribute statistics as the quality level is increased, shown for the final time step of each data set.

chapter has presented work conducted by myself and collaborators on devleoping scalable spatially aware aggregation strategies for two-phase I/O that output the particle data in a layout readily suitable for post hoc visualization. Although the proposed visualization tailored BAT layout is constructed during the I/O pipeline, it does not incur a significant performance penalty. Building the visualization focused data layout during I/O allows eliminating the need for an expensive post hoc data conversion, or even use of the layout in situ as discussed later. Through the use of a tuneable two-phase I/O pipeline and a data layout optimized for fast parallel construction, the presented I/O approaches achieve better I/O performance than standard file per process and single shared file approaches, and demonstrate portability across a breadth of HPC systems. The two I/O strategies presented, the adjustable uniform grid and the aggregation tree, provide different trade-offs with respect to adaptivity and computational cost. The AUG is cheap to compute, but is not able to adapt to a wide variety of nonuniform particle distributions. The aggregation tree requires additional computation to construct a more balanced I/O workload; however, the improved load balance vastly improves the overall performance of the I/O pipeline for unbalanced particle distributions. The presented BAT data layout is suitable for fast parallel construction during I/O, and makes careful trade-offs between memory overhead and its support for LOD and attribute-based query filtering.

CHAPTER 4

POST HOC VISUALIZATION

Post hoc visualization forms a key component of the simulation visualization pipeline. It is during this step that scientists gain insight into the simulated phenomena by asking and answering questions about the data through interactive visualization. Scientists can use visualization systems to produce images, charts, segmentations, and other statistical or visual representations to explore their hypotheses about the simulation. When processing the full-resolution data, these tasks can be performed on a visualization cluster. When working with subsets of the data or multiresolution representations, they can be done locally on a workstation, in virtual reality, or in the web browser. Throughout the history of visualization, a continuing challenge has been developing methods that can scale to the ever-growing data set sizes output by simulations, or that can make such large data accessible on typical workstations or laptops. With each increase in HPC system compute and memory capabilities, scientists can simulate increasingly accurate simulations. These simulations run at higher levels of parallelism and output more data, challenging widely used visualization methods.

This chapter presents work in three key areas that face challenges today when performing post hoc visualization of massive data sets: scalable distributed rendering (Section 4.1), virtual reality (Section 4.2), and the web browser (Section 4.3). These works are discussed in order of decreasing computational capability of the target platform, and thus increasing need for adaptive and multiresolution data management or processing.

Section 4.1 discusses the distributed framebuffer (DFB), a flexible, asynchronous image-processing framework for scalable distributed rendering [281]. The need for high-performance distributed parallel rendering is growing, spurred by trends in increasing data set sizes and the desire for higher fidelity or interactivity. Meeting these demands poses new challenges to existing rendering methods, requiring scalability across a spectrum of memory and

compute capabilities on HPC systems. Whereas the growth in data set sizes demands a large amount of aggregate memory, the desire for more complex shading and interactivity demands additional compute power. A large number of applications' needs fall somewhere in between these extremes of memory or compute scaling, requiring a combination of additional memory and compute. Such applications are not well addressed by existing techniques that focus on scalability in terms of aggregate memory or compute capacity alone. The DFB achieves performance superior to the state of the art for standard use cases, while also providing the flexibility to support a wide range of parallel rendering algorithms and data distributions.

Section 4.2 presents a design study on how consumer VR systems, coupled with state-of-the-art data management and visualization techniques, can improve the workflow of connectomics researchers through the development of a VR tool for neuron tracing (VRNT) [189, 277]. The VRNT is developed through a direct collaboration with neuroanatomists in the Angelucci lab at the University of Utah, to enable tracing neurons in microscope scans of the visual cortex of nonhuman primates in VR. State-of-the-art microscopy techniques can easily produce terabytes worth of volumetric data, which few existing tools for connectomics are capable of handling. Such data sizes also far exceed that which could be rendered naively at a satisfactory frame rate in VR, requiring the development of a real-time page-based data processing system. The VRNT design process explores the use of different rendering, interaction, and navigation methods, as well as force feedback to improve the quality and speed of neuron tracing. A pilot study is conducted using the VRNT to demonstrate that the immersive 3D interface provided through VR substantially improves the overall user experience by allowing neuroanatomists to directly interact with their data.

Finally, Section 4.3 discusses work on improving access to visualization systems capable of handling large-scale scientific data sets by developing new techniques for the web browser [279]. Information visualization applications have become ubiquitous, in no small part thanks to the ease of wide distribution and deployment enabled by the web browser. Scientific visualization applications that rely on native code libraries and parallel processing, have been less suited to such widespread distribution, as web browsers do not provide the required libraries or compute capabilities. This section revisits this gap in visualization

technologies, and explores how new web technologies, WebAssembly and WebGPU, can be used to deploy powerful visualization solutions for large scientific data in the browser. To enable visualization of large volumetric data sets entirely in the browser, a new GPU-driven isosurface extraction method for block-compressed data sets is presented, block-compressed marching cubes (BCMC). BCMC is designed for interactive isosurface computation on large volumes in resource-constrained environments, such as the browser, and is demonstrated on interactive isosurface computation on volumes with up to 1B voxels and up to 1TB in size.

4.1 A Distributed FrameBuffer for Scalable Ray Tracing

Data-parallel and image-parallel rendering can be seen as the two endpoints of a spectrum of possible rendering configurations that blend between distributing subregions of the data or the image as units of work for processing. Applications that fall somewhere in between these two extremes, or those seeking to go beyond these common use cases, can quickly run into issues when using existing methods. For example, whereas a master-worker setup is well suited to image-parallel ray tracing, if the renderer wants to perform additional postprocessing operations (e.g., tone-mapping, progressive refinement), or handle multiple display destinations (e.g., display walls), the master rank quickly becomes a bottleneck. Similarly, whereas existing sort-last compositing algorithms are well suited to statically partitioned data-parallel rendering, extending them to support partially replicated or more dynamic data distributions for better load balancing is challenging. Finally, standard sort-last compositing methods operate bulk-synchronously on the entire frame, and are less suited to tile-based ray tracers in which small tiles are rendered independently in parallel.

This section describes the algorithms and software architecture, the distributed framebuffer, developed to support distributed parallel rendering, with the goal of addressing the above issues to provide an efficient and highly adaptable framework suitable for a range of rendering applications [281] (Fig. 4.1). The distributed framebuffer (DFB) is built on a tile-based work distribution of the image processing tasks required to produce the final image from a distributed renderer. These tasks are constructed per tile at runtime by the renderer and are not necessarily tied to the host application's work or data distribution, providing the flexibility to implement a wide range of rendering algorithms and distribute

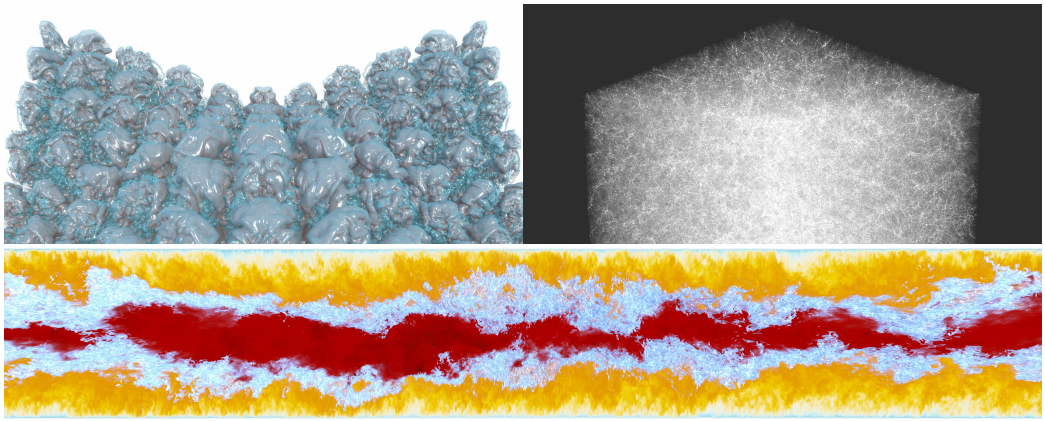


Fig. 4.1: Large-scale interactive visualization using the distributed framebuffer. Top left: Image-parallel rendering of two transparent isosurfaces from the Richtmyer-Meshkov [54] (516M triangles), 8FPS with a 2048^2 framebuffer using 16 Stampede2 Intel Xeon Platinum 8160 SKX nodes. Top right: Data-parallel rendering of the Cosmic Web [131] (29B transparent spheres), 2FPS at 2048^2 using 128 Theta Intel Xeon Phi Knight's Landing (KNL) nodes. Bottom: Data-parallel rendering of the 951GB DNS volume [167] combined with a transparent isosurface (4.35B triangles), 5FPS at 4096×1024 using 64 Stampede2 Intel Xeon Phi KNL nodes.

compute-intensive image processing tasks. The DFB performs all communication and computation in parallel with the renderer using multiple threads to reduce compositing overhead. Although the DFB is flexible enough to support renderers across the spectrum of memory and compute scaling, it does not make a performance trade-off to do so.

4.1.1 The Distributed FrameBuffer

At its core, the DFB is not a specific compositing algorithm per se, but a general framework for distributed rendering applications. A renderer using the DFB specifies a set of tasks to be executed on the rendered image and per tile dependency trees for the tasks. The tasks are parallelized over the image by subdividing it into tiles, where each tile is owned by a unique rank—the tile owner—responsible for executing tasks for that tile. If task dependencies are produced on ranks other than the tile owner, the DFB will route them over the network to the owner. The tile dependency trees are specified per tile and per frame, allowing for view- and frame-dependent behavior.

The tile processing pipeline involves three stages (Fig. 4.2). First, the dependency tree is constructed by the tile operation as task dependencies are received from other ranks. Once the entire tree has been received, the finished tile is computed by the tile operation

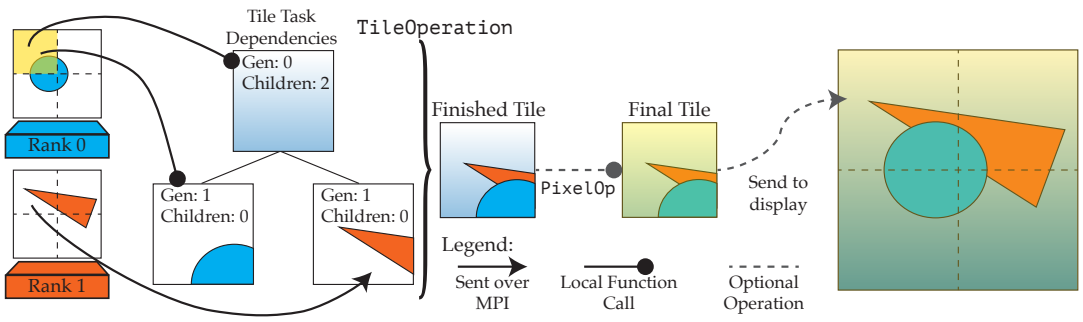


Fig. 4.2: An example of the DFB's tile processing pipeline in a data-parallel renderer. Dependencies are specified on the fly per tile and can be extended by child tiles. To compute the highlighted tile owned by rank 0, the owner sends a background color tile for generation 0, which specifies that two additional tiles will arrive in generation 1, potentially from different ranks. After receiving the full dependency tree, the tile operation produces the finished tile, which is tone-mapped by a pixel operation and sent to the display rank.

and passed on to any pixel operations. The final output tile is then converted to the display image format and sent to the display rank, if needed. The processing pipeline and messaging system run asynchronously on multiple threads, allowing users to overlap additional computation with that performed by the DFB. Although the DFB is presented in the context of distributed rendering, the task input tiles are not required to be produced by a renderer.

4.1.1.1 Tile Processing Pipeline

The DFB begins and ends processing synchronously, allowing applications processing multiple frames, i.e., a renderer, to ensure that tiles for different frames are processed in the right order. Before beginning a frame, the renderer specifies the tile operation to process the tiles it will produce. Each rank then renders some set of tiles based on the work distribution chosen by the renderer. As tiles are finished, they are handed to the DFB for processing by calling `setTile`. During the frame, the DFB will compute tile operations for the tiles owned by each rank in the background and send other tiles over the network to their owner. The frame is completed on each rank when the tiles it owns are finalized, and rendering is finished when all ranks have completed the frame. As each tile is processed independently in parallel, it is possible for some tiles to be finalized while others have yet to receive their first inputs.

To track the distributed tile ownership, the DFB instance on each rank stores a tile

descriptor (Listing 1) for each tile in the image. When `setTile` is called, the DFB looks up the descriptor for the tile and sends it to the owner using an asynchronous messaging layer (Section 4.1.1.2). If the owner is the calling rank itself, the tile is instead scheduled for processing locally.

For each tile owned by the rank, the DFB stores a concrete tile operation instance in the array of descriptors. The base structure for tile operations (Listing 1) stores a pointer to the local DFB instance and a Tile buffer to write the finished tile data to, along with optional accumulation and variance buffer tiles. The `finalPixels` buffer is used as scratch space to write the final tile to, before sending it to the display rank. Depending on the display image format specified (i.e., `RGBA8`, `RGBAF32`, or `NONE`), only a portion of the buffer may have valid data. The `NONE` format is unique, in that it indicates that the display rank should not receive or store the final pixels produced by the tile processing pipeline. This format is useful for driving large display walls, as demonstrated in Section 4.1.2.4.

To implement the corresponding tile operation for a rendering algorithm (e.g., sort-last

```
struct Tile {
    int generation;
    int children;
    region2i screenRegion;
    int accumulationID; // Sample pass for progressive refinement
    float color[4*TILE_SIZE*TILE_SIZE];
    float depth[TILE_SIZE*TILE_SIZE];
    float normal[3*TILE_SIZE*TILE_SIZE]; // Optional
    float albedo[3*TILE_SIZE*TILE_SIZE]; // Optional
};
struct TileDescriptor {
    virtual bool mine() { return false; }
    vec2i coords;
    size_t tileID, ownerRank;
};
struct TileOperation : TileDescriptor {
    bool mine() { return true; }
    virtual void newFrame() = 0;
    virtual void process(const Tile &tile) = 0;

    DistributedFrameBuffer *dfb;
    vec4f finalPixels[TILE_SIZE*TILE_SIZE];
    Tile finished, accumulation, variance;
};
```

Listing 1: The base structures for tiles and tile operations.

compositing) users extend the `TileOperation`, and specify their struct to be used by the DFB. Each time a tile is received by the DFB instance on the tile owner, the `process` function is called on the tile operation to execute the task. The `newFrame` function is called when a new frame begins to reset any per frame state.

Then all the dependencies of a tile have been received, the tile operation combines the inputs to produce a finished tile, which is then passed to the DFB. The local DFB instance runs any additional pixel operations on the finished tile and converts the final pixels to the display color format, outputting them to the `finalPixels` buffer. This buffer is then compressed and sent to the display rank. In addition to the `RGBA8` and `RGBAF32` display formats, the DFB also offers a `NONE` format, which is unique in that it indicates that the display rank should not receive or store the final pixels at all. A useful application of the `NONE` format to enable rendering to large display walls is discussed in Section 4.1.2.4.

4.1.1.1.1 Per tile task dependency trees. The `Tile` structure passed to `setTile` and routed over the network is shown in Listing 1. To construct the dependency tree, each rendered tile specifies itself as a member of some generation (a level in the tree), and as having some number of children in the following generation. The total number of tiles to expect in the next generation is the sum of all children specified in the previous one. Different ranks can contribute tiles with varying numbers of children for each generation, and can send child tiles for parents rendered by other ranks. There is no requirement that tiles are sent in order by generation, nor is a tile operation guaranteed to receive tiles in a fixed order. Tile operations with dependencies beyond a trivial single tile can be implemented by buffering received tiles in `process` to collect the complete dependency tree.

The interpretation and processing order of the dependency tree is left entirely to the tile operation. For example, the dependency tree could be used to represent a compositing tree, input to some filtering, or simply a set of pixels to average together. The creation of the dependency trees by the renderer and their processing by the tile operation are tightly coupled, and thus the two are seen together as a single distributed rendering algorithm. The flexibility of the tile operation and dependency trees allows the DFB to be used in a wide range of rendering applications (Section 4.1.2).

4.1.1.1.2 Pixel operations. Additional postprocessing, such as tone-mapping, can be performed by implementing a pixel operation (`PixelOp`). The pixel operation takes the single finished tile from the tile operation as input, and thus is not tied to the tile operation or renderer. The DFB runs the pixel operation on the tile owner after the tile operation is completed to distribute the work. In addition to image postprocessing, pixel operations can be used, for example, to re-route tiles to a display wall (Section 4.1.2.4).

4.1.1.2 Asynchronous Messaging Layer

An asynchronous point-to-point messaging layer built on top of MPI is used to overlap communication between nodes with computation. Objects that will send and receive messages register themselves with the messaging layer and specify a unique global identifier. Each registered object is seen as a global "distributed object," with an instance of the object on each rank that can be looked up by its global identifier. A message can be sent to the instance of an object on some rank by sending a message to the rank with the receiver set as the object's identifier.

The messaging layer runs on two threads: a thread that manages sending and receiving messages with MPI, and an inbox thread that takes received messages and passes them to the receiving object. Messages are sent by pushing them on to an outbox, which is consumed by the MPI thread. Nonblocking MPI calls are used to send, receive, prove, and test for message completion to avoid deadlocks between ranks. Received messages are pushed on to an inbox, which is consumed by the inbox thread. To hand a received message to the receiving object, the inbox thread looks up the receiver by its global ID in a hash table. Messages are compressed using Google's Snappy library [95] before enqueuing them to the outbox and decompressed on the inbox thread before being passed to the receiver.

The DFB initially also used the messaging layer to gather the final tiles to the display rank. However, unless the rendering workload is highly imbalanced, this approach generates a large burst of messages to the display, with little remaining rendering work to overlap with. This burst of messages also appeared to trigger an MPI performance issue on some implementations. As an optimization, the final tiles are instead written to a buffer, which is compressed and gathered to the display rank with a single `MPI_Gatherv` at the end of the frame.

4.1.2 Rendering with the Distributed FrameBuffer

A distributed rendering algorithm using the DFB consists of two components: a renderer, responsible for rendering tiles of the image, and a tile operation, which will combine the results of each rank's renderer. The following sections discuss distributed rendering algorithms built on the DFB, covering standard image-parallel (Section 4.1.2.1) and data-parallel (Section 4.1.2.2) renderers, along with extensions to these methods enabled by the DFB, specifically, dynamic load balancing (Section 4.1.2.1) and hybrid-parallel rendering (Section 4.1.2.3). Finally, Section 4.1.2.4 discusses an example of how pixel operations can be used to implement a high-performance display wall system.

4.1.2.1 Image-Parallel Rendering

An image-parallel renderer distributes the tile rendering work in some manner between the ranks such that each tile is rendered once (Fig. 4.3). This distribution can be a simple linear assignment, round-robin, or based on some runtime load balancing. The corresponding tile operation expects a single rendered tile as input. The DFB allows for a straightforward and elegant implementation of this renderer (Listing 2). Although image-parallel rendering is typically used for data-replicated rendering, where the data fit entirely in memory, it can also be applied to large out-of-core data, where data are paged on demand between ranks [65, 133], or off disk [293].

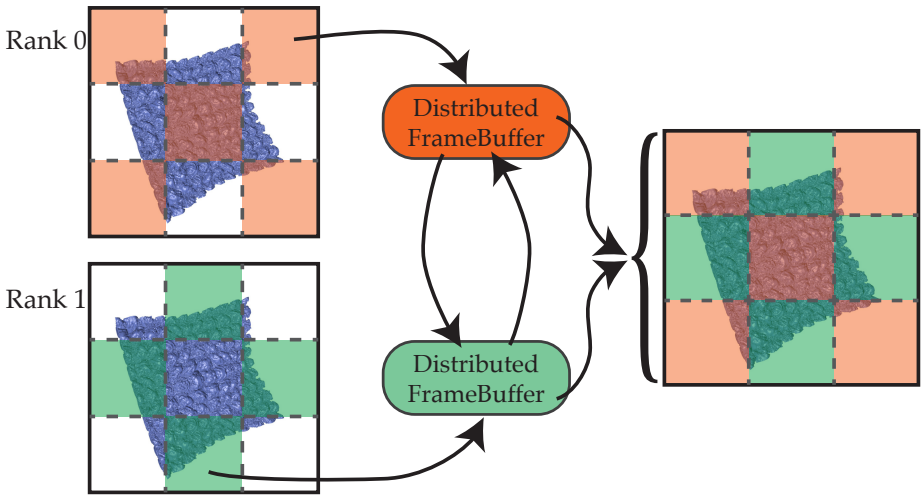


Fig. 4.3: An example of a simple image-parallel renderer using the DFB.

```
struct ImageParallel : TileOperation {
    void process(const Tile &tile) {
        // Omitted: copy data from the tile
        dfb->tileIsCompleted(this);
    }
};
void renderFrame(DFB *dfb) {
    dfb->begin();
    parallel_for (Tile &t : assignedTiles()) {
        renderTile(t);
        dfb->setTile(t);
    }
    dfb->end();
}
```

Listing 2: The tile operation and rendering loop for an image-parallel renderer using the DFB.

The work distribution chosen by the renderer is not tied to the DFB tile ownership, allowing the renderer to distribute work as desired. Although it is preferable that the tile owners render the tiles they own to reduce network traffic, this is not a requirement. This flexibility in work distribution can be used, for example, to implement dynamic load balancing. The `ImageParallel` tile operation can be extended to support receiving a varying number of tiles, and the renderer to assign each tile to multiple ranks. Each redundantly assigned tile uses a different random seed to generate camera rays, thereby computing a distinct set of samples. The rendered tiles are then averaged together by the tile operation, producing a finished tile equivalent to a higher sampling rate. This approach is especially useful for path tracing, as a high number of samples are required to produce a noise-free image. Tiles with higher variance can be assigned to additional ranks, adjusting the sampling rate dynamically. The tile assignment strategy chosen by the renderer can be based on, e.g., estimating the number of samples needed per tile by measuring image variance; the expected rendering cost of the tiles; or hardware performance, in a heterogeneous environment.

4.1.2.2 Data-Parallel Rendering

A standard sort-last data-parallel renderer decomposes the scene into a set of bricks, and assigns one brick per rank for rendering. Each rank renders its local data to produce a partial image. The partial images are combined using a sort-last compositing algorithm

to produce an image of the entire data set. A data-parallel renderer is implemented using the DFB by expressing the sort-last compositing operation as a tile operation, allowing the compositing work to take advantage of the DFB's asynchronous tile routing and processing to execute the compositing in parallel with local rendering. The benefits of this approach are two-fold: the per tile task dependencies allow minimizing compositing and communication work per tile, and overlapping compositing and rendering reduces the additional time spent compositing after rendering is finished.

To compute a per tile compositing dependency tree, each rank collects the bounds of the other ranks' data and projects them to the image (Fig. 4.4). Only those ranks whose data projects to some tile will render inputs for it. Each rank is responsible for specifying the dependency information for the tiles it owns (highlighted in yellow, Fig. 4.4). The tile owner will compute an additional "background" tile and set it as the sole member of generation 0. The background tile is filled with the background color or texture, and sets the number of ranks whose data project to the tile as the number of children.

The renderer (Listing 3) begins by determining the set of candidate tiles that it must either send a background tile for or render data to. The candidate tiles that the rank's

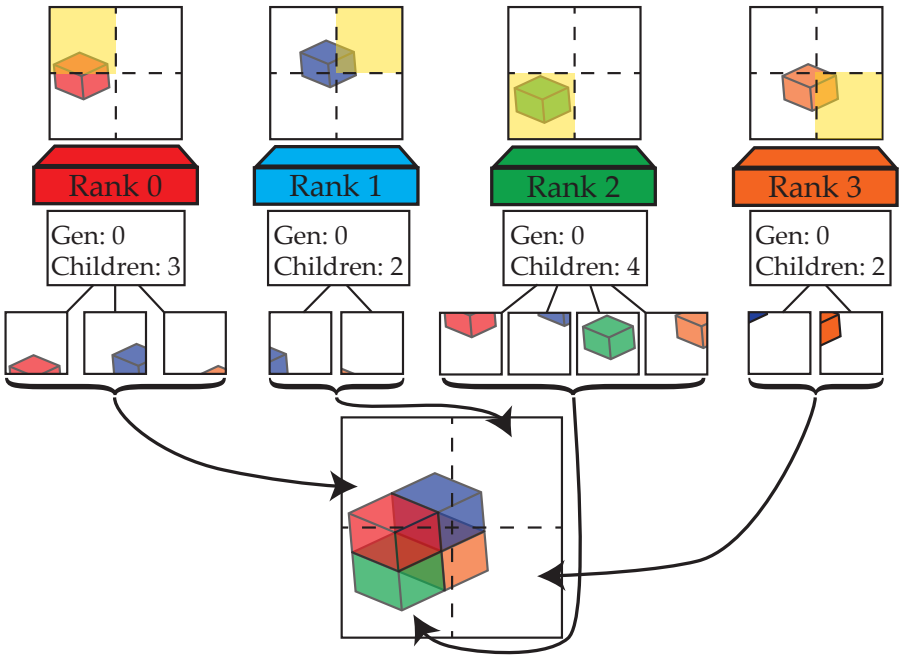


Fig. 4.4: Tile ownership and dependency trees for a data-parallel renderer using the DFB. Each rank owns its highlighted tile, and receives input tiles from ranks whose data projects to the tile. Compositing runs in parallel to local rendering, reducing overhead.

```
void renderFrame(Brick local, box3f allBounds[], DFB *dfb) {
    dfb->begin();
    /* The rank touches the tiles it owns and those touched by the
     * screen-space projection of its data
     */
    Tile tiles[] = array_union(dfb->ownedTiles(), dfb->touchedTiles(local));
    parallel_for (Tile &t : tiles) {
        bool intersected[] = intersectBounds(allBounds, t);
        if (dfb->tileOwner(t)) {
            fillBackground(t);
            t.generation = 0;
            t.children = numIntersected(intersected).
            dfb->setTile(t);
        }
        if (intersected[local]) {
            renderBrickForTile(t, local);
            t.generation = 1;
            t.children = 0;
            dfb->setTile(t);
        }
    }
    dfb->end();
}
```

Listing 3: The rendering loop for a standard data-parallel renderer.

local data may project to are found using a conservative screen-space AABB test, which is subsequently refined. For each candidate tile, the renderer computes an exact list of the ranks whose data touch the tile by ray tracing the bounding boxes. The number of intersected boxes is the number of generation 1 tiles to expect as input to the tree. If the rank's local data were intersected, it renders its data and sends a generation 1 tile. To allow for ghost zones and voxels, camera rays are clipped to the local bounds of the rank's data. As with the outer candidate tile loop, the inner rendering loop is parallelized over the pixels in a tile.

After receiving the entire dependency tree, the `AlphaBlend` tile operation (Listing 4) sorts the pixels by depth and blends them together to composite the tile. The tile fragment sorting is done per pixel, in contrast to the per rank sort used in standard approaches. Sorting per pixel allows for rendering effects like depth of field, side-by-side stereo, and dome projections. As the tile processing is done in parallel, this sorting was not found to cause a bottleneck. In the case that a rank-order sort produces a correct image, the dependency tree can be constructed as a list instead of a single-level tree with tiles ordered back-to-front by

```
struct AlphaBlend : TileOperation {
    Tile bufferedTiles[];
    int currentGen, missing, nextChildren;
    void newFrame() {
        currentGen = 0;
        missing = 1; // Expect a generation 0 tile to start
        nextChildren = 0;
    }
    void process(const Tile &tile) {
        bufferedTiles.append(tile);
        if (tile.generation == currentGen) {
            --missing;
            nextChildren += tile.children;
            checkTreeComplete();
        }
        if (!missing) {
            sortAndBlend(bufferedTiles);
            dfb->tileIsCompleted(this);
            bufferedTiles = {}
        }
    }
    // Check receipt of all children from all generations,
    // advancing currentGen as generations are completed.
    void checkTreeComplete() { /* omitted for brevity */ }
}
```

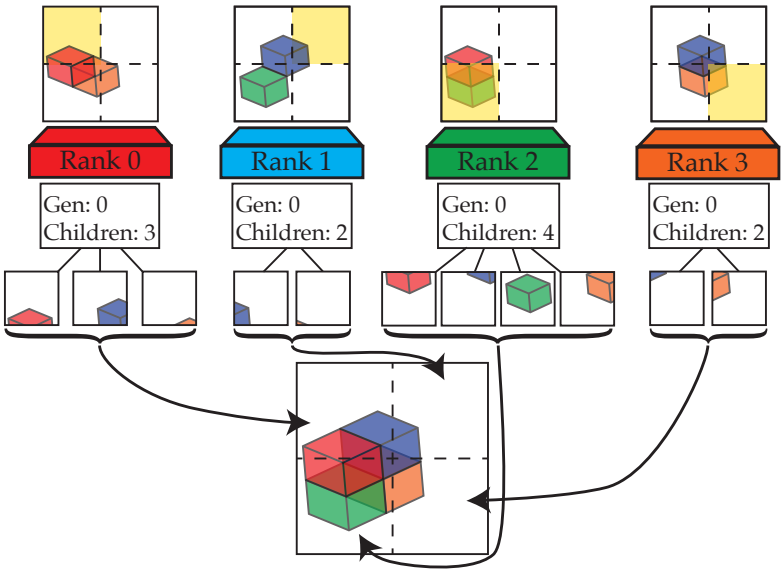
Listing 4: The sort-last compositing tile operation used by the data- and hybrid-parallel renderers. It first collects the dependency tree, and then sorts and blends the pixels to produce the composited tile.

generation. Finally, although the data-parallel renderer has been discussed in the context of rendering a single brick of data per rank, it trivially supports multiple bricks per rank, allowing for finer-grained work distributions or more complex data distributions.

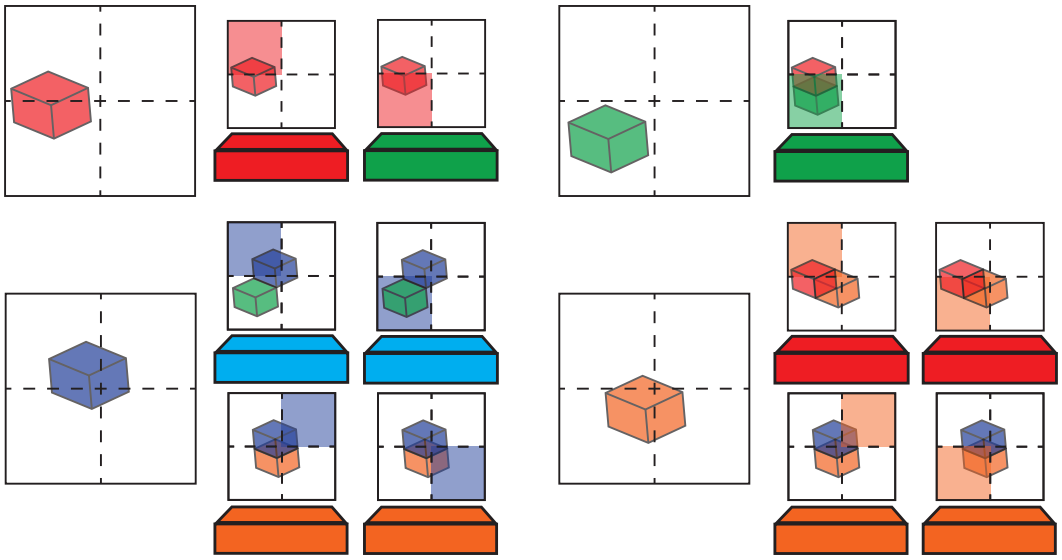
4.1.2.3 Rendering Hybrid Data Distributions

A data-parallel renderer that statically assigns each brick of data to a single rank is susceptible to load imbalance, coming from factors such as the data distribution, transfer function, or camera position. Such uneven distributions of work can lead to a few ranks, or even a single rank, becoming a bottleneck, and thus underutilizing the system. To better distribute the workload, the same brick of data can be assigned to multiple ranks, with each rank potentially assigned multiple bricks (Fig. 4.5). A more fine-grained work distribution can also be created by subdividing the volume into more bricks than ranks. Each rank is responsible for rendering a subset of the tiles the bricks it has project to, thereby dividing

the rendering workload for each brick among the ranks. Although partially duplicating data across ranks increases the memory requirements of the renderer, additional memory is often available given the number of compute nodes used to achieve an interactive frame rate.



(a) Tile ownership and inputs to tile operations



(b) Rendering work assignment for shared data regions

Fig. 4.5: Sort-last hybrid-parallel rendering. (a) Each rank stores two data regions and (b) shares the rendering work for each region among those ranks sharing the region. The compositing tasks remain the same as the standard data-parallel case, but can achieve better load balance.

Rendering such a configuration with a standard compositing approach is either difficult or not possible, as the compositing tree and blending order is set for the entire framebuffer by sorting the ranks [194]. However, the DFB's per tile task dependency trees allow renderers to change which ranks contribute tiles for each image tile independently. This flexibility enables a direct extension of the data-parallel renderer discussed previously into a hybrid-parallel renderer, which balances image and data parallelism to achieve better load balance.

The concept of a "tile-brick owner" is introduced to develop the hybrid-parallel extension. Given a data set partitioned into a set of bricks and distributed among the ranks with some level of replication, the renderer must select a unique rank among those sharing a brick to render it for each image tile. The rank chosen to render the brick for the tile is referred to as the "tile-brick owner." Thus, the hybrid-parallel renderer can be implemented by taking the data-parallel renderer and modifying it so that a rank will render a brick for a tile if the brick projects to the tile and the rank is the tile-brick owner (Listing 5). The task dependency tree and tile operation are the same as the data-parallel renderer; the only difference is which rank renders the generation 1 tile for a given brick and image tile.

The current implementation of the hybrid-parallel renderer uses a round-robin assignment to select tile-brick ownership; however, this is not a requirement of the DFB. A more advanced renderer could assign tile-brick ownership based on some load balancing strategy (e.g., [88]), or adapt the brick assignment based on load imbalance measured in the previous frame. The strategies discussed for image-parallel load balancing and work subdivision in Section 4.1.2.1 are also applicable to the hybrid-parallel renderer. For example, two ranks sharing a brick could each compute half of the camera rays per pixel, and average them together in the tile operation to produce a higher quality image.

The hybrid-parallel renderer supports the entire spectrum of image- and data-parallel rendering: given a single brick per rank it is equivalent to the data-parallel renderer; given the same data on all ranks it is equivalent to the image-parallel renderer; given a partially replicated set of data, or a mix of fully replicated and distributed data, it falls in between.

```
void renderFrame(Brick local[], box3f allBounds[], DFB *dfb) {
    dfb->begin();
    Tile tiles[] = array_union(dfb->ownedTiles(), dfb->allTouchedTiles(local));
    parallel_for (Tile &t : tiles) {
        bool intersected[] = intersectBounds(allBounds, t);
        if (dfb->tileOwner(t)) {
            // Listing 3, lines 9-12
        }
        parallel_for (Brick &b : local) {
            if (tileBrickOwner(b, t) && intersected[b]) {
                // Listing 3, lines 15-18
            }
        }
    }
    dfb->end();
}
```

Listing 5: The rendering loop of the hybrid-parallel renderer. The DFB allows for an elegant extension of the data-parallel renderer to support partially replicated data for better load balancing.

4.1.2.4 Display Walls

The DFB can be used to implement a high-performance display wall rendering system, as demonstrated by Han et al. [109], by using a pixel operation to send tiles directly to the displays (Fig. 4.6). Tiles will be sent in parallel as they are finished on the tile owner directly to the displays, achieving good utilization of a fully interconnected network. Moreover, when rendering with the `NONE` image format, the image will not be gathered to the master rank, avoiding a large amount of network communication and a common bottleneck. As pixel operations are not tied to the rendering algorithm or tile operation, this method can be used to drive a display wall with any of the presented renderers.

4.1.3 Implementation

Although the DFB is applicable to any tile-based rendering algorithm, it is implemented and evaluated within the OSPRay ray tracing framework [291]. OSPRay provides a range of built in volume and geometric primitives used in scientific visualization, advanced shading effects, and achieves interactive rendering on typical workstations and laptops. OSPRay implements its core rendering code for different execution backends through "Devices." For example, local rendering is done by using the `LocalDevice`. To achieve interactive ray tracing performance on CPUs, OSPRay builds on top of Embree [294], the Intel SPMD

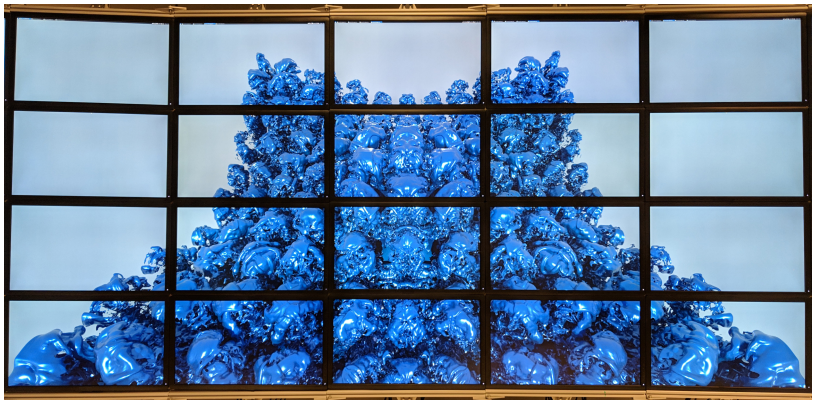


Fig. 4.6: A prototype display wall system using DFB pixel operations to send tiles in parallel from an image-parallel path tracer.

Program Compiler (ISPC) [224], and Intel's Threading Building Blocks (TBB).

Embree is a high-performance kernel framework for CPU ray tracing, and provides a set of low-level kernels for building and traversing ray tracing data structures that are highly optimized for modern CPU architectures. ISPC is a single program multiple data (SPMD) compiler that vectorizes a scalar program by mapping different instances of the program to the CPU's vector lanes, thereby executing them in parallel. TBB provides a set of parallel programming primitives for writing high-performance multithreaded code, similar to OpenMP.

Prior to this work, OSPRay provided a `MPIOffloadDevice` that provided support for image-parallel rendering. The performance of this device has been significantly improved through this work, although it is exposed to users in the same manner as before. Users can continue to run existing OSPRay applications with `mpirun` and pass the `--osp:mpi` argument to the application, and OSPRay will replicate the scene data across a cluster and render it image-parallel using the rendering algorithms described in Section 4.1.2.1 to provide interactive high-quality rendering (Fig. 4.7).

The OSPRay API was originally designed for a single application rank passing its data to OSPRay. Although OSPRay may offload the data in some way to other ranks, this is done without the application's awareness. This API works well for applications that do not need to specify the data distribution; however, it is not applicable to those that do, such as ParaView and VisIt. Maintaining an API that is familiar to users while extending it to a data-distributed scenario poses some challenges. Furthermore, the API should

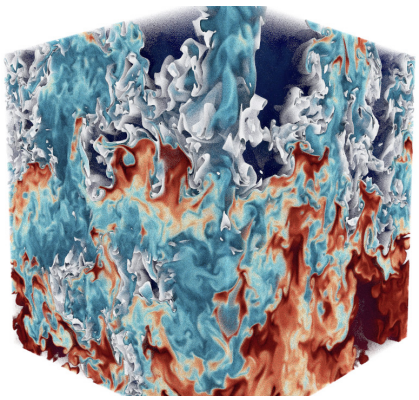


Fig. 4.7: A visualization of the 1024^3 Miranda [56] data set rendered interactively using OSPRay's volumetric path tracer, with support for scattering and volumetric lighting effects.

seamlessly support existing OSPRay modules, which have added new geometries [110, 283, 292, 297], volumes [230, 289, 295], and renderers [274] in a data-distributed setting. To enable applications to leverage the distributed rendering algorithms provided by the DFB in OSPRay, this work introduces a new distributed API and distributed device in OSPRay through the `MPIDistributedDevice`.

4.1.3.1 Designing a Flexible Data-Distributed API

To make the rendering capabilities provided by the DFB available to distributed visualization applications, the `MPIDistributedDevice` extends OSPRay with an API for rendering distributed data. The bulk of the distributed API is similar to local rendering in OSPRay, and should appear familiar to existing users. To use the distributed API, applications can load the MPI module and select the `MPIDistributedDevice`. Each rank can then make independent API calls to set up its local geometries and volumes, after which the distributed scene can be rendered collectively using the algorithms described in Section 4.1.2. As the DFB's data-parallel renderer needs only the bounding boxes of each rank's portion of the data, the distributed API is nearly identical to local rendering, and supports existing user geometry and volume modules without changes.

The distributed API makes a distinction between "distributed" and "local" objects. Distributed objects track the global rendering configuration or scene layout (e.g., the framebuffer, renderer, world, and futures), and they must be created and committed collectively by the ranks to ensure the objects are synchronized properly. With the exception of the list of instances and regions on the world, each rank should set the same parameter

values on distributed objects. Local objects represent local data, geometries, and volumes and are created independently by the ranks. When using the distributed API, only rank 0 will be able to map the framebuffer to access the rendered image.

For rendering data- or hybrid-distributed configurations, applications must use the `mpiRaycast` renderer, which implements the rendering algorithms discussed above. The distributed API also adds an optional "regions" parameter to the `OSPWorld`, which is used to specify one or more bounding boxes that bound the local data owned by the rank to configure the compositor. The region list can be used to clip ghost zones, or to partition nonconvex local data distributions into sets of convex regions that can be composited by OSPRay's sort-last compositor. If no regions are specified, the union of the bounds of the local geometries and volumes is implicitly set as the rank's region bounds. It is also valid for some ranks to have no data in the scene, and thus no regions or instances in with world.

Finally, it is also possible to select only a subset of ranks to participate in rendering by specifying the communicator that OSPRay should use. By default, OSPRay will use all ranks in the MPI world; however, OSPRay uses an MPI + threads model for parallelism, which, when integrating with an MPI-only simulation, may lead to oversubscribing the nodes. Instead, the application could create a subcommunicator with one rank per node and set this as OSPRay's communicator. The data from other ranks on the node can then be gathered to this OSPRay rank using shared memory.

4.1.3.1.1 Configuring the scene distribution using regions. The optional regions parameter on the `OSPWorld` allows each rank to set one or more bounding boxes that contain the data it owns for rendering. Arbitrary geometric and volumetric data can be placed within the regions, including those implemented by user modules. Each region corresponds to a unique piece of data that is owned the rank. Each rank can have zero or more such regions, and can share ownership of regions with other ranks. When rendering, only data contained within the local region bounds on a rank will be visible to camera rays, although data outside are available for volume interpolation and secondary rays. The regions allow applications to provide ghost zones for interpolation and AO, render nonconvex and nondisjoint data distributions (Section 4.1.3.1.2), or share data between ranks for hybrid-distributed and data-replicated rendering (Section 4.1.3.1.3).

4.1.3.1.2 Using regions to clip ghost zones and ensure convexity. Ghost zones, or sometimes "halos," are used in distributed rendering to ensure the image produced by the distributed renderer matches that which would be produced by a renderer running on one rank with the entire data set. For example, an extra layer of voxels is stored around the volume subpiece owned by the rank to avoid interpolation artifacts between neighboring ranks. Similarly, when rendering secondary effects such as ambient occlusion, additional ghost geometries are used to prevent bright spots appearing at rank boundaries where geometry would otherwise be missing. Regions are used only to clip camera rays, allowing secondary rays to still intersect any ghost geometry that may exist outside the region bounds.

The regions can also be leveraged to allow rendering nonconvex or nondisjoint data distributions that would otherwise require redistributing the data when using other sort-last compositors. OSPRay, as with other sort-last compositors, requires the partial images being composited to correspond to convex, disjoint bricks of data. Without this requirement, sorting and ordering the partial images to produce a correct final image may not be possible. However, OSPRay's distributed API allows each rank to specify one or more regions that it owns. Each such region is treated as if it were owned by a unique rank, and rendered to produce an independent set of partial image tiles that are given to the compositor. Nonconvex data-distributions, such as those potentially arising in adaptive mesh refinement simulations [3, 19, 20, 38, 55, 184, 207], can be virtually partitioned into a set of disjoint bricks that are suitable for sort-last compositing (Fig. 4.8). This partitioning takes place only when clipping rays against the region bounds in the renderer, and requires no movement or adjustment of the underlying data.

4.1.3.1.3 Rendering of partially and fully replicated scenes. The regions list can also be used to instruct OSPRay that two ranks are sharing one or more regions, by specifying the same region bounds on each rank. If additional memory is available to duplicate some or all of the scene data, ranks can share regions for better load balancing. During the region exchange step when the world is committed, these shared regions are found and merged in the global list of regions. Each unique region in the list corresponds to a unique piece of data that must be rendered and composited independently. When multiple ranks are found to share the same region, they switch to the hybrid-distributed

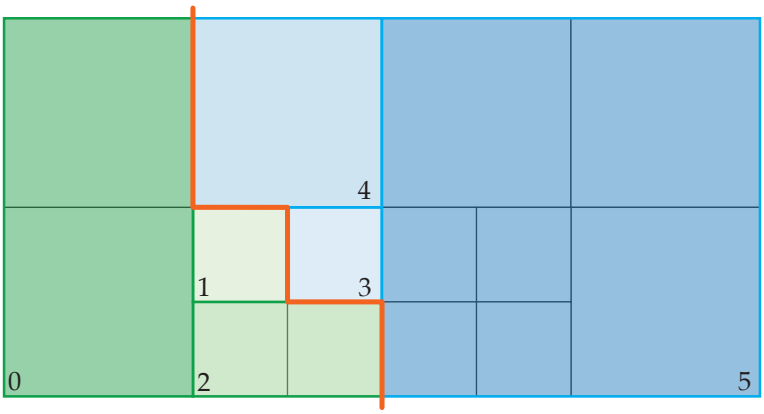


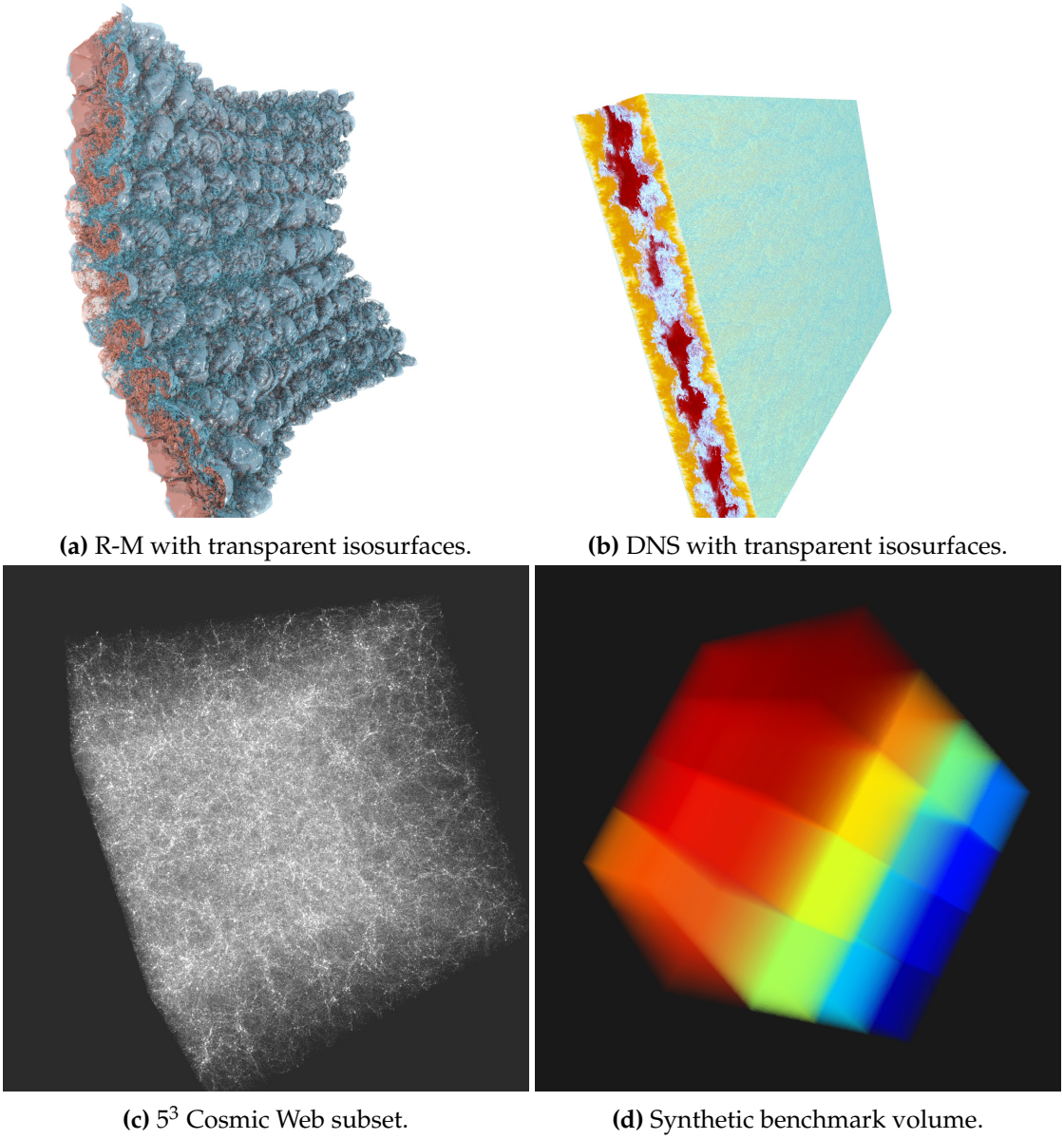
Fig. 4.8: An example of virtually partitioning an Octree AMR data set using regions. The data are distributed among two ranks along the orange line and virtually partitioned using regions to create a set of disjoint bricks that can be composited by OSPRay's DFB. The three regions created by each rank only virtually partition the data, leaving the underlying data unchanged.

rendering algorithm discussed in Section 4.1.2.3. For example, if a region is found to be especially expensive to render, that region can be duplicated onto other ranks for load balancing, leaving the rest of the data as is.

If enough memory is available to fit the entire scene on each rank, the same data and regions can be specified on all ranks. The distributed device can detect that all ranks specified the same region, and switch to use an image-parallel rendering algorithm (Section 4.1.2.1), allowing the use of any of the renderers from OSPRay's local device. For example, the path tracer can be used to provide high-quality secondary illumination effects (Fig. 4.7).

4.1.4 Evaluation

To evaluate the performance of the DFB, a set of benchmarks is run on the rendering algorithms described in Section 4.1.2, using their implementations in OSPRay. The benchmarks are run on two HPC systems, the Texas Advanced Computing Center's Stampede2 [263], and Argonne National Laboratory's Theta [113], on a range of typical image- and data-parallel rendering use cases (Fig. 4.9). To provide a direct comparison point against standard sort-last compositors, a set of benchmarks are performed that compare sort-last compositing using the DFB against IceT for a typical data-parallel use case. To measure performance as the rendering workload varies, the benchmarks are taken while



(a) R-M with transparent isosurfaces.

(b) DNS with transparent isosurfaces.

(c) 5^3 Cosmic Web subset.

(d) Synthetic benchmark volume.

Fig. 4.9: The data sets used in the benchmarks. (a) Two transparent isosurfaces on the Richtmyer-Meshkov [54], 516M triangles total. (b) A combined visualization of the 451GB single-precision DNS [167] with two transparent isosurfaces, 5.43B triangles total. (c) A 5^3 subset of the 8^3 Cosmic Web [131], 7.08B particles rendered as transparent spheres. (d) The generated volume data set used in the compositing benchmarks, shown for 64 nodes. Each node has a single 64^3 brick of data.

rendering a rotation around the data set. Unless otherwise stated, the plots below show the median performance over this camera rotation, with the median absolute deviation shown as error bars. These measures are more robust to outliers, giving some robustness against influence from other jobs on the system. All benchmarks are run with one rank per node, as OSPRay uses threads on a node for parallelism.

Stampede2 and Theta consist of 4200 and 4392 Intel Xeon Phi KNL processors respectively. Stampede2 uses the 7250 model with 68 cores, and Theta uses the 7230 model with 64 cores. Stampede2 contains an additional partition of 1736 dual-socket Intel Xeon Phi Platinum 8160 SKX nodes. Although the KNL nodes of both machines are similar, the network interconnects differ significantly, which can affect the performance of communication in the DFB. Stampede2 employs an Intel Omni-Path network in a fat-tree topology, and Theta uses a Cray Aries network with a three-level Dragonfly topology.

4.1.4.1 Image-Parallel Rendering Performance

The image-parallel rendering algorithm using is evaluated through a strong scaling benchmark performed using OSPRay's scientific visualization renderer. The Richtmyer-Meshkov isosurface test data set is created by extracting two isosurfaces using VTK, which are rendered with transparency and ambient occlusion (Fig. 4.9a). The strong scaling benchmark is run on the Stampede2 SKX nodes at two image resolutions (Fig. 4.10). Although the renderer begins to drop off from the ideal scaling trend as the local work per node decreases, this could potentially be addressed by employing the work-subdivision and load balancing strategies discussed in Section 4.1.2.1.

4.1.4.2 Data-Parallel Rendering Performance

To study the scalability of the DFB when applied to the standard data-parallel rendering algorithm in Section 4.1.2.2, a set of strong scaling benchmarks is run using two large-scale data sets on Stampede2 and Theta. On Stampede2, the benchmark renders a combined visualization of the DNS with transparent isosurfaces (Fig. 4.9b), and on Theta the benchmark renders the 5^3 Cosmic Web subset (Fig. 4.9c). The results of these benchmarks demonstrate that data-parallel rendering algorithms built on the DFB are able to provide interactive frame rates for these challenging scenes and scale up performance with more compute.

Good scaling is observed on the Cosmic web from 32 to 64 nodes (Fig. 4.11). Although

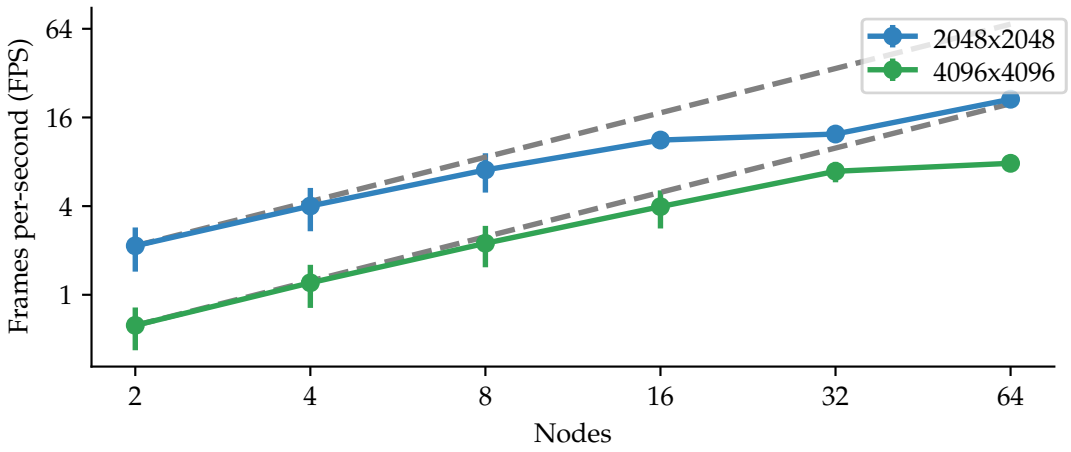


Fig. 4.10: Image-parallel strong-scaling on the R-M transparent isosurfaces data set on *Stampede2* SKX nodes. The image-parallel renderer using the DFB scales to provide interactive rendering of expensive, high-resolution scenes.

performance begins to trail off the ideal trend beyond 128 nodes, absolute rendering performance remains interactive. The DNS benchmarks exhibit near ideal scaling from 16 to 32 nodes (Fig. 4.12a); however, little performance improvement is observed when scaling from 32 to 64 nodes, although some improvement is achieved when stepping from 64 to 128 nodes. To find the cause of the bottleneck at 64 nodes, a breakdown of time spent rendering the rank's local data and the compositing overhead incurred by the DFB is shown in Fig. 4.12b. Compositing overhead refers to the additional time the compositor takes to complete the image, after the slowest local rendering task has completed [101]. The performance breakdown shows that the bottleneck at 64 nodes is caused by the local rendering task not scaling, which could be addressed by employing a hybrid data distribution or the work-splitting techniques discussed previously (Section 4.1.2.3).

4.1.4.3 Compositing Performance Comparison with IceT

To perform a direct comparison with IceT for data-parallel rendering, the benchmark uses a synthetic data set (Fig. 4.9d) that incurs little local rendering work on each rank. Furthermore, the DFB's data-parallel renderer is modified to support compositing with IceT to ensure both methods perform the same local rendering work. The IceT renderer follows the same code-path as the data-parallel renderer to render its assigned brick of data, and then hands the framebuffer off to IceT for compositing. In preliminary tests, IceT's

automatic compositing algorithm selection was found to give the best performance, and this mode is used throughout the benchmarks.

In terms of overall scalability and performance, the DFB scales better then, or at least similar to, IceT, while achieving better absolute rendering performance (Figs. 4.13a, 4.13c and 4.13e). When comparing timing breakdowns (Figs. 4.13b, 4.13d and 4.13f) it is observed that, as expected, the local rendering times are similar, and the performance difference is due to the differing compositing overhead of the two methods. It is important to note that some of the absolute difference in overhead is due to IceT's synchronous design, which makes it unable to overlap compositing with rendering. To account for this difference, one can consider a hypothetical IceT implementation that does overlap compositing and rendering by subtracting the local rendering time from the compositing overhead. However, even when compared to this hypothetical implementation, the DFB still achieves similar or superior compositing performance. Furthermore, when comparing the scaling trends of the two approaches, the DFB scales similar to, or better than, IceT. Although a rigorous comparison is difficult due to the different HPC systems used, the DFB follows similar scaling trends as Grosset et al.'s DSRB [101], while providing greater flexibility.

Finally, the portability of the DFB can be evaluated by comparing the KNL runs on Stampede2 (Figs. 4.13c and 4.13d) and Theta (Figs. 4.13a and 4.13b). The slightly different KNLs on each system will have a minor effect on performance; however, any significant dif-

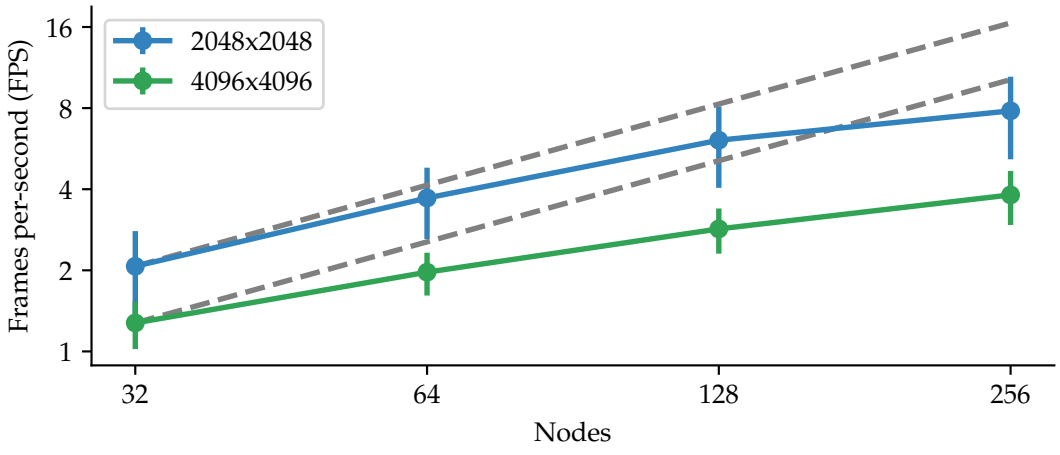
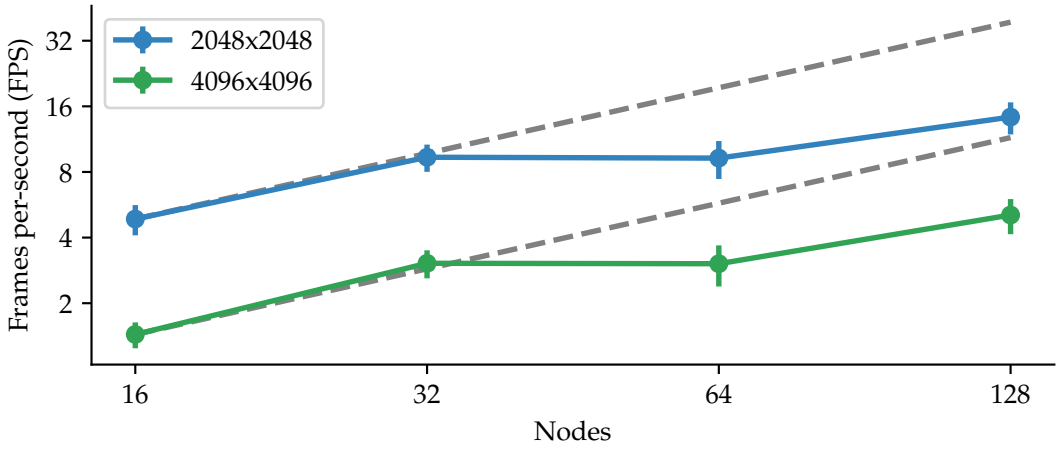
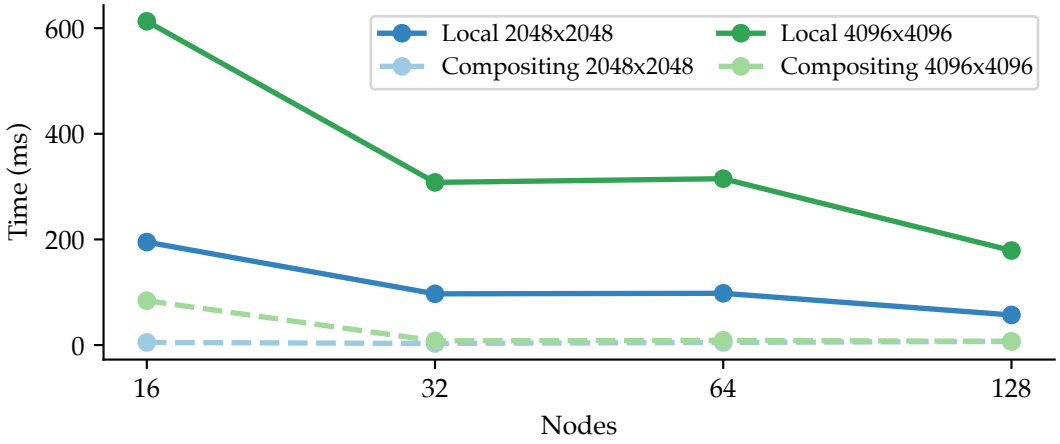


Fig. 4.11: Data-parallel strong-scaling on the Cosmic Web data set on Theta. Nearly ideal scaling is observed at moderate image sizes and node counts, with somewhat poorer scaling at very high resolutions.



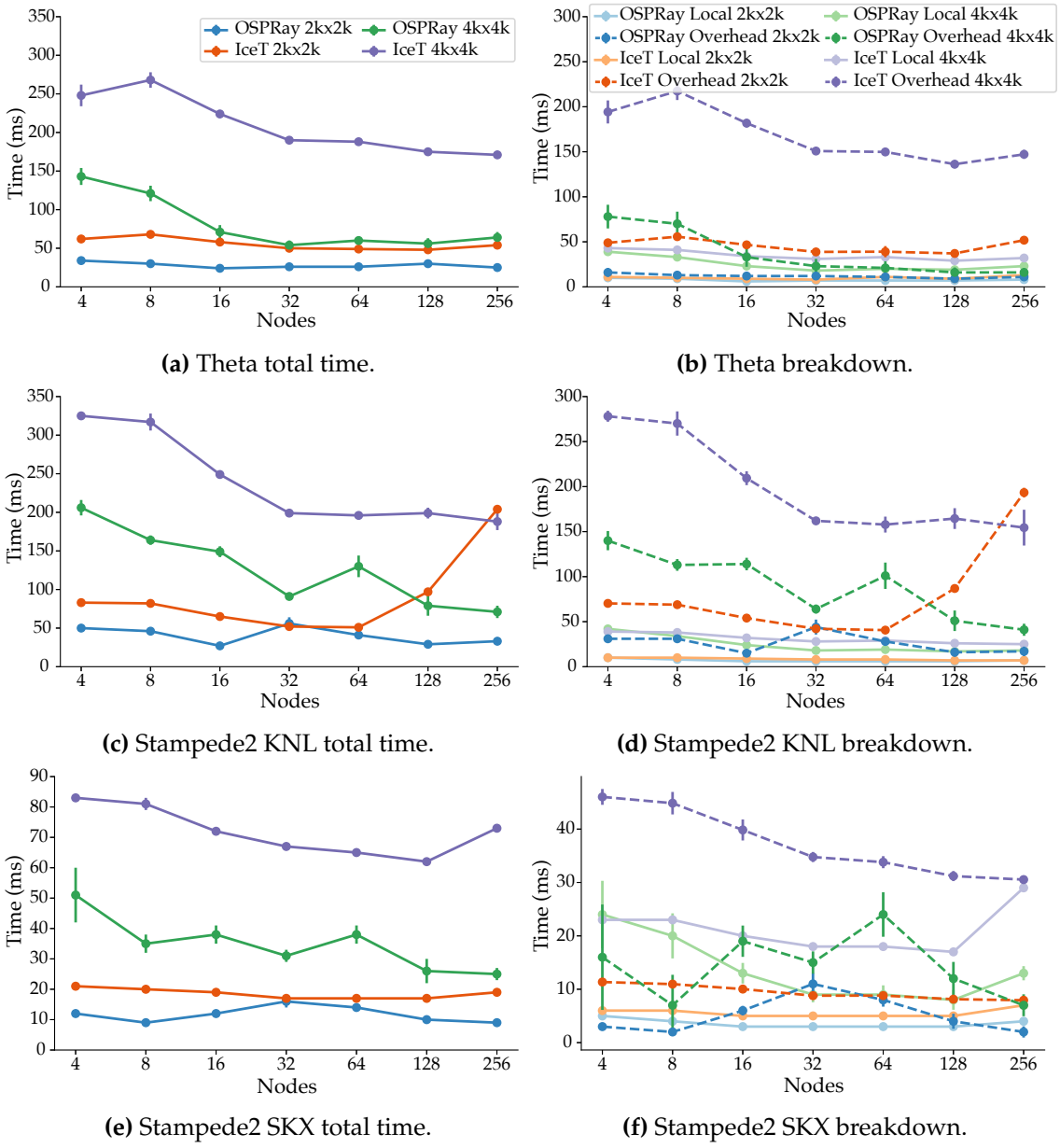
(a) Overall rendering performance.



(b) Timing breakdown of local rendering and compositing overhead.

Fig. 4.12: Data-parallel strong-scaling on the DNS with isosurfaces on Stampede2 KNLs. The lack of scaling from 32 to 64 nodes is attributable to a poor local work distribution (b), which can be partially addressed by using the hybrid-parallel renderer built on the DFB.

ferences are attributable to the differing network architectures and job placement strategies. The benchmarks on Stampede2 exhibit a rather bumpy scaling trend where, depending on the image size, a temporary decrease in the compositing performance is observed at certain node counts. The benchmarks on Theta follow a smoother trend, with better absolute compositing performance. By default, the DFB compresses all messages; however, disabling compression was found to provide better performance on Theta, whereas the runs on Stampede2 encountered significant MPI messaging performance issues at 16 nodes and up without compression. Thus, compression is left as an option to users that is enabled by



(a) Theta total time.

(b) Theta breakdown.

(c) Stampede2 KNL total time.

(d) Stampede2 KNL breakdown.

(e) Stampede2 SKX total time.

(f) Stampede2 SKX breakdown.

Fig. 4.13: Compositing benchmark performance comparison of the DFB and IceT on the synthetic data set. The DFB achieves better, or at least similar, scaling as IceT, while providing faster absolute rendering times. In the timing breakdowns (d-f), This difference is primarily due to the DFB achieving a significant reduction in compositing overhead.

default at 16 nodes. In the benchmarks compression is disabled on Theta and enabled at 16 nodes and up on Stampede2. IceT uses a custom image compression method, which is not easily disabled.

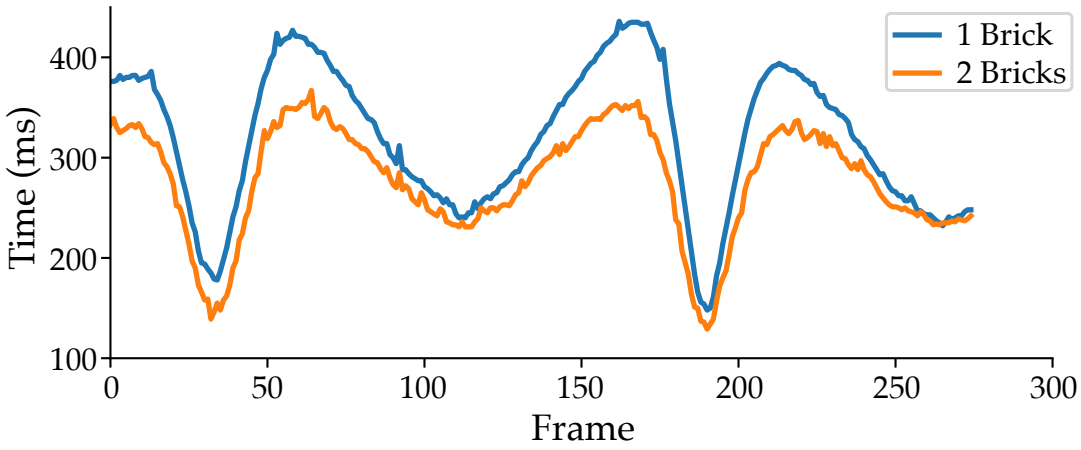
4.1.4.4 Hybrid Data Distribution Rendering Performance

The impact of partial data replication on load balance can be observed by studying the per frame overall time on the DNS with isosurfaces data set on Stampede2 (Fig. 4.14). The volume is partitioned into as many bricks as there are ranks, with bricks redundantly assigned to ranks based on the available memory capacity. When using 64 KNLs, there is enough memory to store two bricks per rank, with 128 KNLs it is possible to store up to four. The rendering work for each brick will be distributed among two or four ranks, respectively. The redundant bricks are distributed using a simple round-robin assignment. A brick distribution based on, e.g., some space filling curve or runtime tuning, could provide additional improvement.

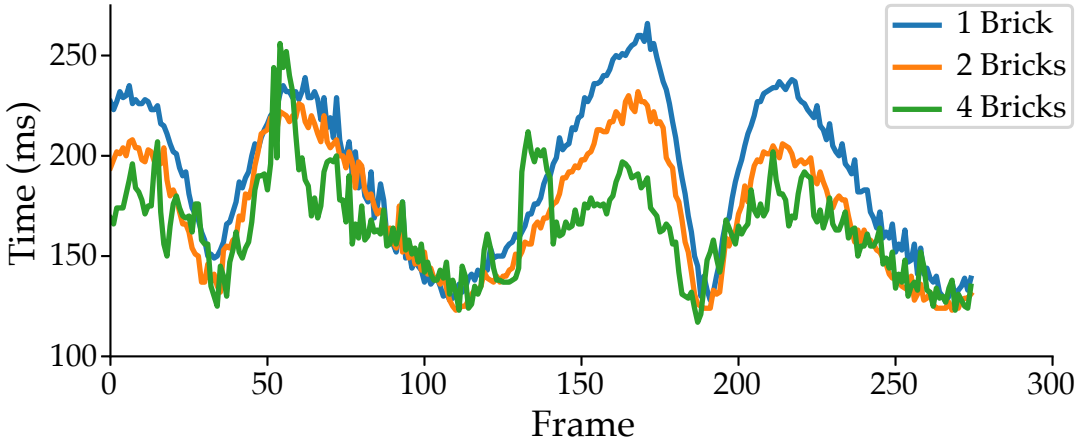
In both the 64 and 128 node runs, the two brick per node configuration provides a consistent improvement over no replication. This improvement is more pronounced for camera positions with greater load imbalance. There are larger fluctuations in rendering performance when storing four bricks per node, although at times the four-brick configuration does outperform the two-brick one. These larger fluctuations could be due to increased memory traffic, which is alleviated as data are cached in the KNL MCDRAM. This theory is further supported by the sharp spikes in performance, when new data must be fetched from RAM.

4.2 A Virtual Reality Visualization Tool for Neuron Tracing

Brain function emerges from the coordinated activity of billions of interconnected neurons that form dense neural circuits. A central goal of neuroscience is to understand how these circuits' computations relate to behavior. The field of connectomics is founded on the principle that understanding the precise wiring of these circuits, i.e., the location of neurons and the connections between them, is crucial to comprehending brain function at a mechanistic level. More insight into the fundamental connectivity within the brain also has the potential to lead to breakthroughs in the understanding of brain diseases and open new



(a) Per frame render time on 64 Stampede2 KNLs, at 4096 × 4096



(b) Per frame render time on 128 Stampede2 KNLs, at 4096 × 4096

Fig. 4.14: Improving load balancing on the DNS with isosurfaces with partial data-replication in the hybrid-parallel renderer. Sharing rendering between two nodes (two bricks per node) gives a consistent improvement, between four tends to give further improvement.

avenues for treatment.

However, obtaining a comprehensive wiring diagram of even relatively small and simple mammalian brains, such as that of a mouse, is a massive undertaking. Similar projects in species with larger brains that are evolutionarily closer to humans, such as nonhuman primates (NHPs), take more time and are more complex. To date, the only species whose nervous system has been completely mapped is the nematode Caenorhabditis elegans [301], which is comprised of only 302 neurons. Currently, the majority of connectome efforts are focused on mapping the mouse brain [29, 35]. However, with recent advances in high-

resolution tissue labeling [181], optical tissue clearing [50,315], and deep tissue imaging [66], mapping the NHP brain at mesoscopic scale is becoming feasible. One major impediment to mapping the NHP brain is the time-consuming, laborious effort of manually tracing labeled neuronal connections through the brain.

A typical neuron imaging and tracing workflow proceeds as follows. First, neurons and their processes are labeled using neuroanatomical tracing methods. Modern approaches to labeling neurons in large brains involve the use of viral vectors carrying the genes for fluorescent proteins [181]. These vectors are injected into the brain to induce expression of these genes within neurons, labeling them at high resolution. Current approaches in connectomics then render the brain optically transparent using clearing techniques such as CLARITY [50], PACT [315], or SWITCH [198]. Imaging labeled neurons through brain tissue, either in brain slices or through whole brains or blocks, produces multiple stacks of images ranging in size from gigabytes to terabytes. Finally, neurons are traced on these 2D image stacks to extract the desired neuronal structures. Depending on the analysis being performed, these structures can be used in simulations or overlaid onto functional maps of the brain, in order to understand the connectivity between brain regions or cells within these regions.

Automated methods for neuron reconstruction continue to improve, with recent works leveraging machine learning techniques [9,261]. However, neuroscientists often find the results of these algorithms unsatisfactory [174]. Thus, tracing neurons remains primarily a manual task. Meijering [190] noted that data quality was the primary reason these algorithms fail in practice, as the current state-of-the-art methods provide error-free results only in highly optimal conditions. For a full review and comparison of recent methods, see the surveys by Magliaro et al. [185] and Acciai et al. [2]. Many labs, such as the Angelucci lab, employ several trained undergraduate students who are responsible for the bulk of the tracing work. Tracing is done on a desktop computer using NeuroLucida [188]. When working on image stacks using this software, the user scrolls through the stack and clicks to mark points along the neuron or to create branches. However, some branches change depth rapidly, cross in complex ways, or have gaps due to imperfections in the labeling or imaging process, making the structure difficult to resolve.

Beyond the mechanics of tracing, another challenge is that microscopy technology

is rapidly outpacing the supporting tools in terms of raw data size. State-of-the-art microscopes regularly produce terabytes worth of images, yet few existing tools are capable of handling data at this scale. Notably, the TeraFly [34] plugin for Vaa3D [220] supports paging in hierarchical volume data to explore large data sets. Other tools are often limited by the RAM capacity of the system.

This sections presents a collaborative effort undertaken with trained neuroanatomists to develop a VR tool for neuron tracing using off the shelf consumer VR systems. The VRNT demonstrates that, when combined with state of the art data management and visualization techniques, the immersive and intuitive interface provided by consumer VR can provide a compelling environment to enhance the workflow of connectomics researchers. The following sections detail the design process (Section 4.2.1), the final design and technical aspects of the neuron tracing tool created through this process (Section 4.2.2), and a pilot study conducted to evaluate its effectiveness (Section 4.2.3).

4.2.1 Design Process

Independent of this work on neuron tracing in VR described in this section, a technology probe was developed (Section 4.2.1.1) with the goal of investigating consumer-grade VR for scientific visualization. One of the data sets used while testing the technology probe was a large microscopy scan acquired by the Angelucci lab, which was down-sampled to fit on the GPU. Positive feedback from an incidental demonstration of this probe prompted further exploration of into using VR for neuron tracing. The VR neuron tracing tool was developed in close collaboration with expert neuroanatomists from the Angelucci lab in an open-ended, iterative process, influenced by the nine-stage framework of Sedlmair et al. [253]. Several iterations were made to distill the fundamental user requirements and add the necessary features to arrive at the tool discussed in Sections 4.2.2 and 4.2.3. A screenshot from the VRNT is shown in Fig. 4.15.

4.2.1.1 Technology Probe

The technology probe was designed to explore the potential of using VR systems for generic scientific visualization tasks, and supported volume rendering, isosurfaces, and particle rendering (Fig. 4.16a). The user sits or stands at a desk and is able to move his or her head to look at the volume, or translate and rotate it using a gamepad. An initial

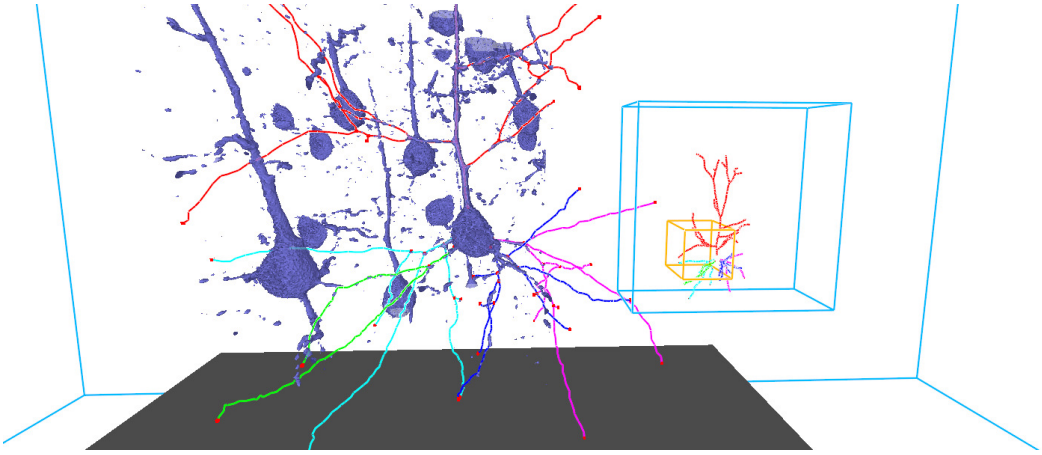
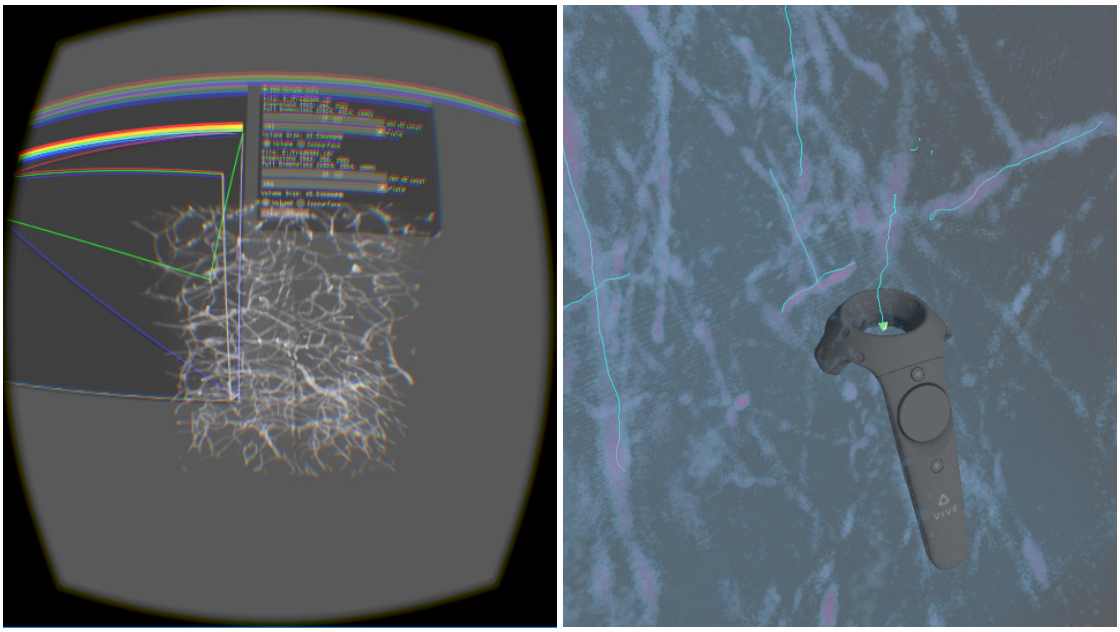


Fig. 4.15: A screenshot of the VRNT using the isosurface rendering mode. The dark gray floor represents the extent of the tracked space. Users can orient themselves in the data set via the minimap (right), which shows the world extent in blue, the current focus region in orange, and the previously traced neuronal structures. The focus region is displayed in the center of the space. The 3D interaction and visualization provides an intuitive environment for exploring the data and a natural interface for neuron tracing, resulting in faster, high-quality traces with less fatigue reported by users compared to existing 2D tools.



(a) Initial Technology Probe **(b)** Prototype

Fig. 4.16: The technology probe and prototype were used to explore different interaction and rendering possibilities for scientific visualization and neuron tracing in VR.

demonstration of this system to neuronatomists in the Angelucci lab with microscope scans of labeled neurons encouraged further pursuing VR to develop a tool for neuron tracing. In particular, the neuroscientists noted that, compared to standard 2D interfaces, the VR system allowed better perception of the spatial relations between neurons, one of the key challenges in neuron tracing. However, in the technology probe, the ability to interact with the data was limited by the restriction of the Oculus DK2 head-mounted display (HMD) [203] and the use of a gamepad as the input device. The Oculus DK2 can track small head movements while the user is facing a webcam style tracker, but does not support walking around a room. Although the gamepad can be configured as a 6 DOF controller, it is not tracked and thus cannot be used to reach out and "touch" the data directly.

The desire for direct 3D manipulation motivated the pursuit of a different interaction paradigm, built using the HTC Vive platform [126]. The HTC Vive supports room-scale VR and includes tracked, wand-style controllers. The room-scale tracking allows users to walk around, as well as into the data, and interact with the data naturally using their hands. Tilt Brush [96], which uses the same wands to paint in 3D, inspired the first prototype of the neuron tracing tool, which extended the painting metaphor to neuron tracing.

4.2.1.2 The Prototype

The first prototype dedicated to neuron tracing was designed to evaluate what different types of interactions would be useful, and explore how they could be mapped to the HTC Vive's control scheme. Based on the available space and hardware setup, a medium-sized tracked area was set up, about $2.5m \times 2m$, and the data placed in a 1.5^3m box in the center of the room at about $1m$ above the ground. The prototype used both wands, one to interact with the data and the other to navigate the space. Using the first wand, the user could hold a button and draw a line coming from the tip of a tetrahedron shown in the middle of the wand's loop (Fig. 4.16b). Only a 256^3 subregion of the volume could be rendered at a sufficient framerate for VR (see Section 4.2.2.2). This subregion is referred to as the *focus region*. The subregion was placed in the 1.5^3m box in the center of the VR space. The second wand could then be used to grab and move the data within this region. A minimap was placed in the corner of the room to orient users within the data set. The minimap displayed the data set bounds and the focus region's location within it. One notable observation was

that given the opportunity to pan, users often preferred to drag neurons closer as opposed to walking toward them.

As neuroanatomists are not familiar with transfer function design, even in a desktop setting, the prototype uses a fixed preset for the data sets, allowing neuroanatomists to focus solely on the task of tracing. Selecting from chosen presets has been found effective in medical visualization and museum installations [318], where users are also unfamiliar with designing transfer functions. Moreover, designing an effective interface for specifying transfer functions in VR is an open and challenging problem.

To evaluate the initial design of the tracing interaction, the neuroanatomists were asked to trace neurons in some data sets acquired by the Angelucci lab. After a short introduction to the control scheme (about 10 minutes), they were free to use the tool as desired. These users noted that the painting metaphor was intuitive. Compared to existing 2D tools, they found the prototype easier to use for exploring the data, allowing them to better resolve complex crossings and spatial relations of neurons in the data.

The prototype, despite being limited to line drawing and simple exploration, provided an initial validation of both the navigation and interaction design. To extend the prototype to a minimally viable tool, additional features that were typically used by neuroanatomists in the neuron reconstruction and analysis process in NeuroLucida were added to the VR tool. For example, color is used to distinguish axons and dendrites, and glyphs to mark areas of interest. NeuroLucida also allows for undoing operations and editing previous traces, which permits review and correction of previously traced neurons. Therefore, the initial prototype was extended by improving the tree drawing system and rendering quality, adding support for undoing and editing, placing markers, selecting line colors, and streaming large volumes from disk. Additional exploration was also conducted to expand the interaction paradigm by integrating haptic feedback. These improvements are incorporated into the current tool described in the following section.

4.2.2 Virtual Reality Neuron Tracing Tool

The design of the final tool focuses on two key aspects: the process of tracing and navigating (Section 4.2.2.1), and meeting the VR rendering performance requirements to provide a high-quality experience and prevent motion sickness (Section 4.2.2.2). To analyze

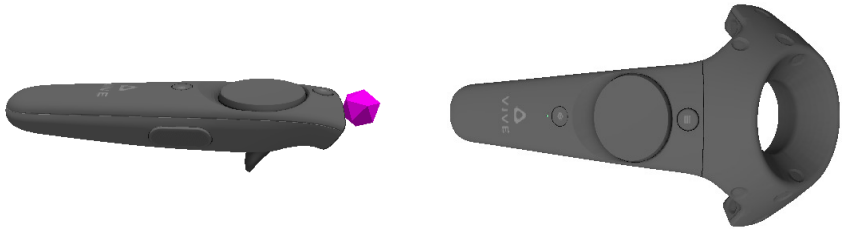
how scientists use the system and allow for its use as a training device, it also provides a recording and playback system (Section 4.2.2.3) that tracks the user's actions for later playback. Finally, the tool must fit into a larger data processing pipeline, which starts at the acquisition of volume data from a microscope and ends with the simulation and analysis of the reconstructions in the context of other brain maps. To fit well into the pipeline, the tool loads the IDX [215] volume format used by the Angelucci lab. Once the neurons of interest have been reconstructed, the data are exported in a standard XML format used by NeuroLucida. Furthermore, previously traced neurons in this format can be opened in the VRNT, allowing for inspection and editing of earlier work.

4.2.2.1 Tracing and Navigation

Tracing neurons and navigating the data are the key tasks when reconstructing neurons. Both interactions require the 3D motion to be intuitive, and therefore both interactions are mapped to the motion of each wand. One wand is used for tracing and the other for navigation (Fig. 4.17). Tracing and navigation actions are initiated by holding the trigger button on the corresponding wand. In the VR environment, the tracing wand is displayed with an icosphere at the top, indicating from where the line will be drawn, similar to a paint brush (Fig. 4.17a). The navigation wand is rendered to match the wand's physical model (Fig. 4.17b).

A neuron forms a tree that consists of a starting point, branch points, and termination points. Traces created by the user are stored in a graph structure that is updated with the user's edits and additions. To trace a neuron (Fig. 4.18), the user presses and holds the trigger button on the tracing wand, placing a starting point. The user then holds the trigger as he or she follows the neuron through the data, drawing a line from the brush. Releasing the trigger ends the line and creates the termination point. The user is then free to continue the line from the termination point, or trace branches as needed.

Tracing the branches of a neuron correctly is critical to properly recover its connections and structure. Moreover, this task is performed often, and therefore it must be easy to do. To create a branch, the user can start a new line along the current tracing and follow the neuron branch out (Fig. 4.19a), or start a new line on the neuron branch and reconnect to the parent tree (Fig. 4.19b). To call attention to the re-connection, the VRNT highlights the



(a) Tracing wand **(b)** Navigation wand

Fig. 4.17: The wand model shown in VR can be changed from the physical model. On the tracing wand (a), the top loop seen in (b) is removed to avoid occlusion while tracing. The button sticking out underneath (a) is the trigger, and the large circular button is the trackpad. The icosphere brush in (a) is colored to match the selected line color.

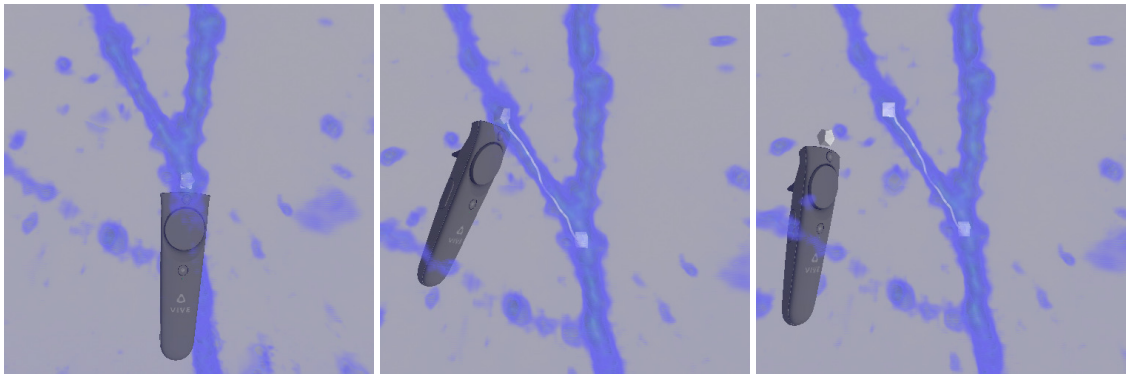
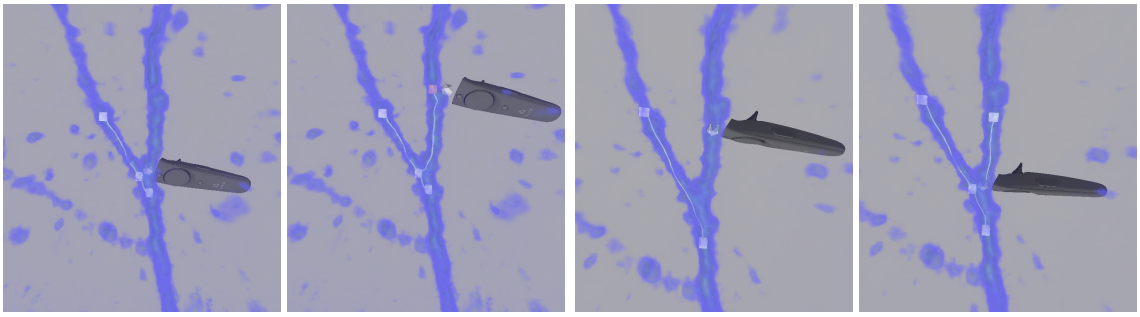


Fig. 4.18: From left to right: the neuron tracing process begins by finding a neuron. A starting point is placed by moving the brush inside the neuron and pressing the trigger. While holding the trigger, the user follows the neuron with the brush, tracing it. To end the line, the trigger is released.



(a) Branching from an existing line **(b)** Connecting a branch back to the parent tree

Fig. 4.19: A branch can be created by placing the brush close to an existing line, where a candidate branch point will be shown (a), or an existing node, and tracing from it. The branch can also be started as a new line and reconnected to the parent tree (b), in which case the candidate branch point created by the connection is shown.

selected node and sends a small vibration to the wand to give a "click" feeling of selecting it. When connecting back to a line, the candidate node that would be created when the trigger is released is displayed as a small cube to indicate where the branching point will be placed (Fig. 4.19b). The visual and physical feedback provides a clear signal to the user that the connection has been selected as desired.

During the tracing process, mistakes may be made that need to be corrected. For example, the user may have an incorrect initial understanding of a complex crossing, the user's hand could slip, or the system could drop a frame or momentarily lose tracking due to occlusion. Depending on the type of mistake, the user may make an immediate correction or revisit the error later. To correct mistakes, the tool provides two methods of undoing and editing: a quick fine-grained undo operation and the ability to remove entire lines and nodes at any time.

To immediately correct mistakes, the user can undo lines in the reverse order in which they were created by pressing the trackpad (Fig. 4.17). This undo is useful for quickly repainting segments where the user is not satisfied with how well the trace follows the neuron. The scope of the undo operation is controlled by placing undo breakpoints along the line every 40 voxels, with each undo operation reverting to a previous breakpoint. Furthermore, any part of the line can be repainted by drawing a new line over the problematic section and reconnecting it after the section. The new line forms a loop in the trace, and the old section will be removed to reduce the graph to a tree. Since a neuron is physically a tree, any loop represents an invalid structure and can be assumed to be an edit.

Scientists may notice errors when revisiting a previously traced section. The undo and line redrawing operations may not be applicable in such instances. Instead, the user can delete specific lines or nodes with the tracing wand. Editing operations are initiated by selecting a line or node with the wand, noted by highlighting the feature and a "click" vibration, and pressing the undo button. The user can then reconnect the disconnected trees as desired. For example, in Fig. 4.19a the selected line (left) or the highlighted node and attached edge (right) could be deleted by pressing undo.

Navigation around the data set is accomplished by walking or by translating the volume. Within the focus region, the user is able to walk around the space to navigate. To explore data outside the focus region or pull the regions closer, a panning action is mapped to

the navigation wand. By holding the trigger button and moving the wand, the user grabs the focus region and translates it through the volume. Via this interaction, arbitrary-sized volumes can be explored using the VRNT. Furthermore, as volume sizes are often larger than available GPU memory, the data are paged on and off as the user pans, described in detail in the following section. To help users track the location of the focus region relative to the data set and previously traced neurons, a minimap is displayed in a corner of the virtual environment (Fig. 4.15). When navigating large data sets, the minimap is useful to keep the user oriented as he or she pans through the space. Traced neurons are also displayed in the minimap to help users track their progress through the data set. To allow navigating at larger scales or getting an overview or close up view of the data, the VRNT also provides a zoom interaction. Zooming is performed by holding the grip buttons on both controllers and moving them further apart or closer together.

4.2.2.2 Rendering

The HTC Vive uses a display panel with a resolution of 2160×1200, providing 1080×1200 pixels per eye at a 90Hz refresh rate. Furthermore, due to lens distortion, it is recommended to supersample the image, effectively doubling the number of rendered pixels. Additionally, the VR environment imposes stringent lower bounds on the acceptable framerate to avoid motion sickness. The combination of high-resolution and frame-rate requirements presents a significant challenge compared to traditional desktop visualization, where low framerates and intermittent pauses for computations or data loading are more tolerable. To meet these requirements, we take cues from best practices for VR game development [286]. In order to communicate with the HTC Vive HMD, VRNT uses the OpenVR SDK, which provides methods for sending images to the eyes and tracking the head and wand positions.

Meeting the 90 frame per second (FPS) rendering target of VR places a tight rendering budget on the tool of about 11ms to render each frame, from which 1ms is potentially consumed by the operating system. Furthermore, as GPUs are pipelined architectures, submitted work is not executed immediately, but enqueued into a command buffer. To account for this, Vlachos [286] recommends submitting draw calls $\approx$ 2ms before VSync. Submitting early allows the GPU to start rendering immediately after presenting the

previous frame, increasing utilization.

Streamlining rendering performance requires pushing all nonrendering or noncritical work onto background threads and strictly budgeting work on the render thread. A single frame is divided as shown in Fig. 4.20. First, VRNT waits until $\approx$ 2ms before VSync by calling `WaitGetPoses` from the OpenVR SDK (left side of CPU in Fig. 4.20), which obtains the most recent head position. After returning from this function, all rendering work is submitted to the GPU. Opaque geometry, e.g., the wands and tracings, is rendered first (1ms). Next, the volume is rendered with raymarching to display a volumetric or implicit isosurface representation (4ms). After submitting the rendering work, VRNT starts the asynchronous volume data upload based upon the user's focus region, and once the rendering finishes, copies the uploaded volume data into the sparse texture (2ms). This time budget leaves a buffer of 3ms to prevent unpredictable interferences that could cause dropped frames. Nevertheless, sometimes a system event or an expensive draw call can consume this buffer, causing a frame to be skipped. When this occurs, the OpenVR SDK will automatically render at half framerate, 45 FPS, while reprojecting the last frame using the latest head tracking information to display at 90 FPS, until the framerate improves. Unfortunately, the reprojection cannot account for motion of the wands, as the image transform is based only on the head motion. When the system is reprojecting, the wands appear to stutter, making the interaction feel sluggish.

Care must also be taken to support smoothly zooming out on the data during navigation. Zooming out increases the number of voxels visible in the focus region, potentially degrading performance below 90FPS. To maintain an acceptable framerate, VRNT switches to a lower resolution volume to reduce the rendering load; when the user returns to the original zoom level, VRNT switches back to the full resolution data. The tool is able to quickly switch between these two volumes by leveraging the IDX format's support for multiresolution queries and the data streaming system discussed below by running two such data streaming systems simultaneously: one that is responsible for the high-resolution data, and a second that is responsible for the lower resolution data shown when zoomed out.

4.2.2.2.1 Data streaming. Typical microscopy volumes exceed the VRAM of current GPUs (4-24GB) and in most cases the RAM of typical workstations (64-128GB); data sets can

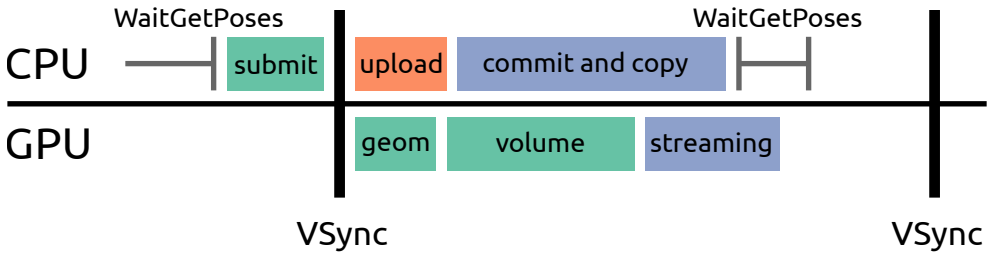


Fig. 4.20: The anatomy of a single frame. `WaitGetPoses` blocks until ≈ 2ms before VSync and returns the latest head tracking data, which allows the renderer to start submitting work before VSync to fully utilize the GPU. Draw calls for the geometry and volume are submitted first, after which the asynchronously uploaded volume data are paged in to the sparse texture.

range from hundreds of gigabytes to terabytes. Exploring such data sets inherently requires a data streaming solution. Moreover, as only 2ms is budgeted for data streaming on the render thread, the work of updating the volume data must be amortized over multiple frames, with as much work performed asynchronously as possible. VRNT uses a two-level caching system: the first level loads and caches pages from disk into RAM, and the second level takes these pages and uploads them to the GPU. The caching system lets the tool keep the current focus region and a small neighborhood resident on the GPU, while a substantial history is cached in RAM. The cache drastically reduces disk access frequency and latency to display pages as users navigate.

The first-level cache takes page requests and immediately returns a future, which can be used to retrieve the page data. In case the page is not available in the cache, a worker thread will be responsible for loading the data from disk, while the requester can asynchronously check for completion and retrieve the page. The second-level cache pushes page queries to a set of worker threads, which request the page from the first-level cache and copy the data into persistently mapped pixel buffer objects (PBOs). By uploading via persistently mapped PBOs, the caching system takes advantage of asynchronous data transfers via the GPU's copy engines, thereby overlapping rendering work with data transfers.

The volume data are stored on the GPU in a sparse 3D texture, a form of virtual memory where individual pages can be committed or decommitted. This texture allows for transparent handling of volume data larger than VRAM. The rendering work for a frame takes long enough for the asynchronous data upload to complete, after which the page is copied into a newly committed page in the sparse 3D texture (commit and copy, streaming

segments of Fig. 4.20). The system uploads only a limited number of pages per frame to stay within its time budget and decommits pages no longer needed.

To minimize visible popping of pages into view, the cache loads a box slightly larger than the focus region and prioritizes pages closer to the user's view. To avoid overwhelming the paging system by requesting many unneeded pages and creating a large backlog, e.g., in the case of quickly panning through the space, only the four highest priority new pages are enqueued each frame. Additionally, if a page is no longer needed by the time the PBO is filled, the page is not committed or copied to the texture, as it would be immediately decommitted. This scheme of limiting the enqueueing rate provides faster responsive times and lends itself to a simpler implementation compared to updating priorities for already scheduled pages in a nonblocking thread-safe manner.

4.2.2.2.2 Volume rendering. VRNT uses a GPU volume renderer written in GLSL [107]. Although the volume data are stored in a sparse 3D texture, this texture type requires no additional consideration in the GLSL code. Sampling missing pages is defined to return 0. In the raymarching step, the volume renderer uses the depth buffer produced by rendering the opaque geometry to terminate rays early in order to correctly composite with the geometry, wands, and tracings.

In a VR application, it is common to step inside the data set. However, when walking through the volume, it appeared to vibrate, or the isosurface to move subtly. As the camera moves through the volume, the voxels sampled by rays leaving the eye will be offset differently in the data, causing them to sample slightly different locations. This artifact can be mitigated by increasing the sampling rate, but such mitigation is prohibitively expensive for VR. The artifact is fundamentally similar to ensuring correct ray sampling across subvolumes in distributed volume rendering [183]. To ensure consistent sampling of the data when inside the volume, the renderer begins sampling at the sample point nearest to the clipping plane, based on starting the ray from its entry point into the volume bounds. This approach corrects only for translation, but was found to be sufficient in practice.

Even when using gradient shading, depth perception can be challenging in volume rendering, particularly when using transparent transfer functions and subtle lighting cues. This lack of depth cues can make it difficult to tell the exact position of the wands when placed inside neuronal structures, due to the faint occlusion effect provided by the volume.

In fact, during the pilot study, one user reported experiencing eye strain while viewing the volume representation, potentially due to the limited depth cues. More advanced rendering techniques such as shadows or global illumination [139] can improve depth perception, and potentially user performance, but come at significant frame-rate or memory cost, making them challenging to apply in a VR setting. To maintain a sufficient framerate for VR, the volume rendering quality in VRNT is relatively simple, providing just gradient shading.

To enhance depth cues, VRNT provides the ability to switch to an implicit isosurface mode, with Phong shading and ambient occlusion [118]. In the isosurface mode, the user can scroll on either of the two wands' trackpads to change the isovalue, which is necessary to resolve crossings or neurons with low intensity values. The front faces of the isosurface are rendered to be semitransparent, allowing the user to see when the brush and traces are placed well inside the neuron. The back faces, however, are rendered to be fully opaque, as it is difficult to perceive depth relations when they are semitransparent. When dealing with noisy data such as microscopy images, isosurfaces are often not ideal, as the noise manifests as small objects in the volume. Especially in a VR environment, these objects result in distracting aliasing artifacts. To counter this effect, volume pages are filtered before they are uploaded to the GPU; the filtering process removes objects less than 11 voxels in size by finding and removing small connected components.

The performance of volume raycasting is directly tied to the number of rays rendered, i.e., the number of shaded pixels. The rendering resolution recommended by the HMD is very high; fortunately, this resolution is needed only at the center of each eye. In the periphery, due to lens distortion and the properties of the human vision system, it is possible to render at much lower resolutions (e.g., using the `NV_clip_space_w_scaling` extension) with little to no perceived difference.

4.2.2.3 Recording and Replaying

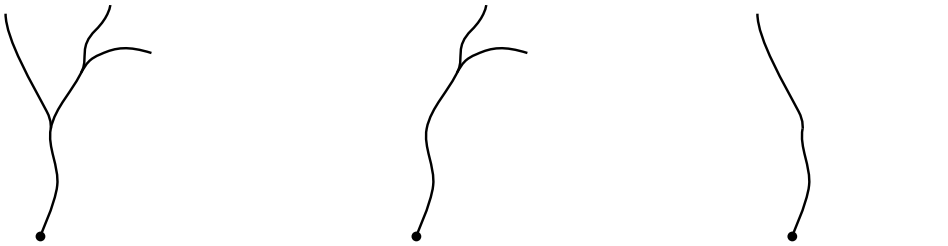
Evaluation and iteration of any tool requires understanding how it is used. Moreover, a recording of an expert's session can serve as training material for novices. In desktop applications, user sessions can be recorded using screen recording software. However, recording the "screen" in VR restricts the playback to a single viewpoint, potentially removing relevant context in the space. For example, a mistake made when tracing may

depend on the user's viewpoint, but to properly observe the situation, it must be possible to watch the user's session from a viewpoint different from the recorded one. Even more concerning is that viewing the recording in VR typically induces nausea due to the mismatch in the recorded head motion and the viewer's head motion. To this end, VRNT provides a recording and playback system for tracking user sessions that is based on the actions performed by the users, instead of video recording.

Such action-based recording can be performed at multiple levels. The low-level wand and HMD state and poses could be saved each frame, the tracing could be played back by stepping through snapshots, or the user's logical operations (e.g., *tracing*, *toggled isosurface*, or *panning*) could be saved. The last option provides the most flexibility for both playback and later analysis. This option also supports playback of the recording on different VR systems or later iterations of the tool with different control schemes, without needing to re-map low-level button presses or HMD information. Moreover, session analysis is easier with such a representation, as queries can be made at a higher level, e.g., "how far does the user trace in a single motion?" or "how long did they use each rendering mode?". Replaying a saved session recreates the entire tracing by moving the wands and HMD as they were during the session. By viewing the hand and head motions during the replay, it is possible to observe differences in how users work. During this time, the user viewing the replay can walk around the space independently.

4.2.3 Evaluation

A pilot study was performed with trained neuronatomists to evaluate the design choices made and compare VRNT to state-of-the-art desktop software. The pilot study involved tracing neurons in two distinct data sets and is run with seven users. The neuroanatomists in the Angelucci lab use NeuroLucida, a standard and widely used desktop software package for neuron tracing, in their day-to-day work. Thus, NeuroLucida is used as the desktop software baseline against which VRNT is compared. In practice, the two primary metrics of concern are accuracy and speed. In terms of accuracy, the goal is to determine the connectivity of neurons as well as possible, including geometric location and tree topology. Misinterpreting a crossing as a branch point or missing branches entirely will cause substantial errors in the subsequent analysis. Nevertheless, as usual in expert-driven



(a) Reference **(b)** Branch missing **(c)** Subtree missing

Fig. 4.21: Examples of different mistakes and their effect on the DIADEM score. Trees (b) and (c) are compared against the reference (a) with scores 0.875 and 0.5, respectively. The error in (c) misses a large subtree, impacting later analysis more significantly than the error made in (b).

systems, the final result is subjective, and experts sometimes disagree on specific choices. Furthermore, some mistakes are more critical than others. Slightly elongating a trace by crossing a small gap may be acceptable, but erroneously attaching a branching structure is not. In terms of speed, the field of connectomics is moving to acquisition of ever increasing amounts of data; therefore, the time necessary to trace neurons is of significant concern.

The DIADEM scoring method is used to automatically compare traces [92], the scoring method takes into account both the length and the connectivity of a trace. The computed score correlates well with expert judgment, and informal comparisons suggest it is a reasonable proxy for accuracy. The score measures the similarity between traces with values ranging from 0 (dissimilar) to 1 (identical). For example, missing a small branch in a large tree (Fig. 4.21b) has a smaller impact on the score than missing a large subtree (Fig. 4.21c).

The first data set consists of six aligned subvolumes containing 34 mostly planar axons, each with a reference tracing (Section 4.2.3.1). The second data set is a single large volume containing several noisy cell bodies (Section 4.2.3.2). In both cases, users are provided a predetermined set of points from which they start tracing a neuron in each tool. To avoid bias, the starting points are split into two sets that are traced on alternate days in different tools, such that no set was traced on consecutive days. In most cases, there were multiple days between sessions, due to users' work schedules. For the first case study, scores are reported with respect to the reference traces, whereas for the second case study, scores are computed by comparing them with traces performed by domain experts. Extensive qualitative feedback was also collected from the users during the study, both informally and through a questionnaire completed at the end of every tracing session.

The pilot study involved with seven users, including two senior neuroanatomists (users 4 and 6), two expert undergraduate students with 2-3 years of experience reconstructing neurons with NeuroLucida in the Angelucci lab (users 5 and 7), and three undergraduate students with no background in neuroanatomy (users 1-3). In a typical lab, the bulk of tracing work is performed by trained undergraduates (e.g., users 5 and 7), who start with little background (e.g., users 1-3) and are trained by senior members of the lab (e.g., users 4 and 6). Evaluating VRNT with a cross-section of the experience levels found in a typical lab can help determine how well the tool fits into existing workflows. Specifically, the tool must be usable by senior members and expert students for tracing, and be easy to learn for new hires such as users 1-3. Based on the range of experience with tracing in NeuroLucida, the users are binned into two groups, an *experts* group, consisting of the neuroanatomists and the expert undergraduates, and a *novices* group, with the three inexperienced undergraduates. During the evaluation, the VR tool ran on a workstation with a dual socket Intel Xeon E5-2680 CPU, 64 GB RAM, an NVIDIA GTX 1080 GPU, and an SSD.

4.2.3.1 Planar Axons Reference Data Set

Although all tracings can contain some subjectivity, it is important to establish a baseline of performance with respect to a given reference. The first data set is the *Neocortical Layer 1 Axons* data set [211] from the DIADEM challenge [92] and the corresponding reference traces. The data set consists of six volumes of neurons in a mouse brain that can be stitched to form a $1464 \times 1033 \times 76$ volume with the provided alignment information. The resolutions of the data are $\approx 0.08\mu m$/pixel in X and Y and $1\mu m$/pixel along Z.

The data set includes 34 reference tracings, of which the first two were used for training and the rest for evaluation. For each neuron, users started from the first point of the reference tracing and traced the corresponding neuron to its perceived termination points. Once all sessions had been completed, the traces made in each tool were compared with the provided reference tracings. In general, the experts rated the reference tracings as acceptable reconstructions, with the exception of a few neurons where branchings were judged to be crossings, or a crossed gap was considered too wide. Table 4.1 shows, for each user, the mean score, reconstruction time, and speed-up across all 32 evalution traces. Speed-up is

Table 4.1: Average scores (with standard deviation) and times in seconds for each tool across the 32 DIADEM traces used for evaluation. The average speed-up over all traces is also shown for each user. Users 1-3 are novices and 4-7 are experts. Both novices and experts performed similarly in VR, and tended to be faster on average. *User 1 miscalibrated the Z level in their NeuroLucida sessions, resulting in much lower scores for the majority of traces.

User	NeuroLucida		VR		Speedup
	Score	Time	Score	Time	
1*	0.27 ± 0.35	324	0.49 ± 0.35	172	1.9
2	0.34 ± 0.34	188	0.54 ± 0.37	161	1.2
3	0.48 ± 0.38	277	0.45 ± 0.36	149	1.9
4	0.57 ± 0.38	412	0.57 ± 0.36	271	1.5
5	0.56 ± 0.37	237	0.41 ± 0.38	151	1.6
6	0.65 ± 0.35	464	0.50 ± 0.40	229	2.0
7	0.51 ± 0.38	262	0.47 ± 0.41	302	0.9

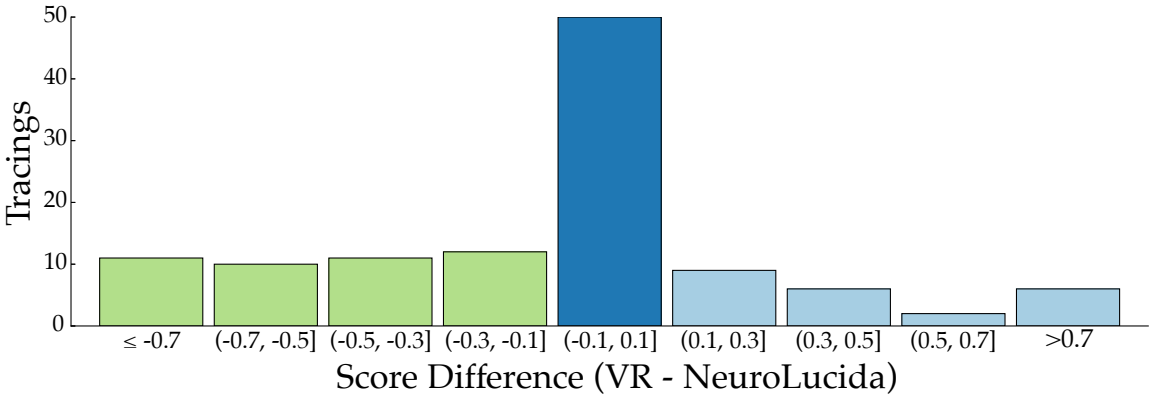
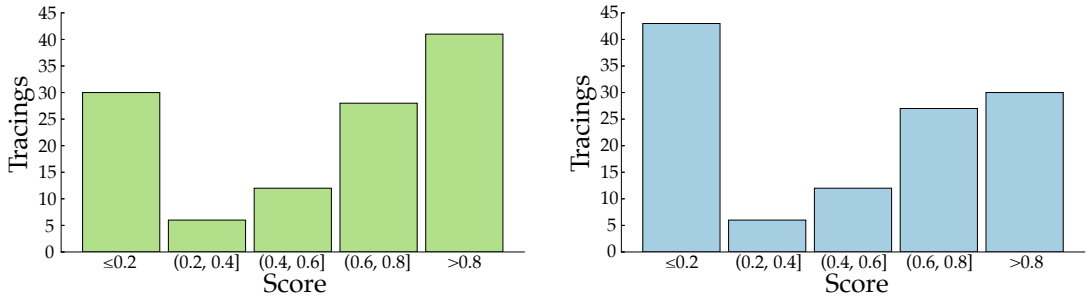


Fig. 4.22: Differences between scores of expert traces in VR vs. NeuroLucida. The difference in the score achieved in VR and NeuroLucida compared to the reference is shown for each trace. Overall, experts performed within the acceptable error range (± 0.1, dark blue) and sometimes better in VR (light blue) when compared to their work in NeuroLucida.

defined as the average time per tracing in NeuroLucida divided by the average time per tracing in VR.

When comparing the scores for traces done by the experts in NeuroLucida vs. those done in VRNT (Fig. 4.22), the majority of traces performed were acceptably equivalent in both tools (dark blue bar, Fig. 4.22), with some neurons traced better in each tool (green and light blue bars, Fig. 4.22). Overall, there was no statistically significant difference between the scores achieved in VR vs. NeuroLucida (Mann-Whitney $U = 6426.5$, $n_1 = 122$, $n_2 = 120$, $p = 0.097$). The distributions of scores for expects and novices are shown in Figs. 4.23



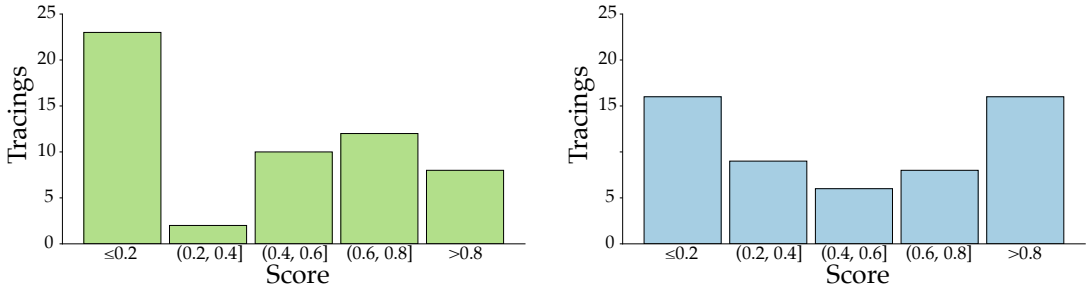
(a) NeuroLucida **(b)** VR

Fig. 4.23: Distribution of scores (higher is better) for experts. In (a) median score: 0.7, mean score: 0.57 ± 0.38. In (b) median score: 0.6, mean score 0.49 ± 0.39. A score of ≥ 0.8 is a tracing acceptably similar to the reference.

and 4.24, respectively.

The distributions of scores for experts (Fig. 4.23) indicate that experts can achieve similar, and sometimes better, tracing quality in VRNT when compared to their current workflow. In cases where experts produced equivalent quality traces in both tools, there is a statistically significant speed-up, with experts being on average $1.7\times$ faster tracing in VRNT (Mann-Whitney $U = 1004.5$, $n_1 = 54$, $n_2 = 54$, $p = 0.005$). Moreover, expert users were similarly consistent in VR and NeuroLucida, as indicated by the mean of standard deviations on each trace, 0.23 and 0.24, respectively. In fewer cases (37%, green bars in Fig. 4.22), the experts performed better in NeuroLucida than in VRNT, beyond the acceptable error bounds. When investigating these cases, it was found that they involved the same neuron for all experts, with each expert making the same mistake. One such neuron is the eighth neuron from the data set, where a stitching issue was misinterpreted as two neurons passing each other in VR (Fig. 4.25). The VR tool performed better in other cases (19%, light blue bars Fig. 4.22) where neurons traveled along the Z axis down through image slices, as this is much harder to follow in NeuroLucida, requiring scrolling through the image stack (Figs. 4.26 and 4.27).

Novice users performed similarly, with 72% of their traces falling within acceptable error or being better than those performed in NeuroLucida (Fig. 4.24). On average, novices were $2.1\times$ faster in VR for traces that they achieved similar scores in both tools. More significant results for the novices are not reported due to the limited data collected. User 1's results were discarded from the summary statistics entirely, as the user made a mistake in their NeuroLucida sessions and miscalibrated the Z level of the traces.



(a) NeuroLucida **(b)** VR

Fig. 4.24: Distribution of scores (higher is better) for novices, excluding user 1. In (a) median score: 0.5, mean score: 0.42 ± 0.37. In (b) median score: 0.49, mean score 0.5 ± 0.37. A score of ≥ 0.8 is a tracing acceptably similar to the reference.

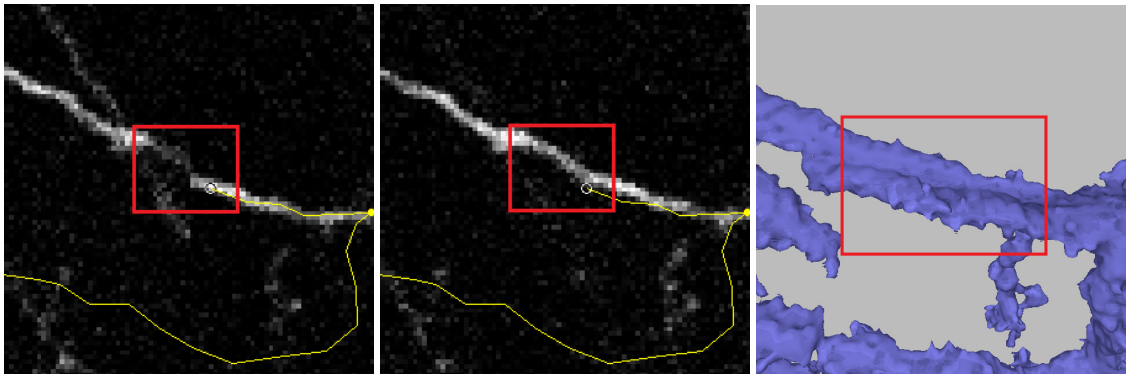
4.2.3.2 Cell Bodies Data Set

To evaluate the usability of VRNT on a data set with neuronal structures that were more familiar to the experts, the second case study used a data set acquired by the Angelucci lab. The data set, shown in Fig. 4.15, consists of neurons in the visual cortex of a marmoset monkey labeled with green fluorescent protein, and was acquired in 2012 using a 2-photon microscope. The volume is $1024 \times 1024 \times 314$ with a resolution of $0.331 \mu m$/pixel in X and Y and $1.5 \mu m$/pixel in Z. The neuronal structures in this data set branch significantly more often than those in the data set described in Section 4.2.3.1. Moreover, this data set has a higher level of noise and frequency of ambiguous cases, and is therefore more challenging to trace. This evaluation is mainly concerned with scaling in the sense of the cognitive load of the user, not necessarily data size. Five starting points were selected in the data set, to be traced by the experts in VR. Furthermore, as there is no reference available, scores were computed by relecting user 6's tracings as the reference (Table 4.2).

Since this data set is more complex, one aspect of interest is whether users would use the

Table 4.2: Average scores (with standard deviation) and times for traces on the Cell Bodies data set. User 6 is used as the reference.

User	VR Score	Time (s)
1	0.54 ± 0.23	537
2	0.58 ± 0.17	252
3	0.70 ± 0.23	207
5	0.66 ± 0.19	360
6	–	469
7	0.59 ± 0.34	542



(a) (b) (c)

Fig. 4.25: A stitching issue clearly visible in NeuroLucida (a-b), but difficult to perceive with volume rendering or isosurfacing (c). What appears as two neurons (c) is in fact a single neuron, slightly misaligned due to stitching issues at the border of two acquisitions. When scrolling through the image slices in NeuroLucida, the stitching issue can be seen by flipping between the slices (a-b) and those above and below. In NeuroLucida, all experts traced the neuron correctly, whereas in VR only one expert traced it correctly. Note that this issue is not specific to VR, but to the volume visualization method chosen.

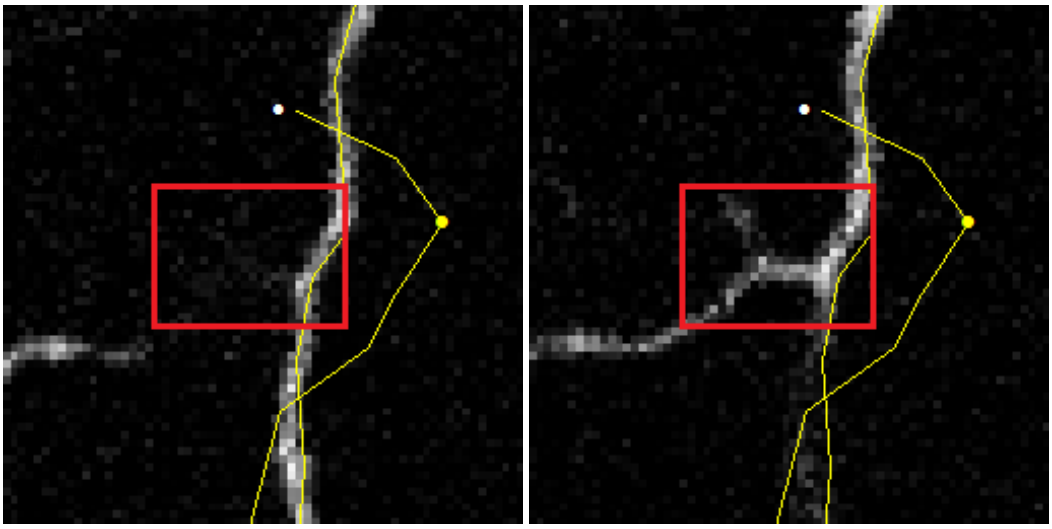


Fig. 4.26: A neuron branching along the Z plane is not visible on the image plane used to trace the main structure (left). The branch can be seen only after scrolling down the stack (right). Only two experts traced this branch correctly in NeuroLucida, but in VR all users traced it correctly.

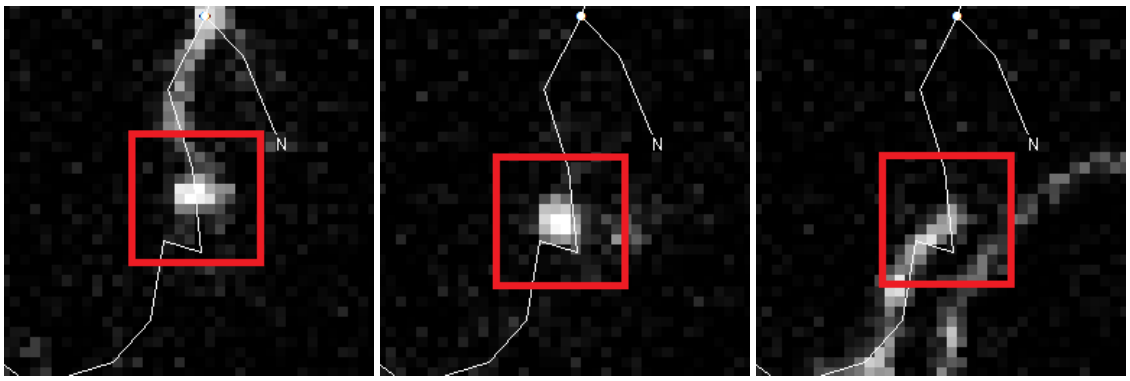


Fig. 4.27: From left to right: a neuron travels vertically through consecutive slices, appearing as a dot (middle) in these images. In NeuroLucida, only two experts traced this correctly, but in VR all users except one expert traced it correctly.

tool differently with respect to the more planar data set in Section 4.2.3.1. For example, if the ratio of time spent tracing vs. navigation change significantly compared to the previous data set. On average, users spent 15% of their time tracing and 48% panning, which is only slightly different from the 21% and 58%, respectively, in the *Neocortical Layer 1 Axons* data set. Users toggled between the volume and isosurface rendering modes more frequently in this data set. Although this result is interesting, it requires further evaluation on multiple data sets and a more rigorous measure of data complexity to provide meaningful evaluation.

During a review of several of user 6's sessions, it was noted that he would change viewpoints frequently to check potential branchings and to obtain a better understanding of the data. Such frequent viewpoint manipulation in NeuroLucida would require moving back and forth through hundreds of images. During these sessions several experts remarked that they would prefer to trace this data in VRNT, as scrolling through image stacks in NeuroLucida becomes more difficult as the data set thickness increases.

4.2.4 Discussion

The design study consisted of open-ended feedback sessions with neuroanatomists and quality evaluation of the tracings produced by using VRNT compared to state-of-the-art desktop software, NeuroLucida. Additional feedback was collected during the evaluation through a survey filled out at the end of each tracing session, and by soliciting feedback with regard to the usability and comfort of each tool. This section describes the users' qualitative responses to VRNT regarding neuron tracing, navigation, and rendering. Moreover, this

section discusses the overall strengths and limitations of the current tool.

4.2.4.1 Tracing

The experts reported that tracing neurons, creating branch points, and correcting mistakes were more intuitive in the VR tool than in NeuroLucida. The combination of tracing in free space with a single button press and the flexible graph editing system allowed users to focus on the data, instead of having to continuously flip back to a toolbar to mark branches and termination points, or frequently scroll through the image stack. Similar responses were recorded in the survey, with users rating the VR tool easier to use for these tasks. However, experts expressed the need for fine manipulation of lines and nodes to edit previous tracings, without going through a delete and re-trace interaction. When analyzing user sessions, it was observed that users employed the quick undo command more often than explicitly deleting lines or nodes; indicating that the delete and re-trace feature may be less intuitive, or that it may be more applicable to post-trace editing sessions. Designing an intuitive system for editing previous tracings in VR poses an interesting challenge. Introducing additional button commands or menus could make the system nonintuitive, and manually switching between tracing and editing modes could break the "flow" of a user during the task.

When replaying the tracing sessions, some errors in traces produced in VRNT were observed to be a result of the user forgetting to return to a branch point. In NeuroLucida, branches and termination points are explicitly marked; when a branch is ended at a termination point, the system scrolls the user back to the branch point to trace the rest of the branches. With VRNT's graph-based editing system, creating branches and termination points is implicit, and the user must remember to return to the branch point after completing the current branch. In the VR sessions, some users placed markers at complex crossings or branch points, as reminders to revisit the location later. Users also requested the ability to hide the data set entirely, allowing them to observe the traced tree structure independently. Although this could be achieved in VRNT by panning the focus region outside the volume to observe the tree from a distance, or viewing the minimap, providing an explicit option would be desirable.

4.2.4.2 Navigation

The users easily adapted to navigating by grabbing the focus region and moving it. During the tracing sessions, some users chose to sit, and the panning system allowed them to easily bring the data closer. One user commented that he felt as productive sitting as he did standing. The current version of our tool does not support rotating the volume, due to the inherent ability in VR to simply walk around the data set instead. However, this feature would be useful when tracing while seated. Instead, when seated, users moved the volume behind them and spun the chair around to view the data from the opposite side. In one case, an expert did not perform this less intuitive action, and misinterpreted a crossing as a branch point. When asked to re-trace this neuron while standing, the expert correctly resolved the crossing by observing it from a different angle.

Users found the minimap to be somewhat useful, especially for displaying the tracings; however, users reported rarely looking at it during active tracing, as it is small and tucked away in a corner. Users did report finding it useful for navigating to the starting point and reviewing the trace. The minimap may be more useful for data sets larger than the ones evaluated, where orienting oneself becomes more challenging. Additionally, in some cases users misinterpreted a neuron as terminating, when it was in fact just at the focus region boundary. Based on this feedback, VRNT was modified to also display the volume bounds in the world space to clearly convey the data set bounds.

4.2.4.3 Rendering

Neuroanatomists are often not familiar with typical scientific visualization representations such as volume rendering and isosurfaces. For example, when viewing the volume representation, users often misinterpreted stitching artifacts between acquisitions (Fig. 4.25). After a second VR session, one of the experts mentioned preferring the volume representation after gaining more understanding of what is shown. In his first session, he primarily used isosurfaces, but found them to be potentially deceiving, as neurons may manifest at some isovalues but not at others, and can appear to change thickness as the isovalue is adjusted.

Users also raised concerns that the volume and isosurface representations could hide or filter out faint or fine-detail features in the data, such as spines or boutons (small important

structures present on dendrites and axons, respectively). Expert users also suggested introducing the option of viewing the original microscope image slices within the volume data, in order to supplement the new representation with something more familiar to the neuroanatomist. It would also be valuable to add support for clipping planes to cull out noisy or dense regions of the data. During the evaluation sessions, users employed the focus region bounds as a form of clipping plane, by panning the data in and out of view, indicating a need for this feature.

When reviewing traces in the reference data sets where users performed poorly in VR, it was found that some of these cases involved crossing a gap in the data where the labeling of the neuron was faint or incomplete. In NeuroLucida, users correctly perceived this gap as caused by nonuniformity in the signal. However, no scale bar is displayed in VRNT tool, which could lead users to misinterpret the size of gaps or structures they are seeing in physical units.

Novice users found the immersive 3D representation provided in VR helpful in understanding the 3D nature of the neuronal structures. One novice mentioned that after using the VR tool, she was better able to construct the 3D structure mentally when working on 2D image slices in NeuroLucida.

4.3 Interactive Visualization of Terascale Data in the Browser

This section discusses work on an initial evaluation as to how new web technologies, WebAssembly and WebGPU, can be leveraged to enable scientific visualization applications directly in the browser. WebAssembly [299] and WebGPU [300] are new web technologies that can help address the issues faced when bringing scientific visualization applications to the browser. WebAssembly is a bytecode format to which native code can be compiled (e.g., through Emscripten [322]) that can be executed at near-native speeds in the browser (Section 4.3.1). WebGPU is a new low-level graphics API for the browser, similar in spirit to DirectX 12, Vulkan, and Metal, that exposes advanced rendering and compute functionalities (Section 4.3.2). The new capabilities provided by these technologies can be used to develop powerful scientific visualization applications.

The capabilities of these technologies are assessed through a series of small visualization application examples (Section 4.3.3). First, the performance of WebAssembly builds of

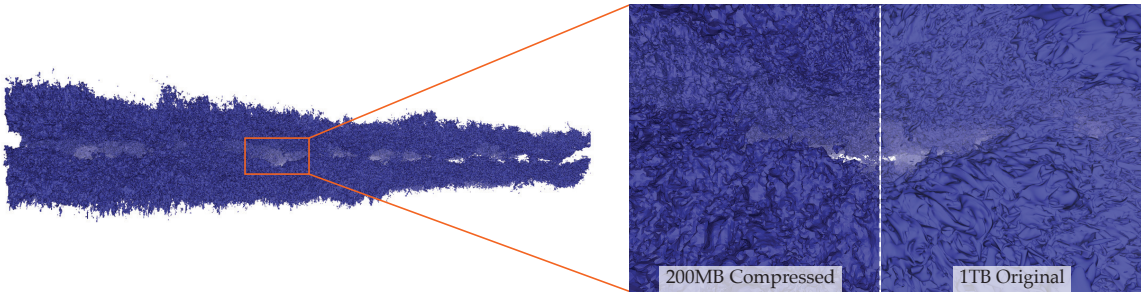


Fig. 4.28: Interactive visualization of an isosurface of a ~1TB data set entirely in the web browser. The full volume is a float64 10240×7680×1536 grid computed by a DNS simulation [166, 167]. The isosurface is interactively computed and visualized entirely in the browser using the BCMC GPU isosurface extraction algorithm for block-compressed data, after applying advanced precision and resolution trade-offs [119, 120, 171]. The surface consists of 137.5M triangles and is computed in 526ms on an RTX 2070 using WebGPU in Chrome and rendered at 30FPS at 1280×720. The original surface, shown in the right split image, consists of 4.3B triangles and was computed with VTK's Flying Edges filter [250] on a quad-socket Xeon server in 78s using 1.3TB of memory.

widely used native libraries for importing images and LIDAR data are compared to their native versions. Marching cubes [179] is used as a proxy for common floating point and memory intensive scientific visualization algorithms, both in WebAssembly and in parallel in WebGPU. Finally, a new GPU-driven parallel isosurface extraction algorithm, BCMC, is presented for interactive isosurfacing of block-compressed volume data sets in memory-constrained client devices to enable large-scale data visualization in the browser (Fig. 4.28). The isosurface extraction algorithm makes use of GPU decompression and a GPU-managed LRU cache to achieve interactive isosurface extraction with a small memory footprint.

4.3.1 WebAssembly

When considering bringing an existing scientific visualization application to the web, or deploying an entirely new one, a major concern is the lack of libraries widely used in native applications, e.g., for loading different data formats or performing computation. Previously, developers may have chosen to port the required libraries to JavaScript (e.g., VTK.js [146]), or recompile them to JavaScript using Emscripten [322]. Porting a substantial set of dependencies to JavaScript is clearly a major undertaking, and although Emscripten can compile C and C++ code to JavaScript, the performance of JavaScript is typically insufficient for computationally intensive tasks. Today, the above issues can be addressed

by using Emscripten, or toolchains for other languages, to compile the required libraries to WebAssembly.

WebAssembly (Wasm) [299] is a standardized bytecode format for a virtual machine that can be run in the browser or natively. In contrast to Java Applets or Flash, WebAssembly is integrated natively into the browser, supporting tight integration with JavaScript and security through sandboxing, and is targetable by C and C++ through Emscripten. WebAssembly is shipping today in the major browsers, Chrome, Firefox, Safari, and Edge, ensuring wide availability on user systems. In a study of various compute benchmarks, Jangda et al. [135] found WebAssembly to be 1.5–3.4× slower than native code, depending on the task. These slowdowns can be partly attributed to enforcing WebAssembly's security guarantees, which require some additional checks to be performed.

4.3.1.1 Compiling Native Libraries to WebAssembly

The Emscripten compiler can be used as a drop in replacement C or C++ compiler and can compile most portable code to WebAssembly. The main challenge when compiling a library to WebAssembly is the number of dependencies required by the library, as each one must also be compiled to WebAssembly. Large libraries such as VTK [249], whose dependencies may themselves have a number of dependencies, can produce a large graph of libraries that must all be compiled to WebAssembly to link against.

To provide a convenient API for calling into C++ libraries from JavaScript and working with C++ objects, library authors can use the binding generation tools provided by Emscripten: WebIDL Binder (similar to SWIG) or Embind (similar to Boost.Python). For smaller libraries or minimal bindings, it is also possible to export a C API and call it directly from JavaScript.

Two example scientific visualization applications run in the browser using libraries compiled to WebAssembly are used to evaluate WebAssembly's performance. The neuron visualization example (Section 4.3.3.1) uses the C libtiff library to load TIFF image stacks, and the LIDAR visualization example, which uses the C++ LASlib [162] library to load las and laz LIDAR data files. The libtiff library depends on zlib and libjpeg to compile libtiff to Wasm these dependencies had to be compiled to Wasm first, and then linked with libtiff to produce the final Wasm library. A minimal C API wrapper was written for the LASlib WebAssembly

library (Listing 6), which is used to discuss the lower level details of calling between JavaScript and C or C++ Wasm modules. The libtiff (github.com/Twinklebear/tiff.js) and LASlib (github.com/Twinklebear/LAStools.js) WebAssembly modules are available on Github.

4.3.1.2 Sharing Memory Between JavaScript and WebAssembly

The heap space of a WebAssembly module is stored within a JavaScript ArrayBuffer object that can be grown to meet dynamic allocation needs of the module (i.e., `malloc` and `new`). Pointers in the module are simply offsets into this ArrayBuffer. Thus, the module can directly share memory with the JavaScript host code by returning a pointer that can be used

```
struct LASFile {
  LASreader *reader;
  std::vector<float> positions;
  LASFile(const char *fname);  // Reads file using LAStools
};
extern "C" LASFile* openLAS(const char *fname) {
  return new LASFile(fname);
}
extern "C" float* getPositions(LASFile *file){
  return file->positions.data();
}
extern "C" uint64_t getNumPoints(LASFile *file) {
  return file->positions.size() / 3;
}
```

```
// Setup call information for the exported C functions
var openLAS = Module.cwrap("openLAS", "number", ["string"]);
var getPositions =
    Module.cwrap("getPositions", "number", ["number"]);
var getNumPoints =
    Module.cwrap("getNumPoints", "number", ["number"]);
// Write file data into Emscripten's virtual filesystem
FS.writeFile("data.laz", new Uint8Array(fileData));
var lasFile = openLAS("data.laz");
// Create a view of the Wasm module's point data
var positions = new Float32Array(HEAPF32.buffer,
    getPositions(lasFile), getNumPoints(lasFile) * 3);
```

Listing 6: A subset of the C API (top) provided by the example LASlib WebAssembly module and its use from JavaScript (bottom). The C API manages loading the data using LASlib and stores it in memory that can be shared with JavaScript. JS creates a view of the module's memory starting at the "pointer" returned by the API to access the data.

to create a view of the module's heap starting at the returned index. This approach is used to share the point cloud data loaded by the LASlib WebAssembly module with JavaScript without copying (Listing 6).

However, it is not possible for WebAssembly modules to directly see JavaScript memory. To pass an array of data from JavaScript to a module, space must be allocated from the module's heap to copy the array into. The index to which the array was written in the heap is then passed to the module as a pointer. To minimize the number of such copies that must be made to pass arrays to the module, it is best to have the module store large arrays in its heap and have JavaScript create an array view of the heap.

4.3.2 WebGPU

WebGPU [300] is a modern graphics API for the web, in development by the major browser vendors and available for testing in the preview builds of Chrome, Firefox, and Safari. WebGPU fills a role similar to Vulkan, DirectX 12, and Metal, providing a low-level API with more explicit control to the user and fixed inputs to the driver, allowing for improved performance compared to WebGL. Moreover, WebGPU exposes additional GPU functionality that is key to implementing general parallel compute algorithms and processing large data, providing support for compute shaders, storage buffers, and storage textures.

The concepts used in WebGPU should be familiar to users of Vulkan, DirectX 12, and Metal, although WebGPU is not as "low level" as the native APIs, striking a good balance between usability and performance. In WebGPU, rendering or compute passes are recorded using a command encoder to buffer up and submit work to the GPU. When recording a pass, a rendering or compute pipeline can be bound. The pipeline specifies the shaders to run, the layout of buffer inputs to the shaders, and, for rendering pipelines, the layout of vertex attributes and output render targets. Groups of buffers and textures matching the specified layouts can be bound to pass data to the shaders. The fixed layout of the pipeline allows the GPU to optimize its execution, while still allowing the data being processed to be changed as needed. This fixed layout is in contrast to WebGL and OpenGL, where the entire rendering or compute state cannot be provided to the driver in advance, limiting the optimizations that can be performed.

4.3.3 Example Applications and Performance Evaluation vs. Native Execution

This section evaluates a set of example applications for common visualization tasks that make use of WebAssembly and WebGPU. The applications cover a range of common operations performed in visualization: loading TIFF image stacks of microscopy volumes (Section 4.3.3.1), interactive visualization of LIDAR data (Section 4.3.3.2), naive serial and data-parallel isosurface extraction (Section 4.3.3.3), and volume decompression (Section 4.3.3.4). A performance comparison against native code on the same task is conducted for example, to assess the capabilities of WebAssembly and WebGPU for scientific visualization applications.

4.3.3.1 Neuron Visualization

The first example is a web application for visualization of connectomics data (Fig. 4.29). The application uses libtiff compiled to WebAssembly to import TIFF stacks that are volume rendered using WebGL2. TIFF files are loaded by writing the file data into Emscripten's virtual in memory filesystem, after which the file name can be passed to `TIFFOpen` and read through Emscripten's implementation of the POSIX file API. The application can also import SWC files to display neuron reconstructions with the volume data. This tool was written to facilitate collaboration on another project, allowing visualization researchers to

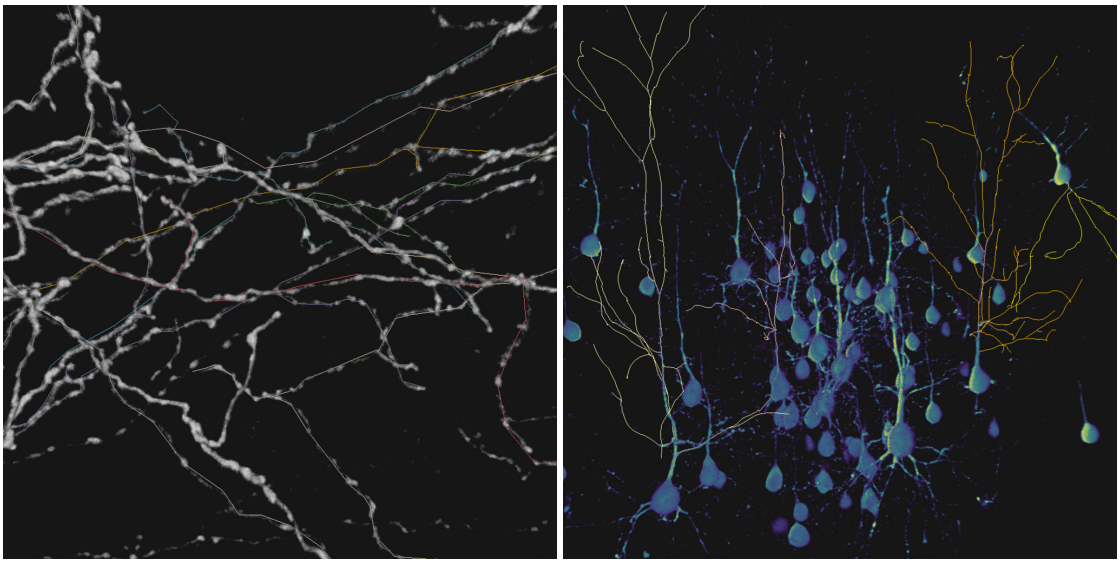


Fig. 4.29: Images of the neuron visualization application, used by neuronatomists to judge the quality of neuron reconstructions (rendered as lines) in the browser.

Table 4.3: Load times for the connectomics data sets using the native build of libtiff and the WebAssembly module in Firefox and Chrome.

Data Set	Native	Firefox 78	Chrome 83
Layer 1 Axons (44MB, 76 files)	0.87s	1.8s	2.6s
Cell Bodies (263Mb, 314 files)	4.1s	8.6s	16s

easily collect feedback from neuroanatomists on reconstruction quality. The neuroanatomist is able to simply open the web page, upload the data and traces, and interactively compare multiple reconstructions. WebGL2 was used for the rendering component of this tool to ease deployment, as WebGPU is not yet available outside preview browser builds.

Table 4.3 compares the time to load two connectomics data sets using the WebAssembly build of libtiff and the native version. The benchmarks are run on a Surface Pro 7 laptop with an i5-1035G4 CPU and 8GB of RAM. The performance difference observed between WebAssembly and native code is similar to those reported by Jangda et al. [135], with WebAssembly being about 2-4$\times$ slower than native code. The browser's implementation of the WebAssembly engine is an important factor, with Firefox outperforming Chrome by 2$\times$.

4.3.3.2 LIDAR Visualization

The second application example is LIDAR visualization, a common task in geospatial applications. To enable loading LIDAR data directly in the browser, the widely used library LAStools [162] is compiled to WebAssembly. LAStools has no external dependencies, making it easy to compile to Wasm; however, it is a C++ library and thus requires a C API to be written or generated to be called from JavaScript. A minimal C API was added on top of LAStools, which supports loading files and retrieving the contained points and colors to provide an illustrative example of the process (see Listing 6). Laz files provided by the user are written into Emscripten's virtual filesystem to allow the library to open them. The loaded points are then rendered as billboarded quads using WebGPU (Fig. 4.30) to display varying size round or square points.

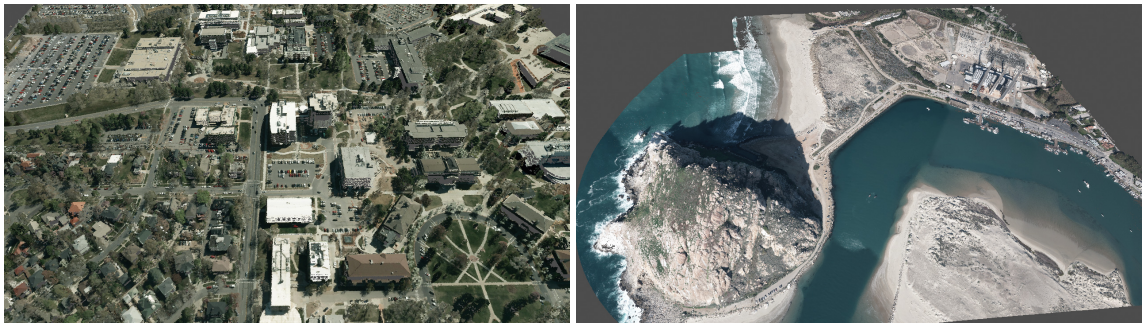
Table 4.4 compares the time to load various LIDAR data files using the native and WebAssembly versions of LAStools. The LIDAR files are provided in the compressed laz format, and contain 9M to 40M points. The benchmarks are run in Chrome Canary and Firefox Nightly, as they are required for WebGPU support. The benchmarks are performed

on a Surface Pro 7. As before, Firefox's WebAssembly engine is found to be 2-3× slower than native code, with Chrome's engine an additional 2-3.5× slower than Firefox. Although performance trails native code overall, the loading times achieved for the data sets are likely acceptable in an application.

Fig. 4.30 reports the rendering performance achieved by the WebGPU LIDAR renderer on the smallest and largest data sets tested, on both a Surface Pro 7 and a desktop with an RTX 2070 GPU. The basic renderer implemented in this example, which does not employ level of detail or filtering to accelerate rendering, is able to provide interactive rendering of the largest data set, even on the integrated GPU of the Surface Pro 7.

4.3.3.3 Marching Cubes

While the prior examples have focused on I/O and memory intensive tasks, namely, loading compressed and complex data formats, another key concern is the performance of compute intensive tasks. The classic marching cubes [179] algorithm is used as a representative proxy for the compute demands of common visualization tasks performed



(a) Utah 9M, points **(b)** Morro Bay, 40M points

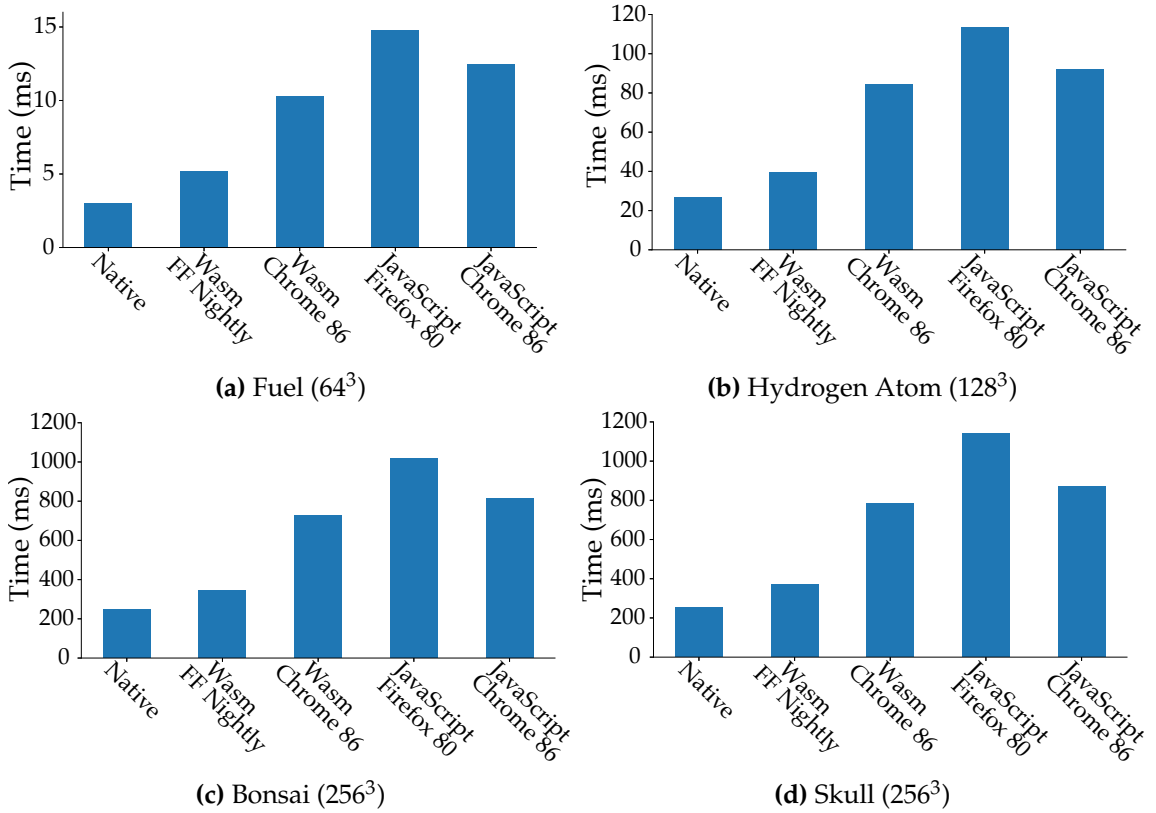
Fig. 4.30: Images from the WebGPU LIDAR renderer. At 1280×720 the images render at (a) 100 FPS and (b) 35FPS on an RTX 2070, and (a) 14FPS and (b) 3.5FPS on a Surface Pro 7.

Table 4.4: Load times for the laz files using the native build of LASlib and the WebAssembly module in Firefox and Chrome.

Data Set	Native	Firefox 80	Chrome 86
Utah 9M	3.8s	9.2s	29s
Morro Bay 20M	7.8s	20s	68s
Utah 30M	13s	30s	100s
Morro Bay 40M	16s	41s	133s

in scientific visualization applications. Serial implementations of marching cubes are evaluated in C++, JavaScript, and WebAssembly, and data-parallel variants evaluated in Vulkan and WebGPU. The data-parallel variants are similar to standard approaches using CUDA [61, 67]. First, a compute shader run over all voxels marks those containing the surface. Next, the active voxel IDs are compacted using a GPU prefix sum and stream compaction. For each active voxel, a compute shader determines the number of output vertices and writes the result to a global buffer. The offsets for each voxel and total number of vertices are computed using a GPU prefix sum. A second pass over the active voxels computes and outputs the vertices to produce a triangle soup.

The serial implementations are benchmarked on the Surface Pro 7 on four small data sets, ranging in size from 64^3 to 256^3, and are run in Chrome Canary and Firefox Nightly (Figure 4.31). Each benchmark computes 100 random isovalues using each variant and reports the average time to compute the surface. The 100 random isovalues are sampled over a range covering surfaces typically of interest to users, excluding noise and values so high



(a) Fuel (64^3) **(b)** Hydrogen Atom (128^3)

(c) Bonsai (256^3) **(d)** Skull (256^3)

Fig. 4.31: Performance of our serial C++, JavaScript, and WebAssembly versions of marching cubes on 100 random isovalues.

Table 4.5: Performance of the naive data-parallel marching cubes implementation on 100 random isovalues. WebGPU performance is typically on par with native Vulkan code. $*$ failed on the Surface Pro 7.

(a) Surface Pro 7

Data Set	Vulkan	WebGPU
Fuel 64^3	25ms	26ms
Hydr. 128^3	58ms	86ms
Bonsai 256^3	228ms	261ms
Skull 256^3	279ms	282ms
Plasma 512^3	1357ms	$*$

(b) RTX 2070

Data Set	Vulkan	WebGPU
Fuel 64^3	2ms	18ms
Hydr. 128^3	8ms	18ms
Bonsai 256^3	49ms	49ms
Skull 256^3	70ms	50ms
Plasma 512^3	322ms	329ms

as to produce few output triangles. The results of the benchmarks are shown in Fig. 4.31. The WebAssembly version performs well in Firefox, on average performing only 1.5× slower than native code. Firefox's JavaScript engine is on average 4.4× slower than native code, with the performance gap between JavaScript and WebAssembly increasing with data set size. Chrome's WebAssembly performance is on par with its JavaScript performance, averaging 3.2× and 3.6× slower than native code, respectively.

The data-parallel variants are benchmarked on the Surface Pro 7 and a desktop with an RTX 2070 (Table 4.5), and test on an additional 512^3 data set. The first computation is discarded in each benchmark, as WebGPU was found to have a high first launch overhead. On the Surface Pro 7, there is a moderate improvement over the fastest serial implementation in the browser on larger data sets, although both the Vulkan and WebGPU variants achieve performance just on par with the serial native C++ version. When compared to the fastest WebAssembly version, the Bonsai and Skull see performance improvements of ∼1.3×. However, significant performance improvements over the serial implementations are observed on the RTX 2070, where the data-parallel variants achieve a 7× improvement on the 256^3 data sets over WebAssembly in Firefox, and a 5× improvement over the serial native implementation.

The most exciting takeaway from this evaluation is that the performance of WebGPU is typically on par with native Vulkan code, indicating that little performance is lost when moving GPU compute kernels to the browser. On smaller data sets, WebGPU is found to be slower, where overhead in the WebGPU mapping to the underlying native graphics API may impact performance. This overhead is hidden on larger data sets, where the

computation dominates the total execution time.

4.3.3.4 ZFP Decompression

When visualizing large-scale data on a remote client (e.g., the 1TB volume shown in Fig. 4.28), transferring the entire data set to the client can be impractical or impossible. To reduce bandwidth and memory requirements, a compressed version of the data can be transferred instead, that the client can decompress as needed. ZFP [171] provides fast and high-quality compression of floating point scientific data, and is especially suited to large volumetric data. ZFP compresses volumetric data in 4^3 blocks, using fixed or variable bitrate compression. ZFP's fixed rate compression mode can be quickly compressed and decompressed in parallel on the GPU, and a CUDA implementation is provided with the library. This example evaluates the performance of parallel decompression in the browser, by porting ZFP's CUDA decompressor to WebGPU and Vulkan.

The decompression benchmarks are run on both the WebGPU and Vulkan ports of the CUDA decompressor on three data sets, using the Surface Pro 7 and an RTX 2070 (Table 4.6). Each data set is compressed at three different bitrates: two, four, and eight. The benchmark decompresses each compressed version 10 times and reports the average decompression performance achieved. The native comparisons on the CPU and in CUDA use ZFP prerelease version 0.5.5-rc1 built from Github. As CUDA is not available on the Surface Pro 7, and parallel decompression with OpenMP is not supported in ZFP, ZFP's

Table 4.6: ZFP decompression performance of the WebGPU implementation compared to a native Vulkan version and ZFP's original CUDA and serial decompressor. Neither the Vulkan nor WebGPU implementation is on par with the original CUDA implementation, though both are still capable of fast parallel decompression.

(a) Surface Pro 7

Data Set	CPU (Serial)	Vulkan	WebGPU
Skull 256^3	329MB/s	756MB/s	697MB/s
Plasma 512^3	349MB/s	898MB/s	777MB/s

(b) RTX 2070

Data Set	CUDA	Vulkan	WebGPU
Skull 256^3	67.1GB/s	14.8GB/s	7.6GB/s
Plasma 512^3	67.8GB/s	19.4GB/s	11.8GB/s
Miranda 1024^3	113GB/s	19.6GB/s	13.7GB/s

serial decompressor is used as the native comparison point on the Surface Pro 7. The decompressed output of the largest data set, Miranda (4GB), does not fit in the Surface Pro 7's 3.8GB VRAM, and thus the Surface Pro 7 is used to evaluate only the smaller data sets.

Although the Vulkan and WebGPU versions achieve slightly over 2× faster decompression on the Surface Pro 7 compared to the serial CPU decompressor, both are slower than the CUDA decompressor, by up to 8× and 6× on the Miranda. The WebGPU implementation is also found to trail the Vulkan version slightly, as observed previously on the data-parallel marching cubes example. Although the Vulkan and WebGPU implementations have room for improvement compared to ZFP's CUDA decompressor, the performance improvements provided over serial decompression are substantial.

4.3.4 Parallel Isosurface Extraction from Block-Compressed Data Using WebGPU

Based on the performance results observed on the example applications, it is clear that the browser can provide a capable environment for scientific visualization tasks. However, working with large data sets in the browser using WebGPU is challenging due to limitations on network transfer bandwidth and GPU memory, requiring a combination of adaptive precision and resolution compression (e.g., [119, 120, 171]).

The following observations motivated the design of the BCMC algorithm to enable interactive isosurface computation on large-scale data in the browser. First, the isosurfaces of interest to users typically occupy a sparse subset of the volume, which likely does fit in memory. Users are also likely to explore nearby isosurfaces, touching the same regions of the volume repeatedly. Third, ZFP's fixed-rate compression mode allows specific 4^3 blocks to be decompressed, without decompressing the entire data set. Finally, WebGPU performance is on par with native, and thus pushing as much computation as possible to the GPU will be beneficial for processing large data sets.

An illustration of the processing pipeline of the BCMC algorithm is shown in (Figs. 4.32 to 4.34), the code is also available on GitHub[1]). BCMC begins by uploading the ZFP fixed-rate compressed data to the GPU, allowing it to decompress blocks as needed without additional interaction with the CPU. To compute the isosurface, BCMC first finds the blocks

[1] `https://github.com/Twinklebear/webgpu-bcmc`

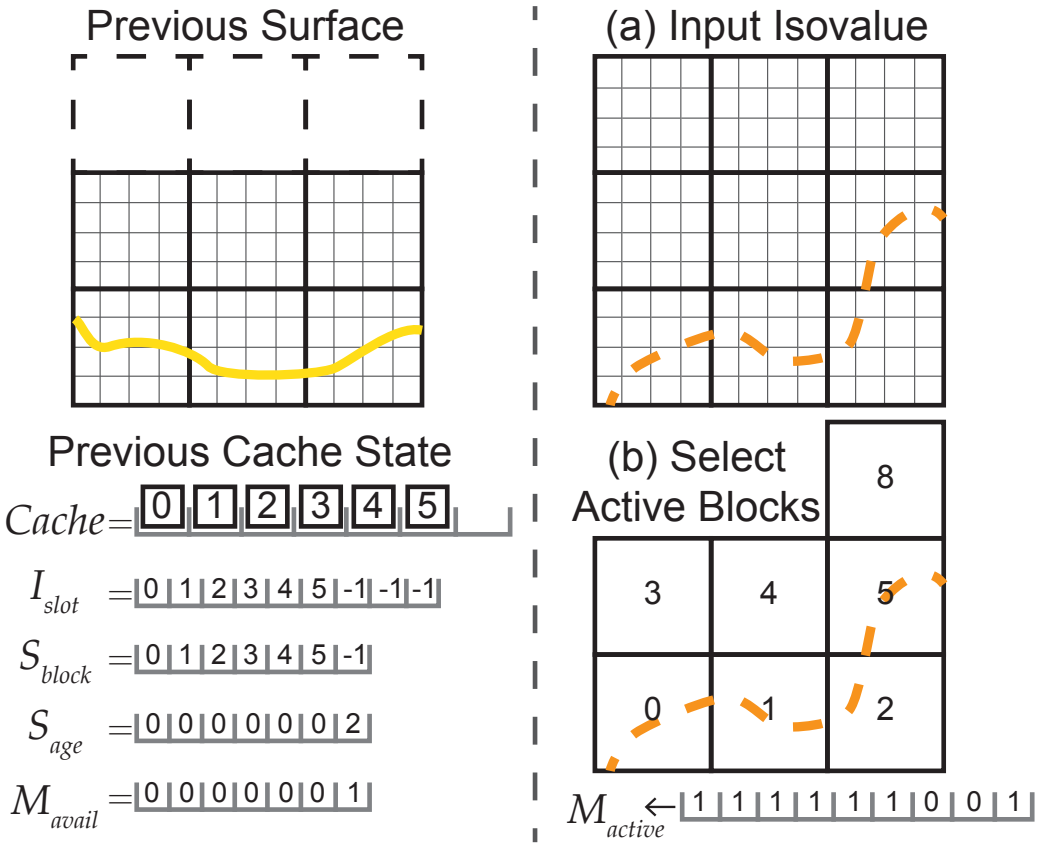


Fig. 4.32: Starting from an existing cache used to compute a previous surface (or an empty one), (a) the surface at some new desired isovalue must be computed. (b) BCMC first finds the blocks that contain the isovalue or are needed to provide neighbor data for blocks containing it.

containing the data required to compute it (Section 4.3.4.1). BCMC then determines which of these blocks are in the cache or must be added to it (Section 4.3.4.2). Each new block is assigned a slot in the cache and decompressed into it (Section 4.3.4.2.1). Active blocks are processed by loading them and the required neighbor voxels into shared memory (Section 4.3.4.3). For each active block, BCMC computes the number of vertices output by its voxels (Section 4.3.4.4). Finally, for each block that will output vertices, BCMC computes the vertices to output a triangle soup (Section 4.3.4.5).

4.3.4.1 Selecting Active Blocks

As done by Liu et al. [172], BCMC precomputes and stores the value range of each block when loading the data. To find active blocks, a compute shader is run over the blocks that marks a block as active if its range contains the isovalue (Fig. 4.32b). However, in

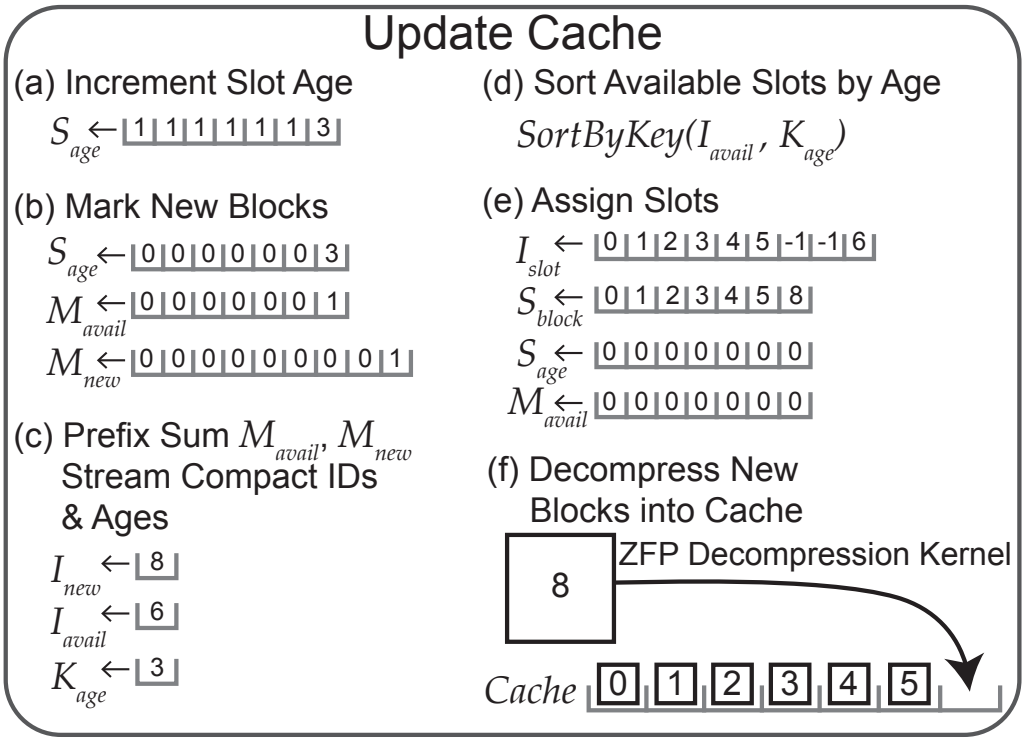


Fig. 4.33: BCMC's GPU LRU cache determines which of the blocks needed for the surface are already in the cache and those that must be decompressed and added to it (also see Algorithm 1). (a) First, the age of each slot is increased. (b) Next, a GPU kernel marks the new blocks that must be decompressed, the kernel also marks slots occupied by blocks that are not needed as available. The kernel resets the age of slots occupied by blocks needed for the current surface and marks them unavailable. (c) Lists of the IDs of the available slots and new blocks are compacted on the GPU. (d) The available slots are sorted by age, to evict the oldest blocks first. (e) The new blocks are assigned to available slots in the cache. (f) New blocks are decompressed into their assigned slot using the WebGPU ZFP decompression kernel.

contrast to Liu et al., BCMC does not have access to the full decompressed volume in a 3D texture. Thus, marking only the blocks containing the isovalue as active is not sufficient, since the data required for their dual grid cells sharing vertices with neighboring blocks would be missing (Fig. 4.35). To ensure the neighbor data are available, BCMC marks a block as active if its range or the union of its range and any of its neighbors' ranges contain the isovalue. The output of the active block selection kernel is the active mask list M_{active}. The list of active block IDs, I_{active}, is computed using a GPU prefix sum and specialized stream compaction, "StreamCompactIDs." The specialized compaction writes the element's index in the active mask list at the output offset (computed in the prefix sum) instead of the element's value, compacting the IDs of the active elements.

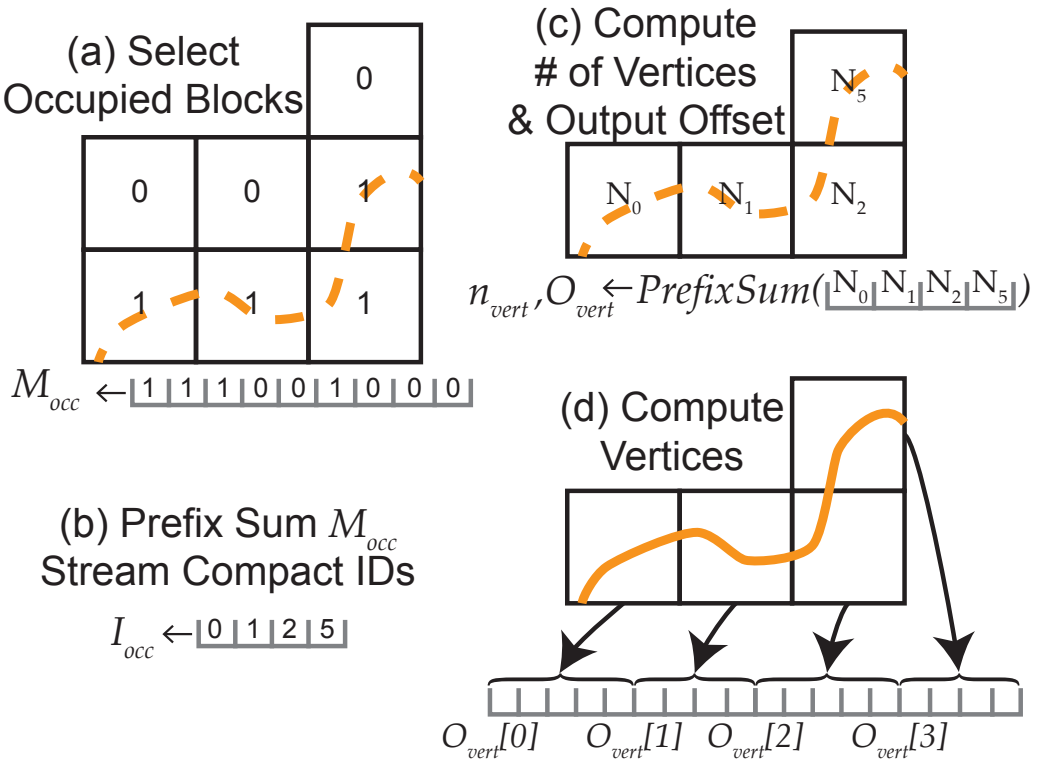


Fig. 4.34: (a,b) The active blocks are filtered down to just those that will output vertices, (c) after which BCMC computes the number of vertices that will be output by each block and (d) outputs them to a single vertex buffer.

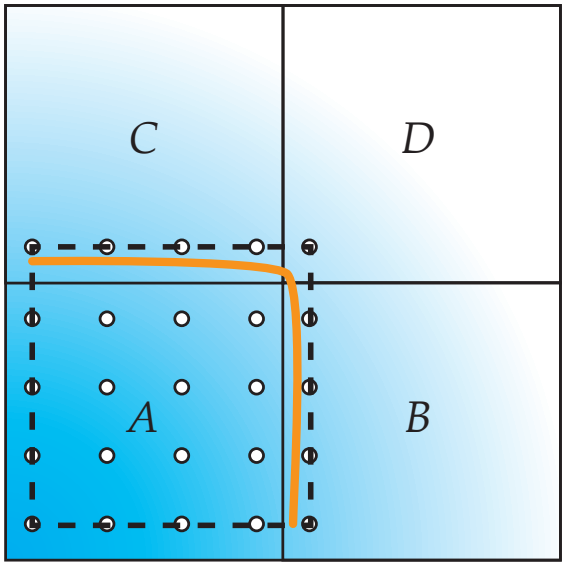


Fig. 4.35: The dual grid of block A (dashed) overlaps its neighbors B, C, D, which must also be decompressed to provide the data required to compute the surface (orange).

4.3.4.2 GPU-Driven LRU Block Cache

Given M_{active} and I_{active}, BCMC determines which active blocks are already in the cache and which must be decompressed and added to it. This task is performed using an entirely GPU-driven LRU cache, to avoid reading back M_{active} or I_{active} to the host and to minimize the amount of serial computation performed in the algorithm.

The cache is comprised of a growable number of "slots," within which a decompressed block can be stored. The initial number of slots is set by the user, e.g., enough to store 10% of the blocks in the volume. The cache uses two arrays to track the status of each slot: S_{age}, which stores the age of the item in the slot, and S_{block}, which stores the ID of the block in the slot or -1 if the slot is empty. An additional array, I_{slot}, stores the ID of the slot occupied by each block, or -1 if the block is not cached. A final array stores the actual data for each slot, containing the decompressed 4^3 blocks.

The GPU parallel cache update proceeds as shown in Algorithm 1 (also see Fig. 4.33). First, each slot's age is incremented in parallel using a compute shader. Next, the list M_{new} is computed, that marks, for each block, if it must be newly decompressed and added to the cache; and the list M_{avail} updated, that marks, for each slot, if it is available. The lists M_{new} and M_{avail} are computed in a compute shader run for each block b that checks if the block is active (i.e., $M_{active}[b] = 1$). If the block is active and cached (i.e., $I_{slot}[b] \neq -1$), the thread resets the age of its slot, marks the slot unavailable in M_{avail}, and marks the block as not new in M_{new}. If the block is active and not cached, it is marked as new. If a block is cached and not active, its slot is marked as available, making it a candidate for eviction. A GPU prefix sum is then performed on M_{new} and M_{avail} to compute the number of new blocks and the number of available slots. If there are no new blocks, the cache update is terminated. If fewer slots are available than are needed, the cache is grown. The new slots added when growing the cache are marked available and assigned a high age.

The "StreamCompactIDs" kernel is used to compute the list of new block IDs, I_{new}, and available slots, I_{avail}. The active list and output offsets come from M_{new}, O_{new} and M_{avail}, O_{avail}, respectively. The available slot ages, K_{age}, are computed with a standard stream compaction using the output offsets O_{avail}. I_{avail} is then sorted in descending order by age using a GPU sort by key, where K_{age} contains the keys. Finally, each new block is assigned a slot s from I_{avail} using a compute shader run for each new block. If a block p was previously

1: **function** UPDATECACHE(M_{active})
2: $IncrementSlotAge(S_{age})$
3: $M_{new}, M_{avail} \leftarrow MarkNewBlocks(M_{active}, S_{age}, I_{slot})$
4: $n_{new}, O_{new} \leftarrow PrefixSum(M_{new})$ ▷ Exit if $n_{new} = 0$
5: $n_{avail}, O_{avail} \leftarrow PrefixSum(M_{avail})$ ▷ Grow if $n_{new} > n_{avail}$
6: $I_{new} \leftarrow StreamCompactIDs(M_{new}, O_{new})$
7: $I_{avail} \leftarrow StreamCompactIDs(M_{avail}, O_{avail})$
8: $K_{age} \leftarrow StreamCompact(M_{avail}, O_{avail}, S_{age})$
9: $SortByKey(I_{avail}, K_{age})$ ▷ Sort available slots by age
10: $AssignSlots(I_{new}, I_{avail}, S_{age}, S_{block}, I_{slot})$
11: $DecompressBlocks(I_{new}, I_{slot})$

Algorithm 1: The GPU-driven cache update algorithm. Each function call corresponds to a compute shader dispatch or data-parallel primitive (prefix sum, stream compact, sort) run on the GPU.

stored in the slot (i.e., $S_{block}[s] \neq -1$), it is evicted by setting $I_{slot}[p] = -1$. The new block is then set as the item in the slot, the slot assigned to the block in I_{slot}, the slot marked unavailable, and the slot age reset to 0.

4.3.4.2.1 Decompression of blocks into cache slots. The WebGPU port of ZFP's CUDA decompressor (Section 4.3.3.4) is used to decompress the data, by modifying it to decompress just the blocks specified in I_{new}. Each new block b is decompressed by a thread on the GPU in a compute shader. The thread writes the 4^3 decompressed block as a row-major array into the cache slot assigned to the block ($I_{slot}[b]$). This approach has the added benefit of eliminating the scattered writes required when outputting the entire volume, improving the write access patterns of each decompression thread.

4.3.4.3 Loading Blocks and Neighbor Voxels into Shared Memory

Data from both the block itself and its neighbors in the $+x/y/z$ directions, with which the block's dual grid shares vertices (Fig. 4.35), are needed to compute the block's vertices. As BCMC does not store the full volume, accessing the neighbor voxels cannot be done with a 3D texture lookup as done by Liu et al. [172]. Instead, BCMC loads the data for the block's dual grid into shared memory to enable fast access to the data by the thread group processing the block. This step is performed by both the vertex output location computation (Section 4.3.4.4) and the final vertex computation (Section 4.3.4.5). In both steps, BCMC assumes the block has neighbors along all axes, i.e., a 4^3 dual grid, and runs a thread group

per block with 64 threads, one thread per dual cell.

First, the thread group loads the 64 voxels of the 4^3 block into shared memory, with each thread loading a voxel. If the block has neighbors to its positive side, the block dimensions are increased along the corresponding axis by one. However, it is not safe to assume the required face, edge, and corner neighbors exist, as only the active blocks are guaranteed to be decompressed and in the cache. For example, a block that is marked active because it is needed by a neighbor may not have any of its $+x/y/z$ neighbors active, and would thus attempt to load invalid data. After tentatively increasing the block's dimensions based on its location in the grid, the thread group tests if the required neighbors are active in M_{active}. If the required face, edge, or corner neighbor in some direction is not active, the block's size along that direction is reset to 4.

The thread group then loads the required neighbor data from the neighbor blocks. The threads responsible for the block's face, edge, and corner voxels are also responsible for loading the neighboring face, edge, and corner voxels, respectively. Finally, the thread group is synchronized using a barrier to wait until the data are loaded.

4.3.4.4 Computing Vertex Output Locations

The next step in BCMC is to determine the output offsets for each block's vertices and the total number of vertices to be output. Before doing so, a compute shader is run over the active blocks to determine the subset that will output vertices (Fig. 4.34a). Each thread group loads a block into shared memory as described in Section 4.3.4.3. Each thread then checks if its dual cell will output a triangle and writes this result to shared memory. If the block will output a triangle, thread 0 marks the block as occupied in M_{occ}; otherwise, it is marked as empty. The list of occupied block IDs, I_{occ}, is found through a GPU prefix sum over M_{occ} and "StreamCompactIDs" (Fig. 4.34b). It is not sufficient to mark unoccupied each block that was marked active as a result of the union of it and one of its neighbors' ranges, as the block's dual cells shared with the neighbor may output vertices that the block is responsible for computing (Fig. 4.35). However, it may be possible to determine this by checking which side the neighbor was on.

BCMC then computes the output offsets for each occupied block's vertices (Fig. 4.34c). After loading the block and neighbor data into shared memory, BCMC proceeds as described

by Liu et al. [172] to compute the vertex output offsets. Each thread computes the number of vertices that will be output by its dual cell and writes this value to shared memory. The thread group then performs a sum reduction to compute the total number of vertices that will be output by the block, which thread 0 writes to global memory. The per block output offsets and total output size are computed by a GPU prefix sum over the per block vertex counts. As suggested by Liu et al., BCMC does not perform a global prefix sum or store the per voxel prefix sums computed within each block.

4.3.4.5 Vertex Computation

Each thread group loads an occupied block into shared memory and computes the number of vertices that will be output by each voxel as before, writing the result to shared memory. The thread group then performs a parallel prefix sum over this shared memory to compute the per voxel output locations relative to the block's global offset. Each thread then computes the vertices for its dual cell and writes them to the global vertex buffer (Fig. 4.34d). Outputting an indexed triangle mesh using Flying Edges [250] within each block, as proposed by Liu et al. [172], is also possible.

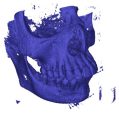
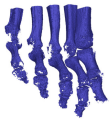
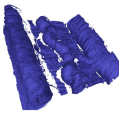
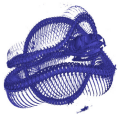
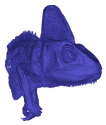
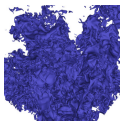
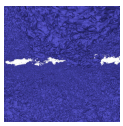
To reduce the memory required to store the vertices, BCMC adopts a compressed vertex format. Vertices are stored relative to the block origin and quantized to 10 bits per coordinate. For each vertex, BCMC stores the three 10-bit coordinates packed into a uint32, along with the block ID as a second uint32, for a total of 8 bytes per vertex. Blocks are indexed in row-major order, providing a simple mapping from ID to 3D position. When rendering, the block position is computed from its ID and used to offset the dequantized vertex.

4.3.5 Performance Evaluation

BCMC is evaluated on nine data sets, varying in size and isosurface topology (Table 4.7). As ZFP supports only floating point data; the nonfloat data sets are converted to 32 bit floating point before compressing them. ZFP is able to compress the data sets to far less than 8 or 16 bits per voxel, and bitrates as low as 2 have been demonstrated to have low impact on quality [171]. The benchmarks are run on a Surface Pro 7 with an integrated GPU with 3.8GB of VRAM and a desktop with an RTX 2070 with 8GB of VRAM.

The isosurfaces used in the evaluation can be classified into "nested" and "turbulent sheet" topologies, and have different cache behaviors. Isosurfaces with nested topologies

Table 4.7: Example isosurfaces on the data sets used in the benchmarks. Isosurfaces are typically sparse, requiring little data to be decompressed and cached to compute each surface, and even fewer blocks to be processed to compute the surface geometry.

Image	Original	Adaptive Prec. & Res.	Isovalue	% Active	Cache	% Occ.	Triangles	Vertex Data
	256^3 67.1MB	4.1MB $\approx \frac{1}{16}$	39	38.9%	26MB	19.8%	2.1M	51MB
	256^3 67.1MB	4.1MB $\approx \frac{1}{16}$	117	11.7%	7.8MB	5.9%	743K	18MB
	{512, 512, 373} 391MB	49.3MB $\approx \frac{1}{8}$	409	22.6%	89MB	9.9%	6.2M	148MB
	512^3 537MB	67MB $\approx \frac{1}{8}$	1.19	15%	81MB	9%	9.5M	229MB
	{832, 832, 494} 1.37GB	86MB $\approx \frac{1}{16}$	868	5%	70MB	3%	6.7M	160MB
	{1024, 1024, 795} 3.33GB	209MB $\approx \frac{1}{16}$	111	5%	167MB	2%	10.2M	245MB
	{1024, 1024, 1080} 4.53GB	283MB $\approx \frac{1}{16}$	21290	4.5%	227MB	1.9%	11.2M	270MB
	1024^3 4.3GB	537MB $\approx \frac{1}{8}$	1.39	26%	1.1GB	12%	67M	1.6GB
	{10240, 7680, 1536} 966GB	199MB $\approx \frac{1}{4854}$	0.936	43%	1.4GB	24%	135M	3.3GB

are typical in MRI and CT scans (Skull, Foot, Backpack, Stag Beetle, Kingsnake, and Chameleon), although they do occur in simulations as well (Plasma). In the nested topology isosurfaces, surfaces at lower values enclose those at higher values. Such isosurfaces can be extremely cache friendly, as after computing a surface at a lower value, a large number of the blocks will be reused for neighboring surfaces, and even distant enclosed ones.

Isosurfaces with turbulent sheet topologies are typically found in simulations where the surface represents a moving interface, e.g., fluids mixing or turbulent flows (Miranda and DNS). In a turbulent sheet topology isosurface, surfaces move along one or more axes of the domain with the isovalue, and do not enclose each other. Moreover, the turbulent nature of the surface results in a large number of blocks being occupied and output triangles. Isosurfaces with turbulent sheet topology tend to be less cache friendly for random isovalues, as different surfaces occupy different and distant regions of the domain, with relatively few shared blocks; however, neighboring isovalues do share blocks as the surface does not make large jumps in the domain.

4.3.5.1 Isosurface Extraction

Three benchmarks are run on each data set, covering different levels of cache friendliness and user interaction modes when exploring a volume. The first benchmark computes 100 random isovalues, representing a cache-unfriendly exploratory use case. The other two benchmarks sweep the isosurface up or down the data set's value range, as may be done when comparing neighboring values in the data, and are relatively cache friendly.

Although the sweep up and down benchmarks may seem to be the same, the cache behaviors of the two differ on the nested isosurfaces. On these data sets, an up sweep will frequently operate in cache, as the number of active blocks decreases and the inner surfaces are likely to require blocks that were already decompressed for the previous containing surface. However, a down sweep is less likely to hit cache, as the surface area of the isosurface increases to cover more blocks, with these blocks less likely to be cached.

As before, the isovalues are sampled over a value range covering surfaces typically of interest to users, excluding noise and values so high as to produce few output triangles. Example configurations for each data set are shown in Table 4.7. The first computation in each benchmark is discarded, as WebGPU has a high first launch overhead. The average

cache hit rates and extraction time for the benchmarks are shown in Table 4.8.

On the nested isosurface data sets, high cache hit rates are observed for all three benchmarks, with sweep down performing the worst on average due to the lower cache hit rate. A lower cache hit rate means a larger number of blocks must be decompressed to compute each surface, impacting compute time. It is interesting to note that random achieves a higher hit rate than sweep down. After computing some sampling of the nested isosurfaces in random, the cache contains the most frequently used blocks across these surfaces, allowing for high cache hit rates and better performance.

In contrast, on the turbulent sheet data sets, high cache hit rates are observed on the sweep benchmarks and low hit rates on the random benchmark. When computing random isovalues, the surface can make large jumps in the domain, covering entirely different regions of the volume. This issue is exacerbated by the high resolution of the data sets and the turbulent nature of the computed isosurfaces, leading to a large number of blocks covered by each surface with few shared between them. As a result, large numbers of blocks must be decompressed each time, severely impacting compute time. However, the surface does not move significantly when sweeping the isovalue up or down, resulting in high cache hit rates and better performance.

As indicated by the cache hit rates on the nested isosurface data sets, so few new blocks must be decompressed to compute each new surface (on average, $< 0.5\%$ of the total blocks in the data set) that decompression does not occupy a large portion of compute time. The bulk of time is spent in the first pass to select the active blocks and the mark new items step of the cache update, the latter of which then determines no new items are to be added.

In contrast, on the turbulent sheet data sets, a larger percentage of time is spent in decompression due to the higher miss rate and higher absolute number of active blocks. On the random benchmarks, decompression occupies an average of 63% of compute time, with an average of 12% of blocks decompressed for each new surface. On the sweep benchmarks, time is more evenly divided among decompression, finding active blocks, and updating the cache, with an average of 0.8% of blocks decompressed for each new surface.

When new items need to be added to the cache, the bulk of the cache update time is spent sorting the available slots by age. Highly optimized libraries such as Thrust [15] and VTK-m [196] that provide optimized parallel primitives are not available in WebGPU, and

Table 4.8: Average cache hit rates and isosurface computation performance for the data sets and benchmarks performed. Isosurfaces are typically sparse and occupy neighboring regions of the domain, leading to high cache rates for the sweep benchmarks. The topology of the nested isosurfaces also allows for high hit rates on random isovalues due to the high amount of overlap, whereas the turbulent sheet isosurfaces do not overlap as much and see far lower hit rates. Although the cache space required for the Miranda and DNS is small enough to fit in the Surface Pro's 3.8GB VRAM, for most isovalues, the cumulative size of the cache and output vertex data is not.

(a) Random.

Data Set	Hit Rate	RTX 2070	Surface Pro 7
Skull	98.4%	57.7ms	99.8ms
Foot	98.9%	57.1ms	107.9ms
Backpack	99.4%	68.1ms	251.0ms
Plasma	96.4%	107.2ms	318.6ms
Stag Beetle	98.2%	123.1ms	322.1ms
Kingsnake	99.2%	212.6ms	926.5ms
Chameleon	90.9%	371.6ms	1342ms
Miranda	46.8%	2218ms	(oom)
DNS	62.9%	2632ms	(oom)

(b) Sweep up.

Data Set	Hit Rate	RTX 2070	Surface Pro 7
Skull	100%	54.4ms	85.1ms
Foot	100%	58.4ms	99.2ms
Backpack	100%	99.1ms	216.3ms
Plasma	99.8%	94.3ms	265.3ms
Stag Beetle	99.6%	125.1ms	337.8ms
Kingsnake	100%	185.3ms	811.1ms
Chameleon	98.9%	348.7ms	1470ms
Miranda	95.7%	550.5ms	(oom)
DNS	98.4%	972ms	(oom)

(c) Sweep down.

Data Set	Hit Rate	RTX 2070	Surface Pro 7
Skull	94.8%	93.1ms	161.5ms
Foot	97.4%	90.1ms	164.4ms
Backpack	96.8%	109.7ms	349.3ms
Plasma	98.5%	126.1ms	431.8ms
Stag Beetle	92.0%	151.2ms	464.5ms
Kingsnake	97.0%	284.5ms	1306ms
Chameleon	96.6%	337.1ms	1482ms
Miranda	95.2%	549.6ms	(oom)
DNS	97.9%	841.3ms	(oom)

the unoptimized sort by key implemented in BCMC is a bottleneck when processing many items.

4.3.5.2 Rendering Performance

Table 4.9 reports the rendering performance achieved on the example isosurfaces computed in Table 4.7. BCMC outputs a triangle soup in a compressed vertex format, which is rendered with WebGPU. On both the Surface Pro 7 and RTX 2070, the output mesh can be rendered in real-time, even for large isosurfaces. Although the rendering modality is relatively simple, these results are encouraging for large-scale data visualization in general, demonstrating that WebGPU can achieve high framerates for large geometry. The quality of the isosurfaces is primarily dependent on the reconstruction accuracy of the compression method, which has demonstrated high accuracy even at high compression ratios [119, 120, 171]. As the vertex buffer occupies a substantial amount of memory, it would be valuable to explore applying implicit isosurface raycasting methods on top of the existing block structure. The blocks can be seen as a macrocell grid [213] for space-skipping, and implicit isosurface ray tracing performed within these blocks.

4.4 Summary

Supporting the ever-growing size of scientific data sets poses a continuing challenge to post hoc visualization. Moreover, it is not sufficient to support processing the data only at full resolution on HPC systems. To make massive data sets accessible to scientists, new adaptive and multiresolution methods must be developed that enable exploring such data

Table 4.9: Rendering performance for the isosurfaces shown in Table 4.7. WebGPU is capable of interactive rendering of large triangle meshes even on lightweight clients.

Data Set	Triangles	RTX 2070 (FPS)	Surface Pro 7 (FPS)
Skull	2.1M	180	44
Foot	743K	174	89
Backpack	6.2M	146	25
Plasma	9.5M	118	24
Stag Beetle	6.7M	139	19
Kingsnake	10.2M	128	24
Chameleon	11.2M	123	19
Miranda	67M	53	(oom)
DNS	135M	36	(oom)

interactively in constrained environments, such as VR and the web browser. This chapter has presented three works that address these challenges. The distributed framebuffer provides a flexible asynchronous tile-based processing pipeline for distributed rendering, enabling renderers to work across the spectrum of memory and compute scaling. The scalability and flexibility of the DFB is demonstrated on interactive distributed data-, image-, and hybrid-parallel rendering of massive data sets at full resolution with high-quality illumination effects. The VR neuron tracing tool was developed through a collaborative design study with neuronatomists to enable neuron tracing on large connectomics data sets in VR. To support immersive interactive exploration of large data in VR, VRNT employs a low-latency on-demand data processing approach based on a page-based processing system and multiresolution data layout. A pilot study was conducted with expert neuronatomists that demonstrated that the immersive interface and intuitive interaction modes provided by VRNT improved neuron tracing performance and understanding. Finally, a fully GPU-driven algorithm for isosurface extraction on block-compressed volumes, BCMC, was proposed to enable accessible interactive isosurface extraction on massive data sets in the web browser. BCMC decompresses and caches volume blocks as needed to reduce its memory footprint and moves all computation to the GPU to achieve fast isosurface extraction. BCMC achieves interactive visualization in the browser on the same 1TB DNS data set that previously required a small set of nodes on an HPC cluster to render.

It is important to note that the techniques discussed are applicable beyond the contexts that they were initially proposed in. For example, the scalable distributed rendering provided by the DFB is also valuable for in situ rendering, as discussed in the next chapter. The low-latency on-demand processing strategies developed to ensure high performance in VR are also applicable to traditional desktop applications that wish to process massive data sets while maintaining interactivity. Finally, BCMC is just as valuable outside the browser in native applications, where it can be used for high-performance out-of-core processing of massive volumes on GPUs. Bringing these approaches for adaptive and multiresolution processing together provides a compelling view of future exascale post hoc visualization tools, where massive data sets can be explored on workstations or laptops without major sacrifices in interactivity or quality. There also remain exciting avenues to further develop the work discussed in this chapter.

CHAPTER 5

IN SITU VISUALIZATION

Although the compute capability available to simulations has grown rapidly, the I/O bandwidth available to save the simulated data has not. This widening gap between FLOPs and I/O bandwidth means that far more data can be simulated than can possibly be saved to disk for post hoc visualization, impacting the amount of scientific insight that can be gained from massive simulations. In situ visualization has been proposed as one approach to address this issue, and has been identified as a key technology to enable science at exascale [5]. By running the visualization tasks during the simulation, in situ visualization enables the visualization to access the data at a much higher resolution or temporal frequency than could be saved to disk. However, a key concern when moving the visualization to run in situ is the impact that the visualization will have on the simulation. This chapter discusses work toward addressing this concern.

Rendering images is a common task performed in situ, as the resulting images are small in size compared to the full data and can be saved frequently and viewed post hoc. The rendered images have been used in a number of ways to enable different forms of post hoc visualization. A common use case is to produce static images [272, 273] or movies [74]. To provide some ability to interactively adjust the visualization parameters post hoc, recent approaches have proposed rendering explorable image databases [7, 80, 142, 298, 321]. Creating image databases requires rendering a large number of images, covering a sweep of the desired visualization parameter space. Interactive approaches have also been proposed that allow scientists to connect a remote visualization client to the simulation [129, 235, 280, 282, 319]. Although these applications differ widely in the number and type of images rendered, they fundamentally require a highly scalable and flexible distributed rendering technique to produce the images without incurring a significant impact on the simulation. Section 5.1 discusses how the Distributed FrameBuffer (Section 4.1, [281]) can

be applied to enable scalable in situ rendering of image databases [275].

Although tightly coupled in situ visualization provides a clear benefit by allowing the simulation and visualization to share memory directly, not all visualization tasks are well suited to a tightly coupled configuration [17,74,108,195,235,323,325] The visualization task may require access to more data than are stored on each rank, may not scale as well as the simulation, may require more memory than is available on each node after considering the space already occupied by the simulation, or may need more time to execute than the simulation has budgeted for visualization.

One approach to address the first two issues is to adopt a data-staging in situ visualization approach. Data-staging approaches move the visualization to run on a subset of ranks that collect the required simulation data from the other ranks [17]. As a result, the visualization task has access to more data than each rank's individual piece of the domain and is run at a smaller scale where it may actually perform better. Data staging for in situ visualization follows a similar communication pattern to data aggregation for two-phase I/O, allowing it to be easily integrated into simulations by integrating it within the I/O pipeline [177, 284, 285]. The simulation can then simply adjust the frequency at which it "writes" the data to run the in situ pipeline that transforms the data into the desired visualization outputs. This ease of adoption makes data staging approaches a valuable option even for tasks that can be run in a tightly coupled configuration. Section 5.2 discusses work in progress on integrating support for in situ visualization into the adaptive spatially aware two-phase I/O pipeline presented in Section 3.2.

Loosely coupled in situ visualization provides the flexibility to address all four challenges, making it a compelling option at scale [148,149]. Loosely coupled approaches run the visualization in a separate process from the simulation, and optionally on separate compute nodes, to minimize its impact on the simulation. Running the visualization in a separate process also gives it access to additional compute time without impacting the simulation, as the visualization can continue processing while the simulation proceeds with the next time step. Running loosely coupled in situ visualization on nodes other than those used by the simulation gives the visualization access to additional compute and memory capacity while further reducing its impact on the simulation. As with data staging approaches, loosely coupled visualization can be run at smaller scales than the simulation

to avoid scalability issues. Section 5.3 discusses work on data transfer strategies to enable portable, low overhead loosely coupled in situ visualization [278,280,282].

5.1 Accelerating In Situ Rendering of Image Databases with the Distributed FrameBuffer

Rendering the image databases used by explorable image database methods [7,80,142, 298,321] involves in situ rendering of a large number of images, corresponding to a sweep of the visualization parameter space. The visualization parameter space covers multiple viewpoints, different isovalues, transfer functions, etc., requiring hundreds to thousands of individual images to be rendered in situ. The resulting image database can then be interactively explored post hoc by adjusting the selected parameter set to combine the images in the database to create the desired visualization.

The work distribution over the ranks when rendering the image database is likely to vary for each image, as each uses a different set of visualization parameters. Whereas for one image some ranks may have little to no rendering work if their data are not visible for the given parameters, the next image may be a close up view of just their data, making them a bottleneck. One approach to address this issue is to render multiple images in parallel [1]. Rather than having underutilized nodes wait on those that have a higher workload to complete the image, as would occur when rendering the images serially, they can instead continue on to render other images for the database, where they in turn might have a higher workload. This approach allows the nodes to automatically adjust the workload to complete the image database faster, without requiring a complex load balancing or data movement strategy.

This section discusses how the Distributed FrameBuffer [281] (Section 4.1) can be applied to accelerate in situ rendering of image databases [275]. OSPRay [291], the ray tracing framework within which the DFB is implemented, recently added support for asynchronous single-node rendering, which the DFB is extended to support as well. Asynchronous rendering enables applications to render multiple images with different scene configurations in parallel, as is needed to accelerate the rendering of large image databases. To render multiple images in parallel, the application must create one framebuffer for each active render, along with different OSPRay scene objects for each unique configuration of the

scene to be rendered. Rendering an image database frequently involves rendering the same data with different appearance information, for example, a volume with multiple transfer functions. OSPRay separates the underlying data (i.e., the volume) from its appearance information (i.e., the transfer function), allowing multiple asynchronous renders to share the same underlying data to reduce memory overhead.

The DFB's applicability for accelerating image database rendering is demonstrated through a mini-app called mini-cinema[1] that outputs Cinema-style image databases [7]. Mini-cinema supports volume rendering and isosurface rendering (Fig. 5.1). Isosurfaces can be rendered explicitly by extracting triangles using VTK or by using OSPRay's implicit isosurfaces to reduce memory overhead. The application also supports transparent isosurfaces and can compute ambient occlusion on the isosurface by duplicating it across the ranks to provide ghost zones. A new OSPRay scene is created for each unique set of appearance configurations, applying the new appearance information to the underlying shared volume and geometry data. As the new objects created for each reference only the existing data, the largest amount of additional memory required for each asynchronous render is the framebuffer to store the output image. The mini-app can be configured to render a specific number of images in flight, to limit memory consumption and avoid oversubscribing those nodes that are more heavily utilized for multiple frames. After an asynchronous render has completed, the image is written to disk on a background thread.

A set of benchmarks are run using mini-cinema to measure the impact of varying the number of images in flight on total task completion time. The benchmarks are run using the 1024^3 Miranda data set [56] and renders two isosurfaces, containing 62.47M and 67.65M triangles respectively, combined with the volume data over a 250 camera position orbit for a total of 500 images. Each image rendered is 1024×1024. The benchmark is run on 16 Stampede2 Skylake Xeon (SKX) nodes with one rank per node, and the number of images in flight varied from 1 to 128. Each Stampede2 SKX node has two Intel Xeon Platinum 8160 CPUs and 192GB of RAM. When run with one image in flight, the image set is rendered in 23.7s, when run with up to four images in flight it is rendered in 14.8s, an improvement of 8.9s. Increasing the allowed number of images in flight further does not positively or

[1] `https://github.com/Twinklebear/mini-cinema`

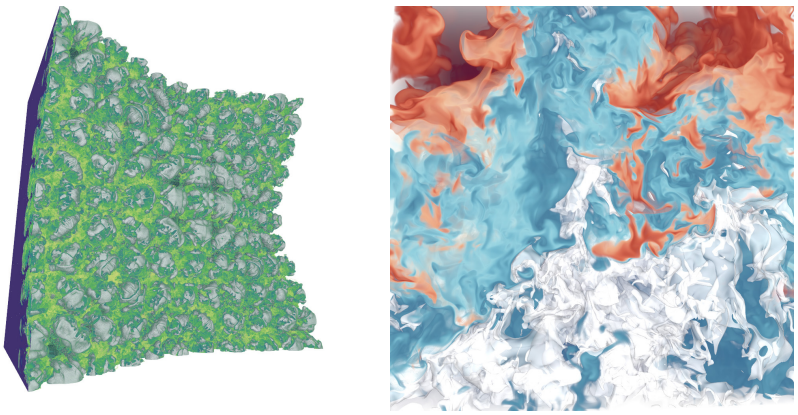


Fig. 5.1: Example images rendered using the image database rendering test app, mini-cinema. The app uses OSPRay's asynchronous rendering API and the DFB to render multiple images simultaneously to better utilize the nodes. The Richtmyer-Meshkov [54] (left) uses ghost zones for ambient occlusion on the isosurface. The Miranda [56] (right) combines a semitransparent isosurface with the volume.

negatively affect the total task time, likely due to the images completing quickly enough that not many more than four images are in flight at a time. The best number of images in flight is highly dependent on the rendering configuration. More expensive individual images may result in more in flight at a time, and lead to reduced performance if oversubscribing the nodes.

5.2 Data Staging In Situ Within an Adaptive Spatially Aware Two-Phase I/O Pipeline

In a data-staging in situ pipeline, visualization tasks are run on a subset of ranks that collect the data to be processed from the other ranks, in the same manner that the aggregator ranks collect data to be written to disk from the ranks in their aggregation subgroup in a two-phase I/O pipeline. This similarity has made the I/O library a common path for integrating support for in situ visualization, as done by ADIOS [177] and GLEAN [284, 285]. From a practical standpoint, integrating in situ visualization into the I/O library enables simulations already using the I/O library to adopt in situ visualization without any code modification, making it a valuable approach even for visualization tasks that are suited to tightly coupled in situ use cases. I/O library-based data staging methods are typically paired with a distributed data query system, for example, PreDatA [326], DataSpaces [68], or FlexPath [63]. A distributed data query system is required to support visualization tasks

that need access to more data than are available locally on each aggregator rank during I/O. I/O library-based data staging in situ visualization has been used effectively in practice to enable scalable in situ visualization. The scalability of such approaches is dependent on the I/O library's scalability; however, the I/O library is often not the bottleneck in the pipeline. The performance of the in situ pipeline is typically limited by the scalability of the visualization tasks being run and the amount of data they must fetch from other ranks.

To ease integration of in situ visualization into simulations using the adaptive, spatially aware two-phase I/O approach proposed in this dissertation (Section 3.2), it is similarly possible to integrate support for in situ visualization within the I/O pipeline. In situ tasks that require only local data can be run on the ranks prior to data aggregation, while those that require access to more data or perform better at smaller scales can be run on the aggregators after the construction of the BAT data layout, as shown in Fig. 5.2. Visualization tasks run on the aggregators can then leverage the BAT layout to accelerate spatial and attribute filtering. In situ visualization tasks are also able to leverage the same data query system used for reads to provide scalable distributed data access in situ. To fetch data from other ranks, visualization tasks send spatial queries to the rank with the desired data, as done for reads (Section 3.2.2). Data queries can optionally include attribute filters to reduce the amount of data that are transferred back to the requesting rank.

Integrating in situ visualization into the adaptive, spatially aware I/O pipeline proposed

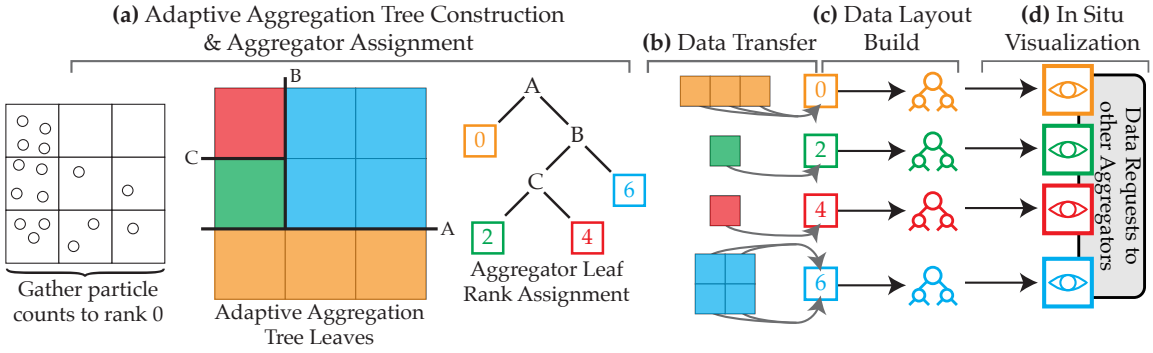


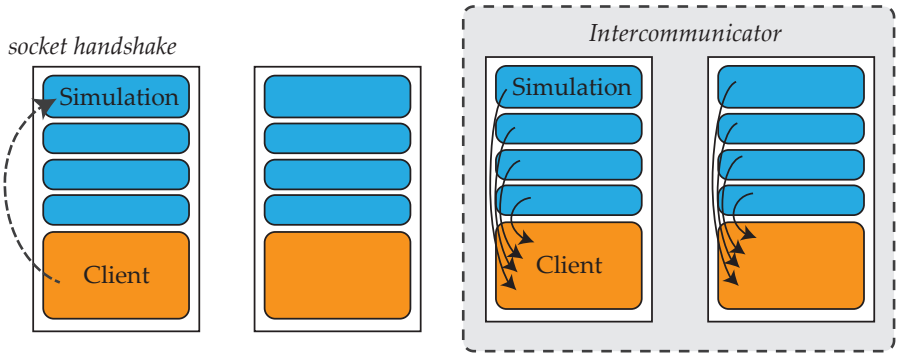
Fig. 5.2: The spatially aware two-phase I/O pipelines can be modified to run in situ visualization in a data staging configuration, by running the visualization tasks on the aggregators. The above illustration demonstrates such an integration within the aggregation tree two-phase I/O pipeline. The same method of fetching data from read aggregators used during reads can also be used to enable distributed data access for in situ visualization tasks. Moreover, the BAT layout built during I/O is available on each aggregator to accelerate spatial and attribute queries.

in this dissertation provides some additional benefits over existing approaches in nonadaptive or spatially unaware I/O pipelines. First, the I/O load balancing performed by the aggregation tree implicitly load balances visualization tasks that scale with the particle count by computing aggregation subgroups that contain roughly equal numbers of particles. Prior work has demonstrated that load balancing can improve performance of tightly coupled in situ by leveraging the simulation's built-in load balancer [231]. Similar performance benefits could be expected when using the aggregation tree for data staging in situ. Moreover, the aggregation tree is not tied to a specific simulation's load balancing strategy, enabling load balanced in situ visualization in any simulation using the method. Furthermore, by adopting a spatially aware aggregation strategy, the presented I/O approaches ensure that the data on each aggregator form a convex subregion. The data access patterns of visualizations tasks frequently exhibit spatial locality, or when run distributed, decompose the data into convex subregions for processing. By providing a convex subregion on each aggregator, it is possible to reduce the amount of data that would have otherwise been fetched from other ranks to compute the visualization.

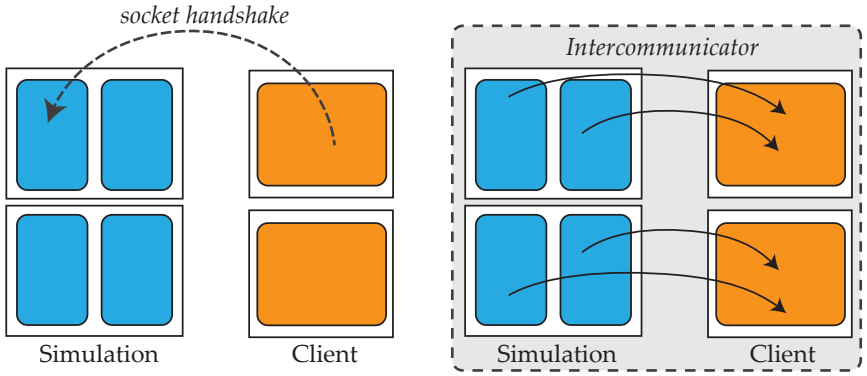
5.3 Low Overhead Loosely Coupled In Situ Visualization

Separating the visualization and simulation into separate processes provides several benefits at scale [148], although such separation is not without its own challenges. A common challenge when adopting loosely coupled in situ visualization is the task of efficiently transferring data between the simulation and visualization. As the two are no longer run in the same process, some form of synchronization and data communication is required, either via shared memory or the network. Furthermore, when run in an $M : N$ configuration of M simulation ranks and N visualization ranks where $M > N$, it may be desirable to restructure the data from the simulation's M regions down to N regions to provide the visualization task a simpler data distribution to process. This sections discusses work on two data transport models for low-overhead loosely coupled in situ visualization.

Two methods for transferring the data from the simulation to the visualization are discussed: a data restructuring model and a simulation-oblivious model. Both models share common functionality to support on-demand execution of the visualization, flexible $M : N$ configurations, and to reduce memory overhead (Fig. 5.3). To support on-demand



(a) Shared node configuration.



(b) Separate node or resource configuration.

Fig. 5.3: Visualization clients using the restructuring or simulation-oblivious data transport models can be run on the same nodes as the simulation (a), and thus take advantage of shared memory data transfers at the cost of oversubscription, or on separate nodes, where the applications will not impact each other at the cost of communicating over the network (b). (a) shows a $4:1$ configuration run on two nodes, (b) shows a $2:1$ configuration on four nodes.

execution, visualization clients using the presented methods can connect and disconnect as desired from the simulation using a socket handshake. The clients can be started manually by the user [108] or automatically using, for example, in situ triggers [161]. To allow for a wide range of $M:N$ configurations, neither approach imposes requirements as to where the visualization clients are run. The clients can be run on the same nodes as the simulation, distinct nodes on the same HPC resource, or an entirely separate HPC resource. Finally, to reduce memory overhead, the approaches do not buffer the previous time step to send to clients that request data while the next time step is being computed. Instead, clients requesting data block until the next time step is complete and can be sent directly from the simulation's data buffers. Each approach is implemented in two lightweight libraries, one

that is linked into the simulation and another that is linked into the visualization.

Section 5.3.1 discusses a spatially aware data restructuring model for particle data [282]. The restructuring model treats the simulation as a data server that accepts spatial queries from the visualization clients. The clients are thus free to query data from the simulation to adjust the data distribution as needed for the computation being performed, e.g., to each retrieve a single convex subregion on each client. The restructured data distribution matches that which would be used when running the visualization tasks post hoc, simplifying the process of migrating visualization tasks to run in situ.

However, restructuring the data to match the visualization client's parallelism comes at some cost. Moreover, as restructuring is performed while transferring data to the visualization clients, the simulation must wait for the restructuring to complete before moving on to the next time step. Section 5.3.2 discusses a simulation-oblivious data transfer model that minimizes the impact of data transfer on the simulation [275, 280]. The simulation-oblivious model treats each rank as an opaque block of bytes. These blocks are assigned to the visualization clients using a simple rank-based mapping. By eliminating data restructuring during data transfer, the simulation-oblivious model is able to achieve high data transfer performance and minimize the impact of the visualization on the simulation, while leaving the visualization clients free to restructure the data after the transfer if desired.

The models presented are evaluated in Section 5.3.3. The simulation-oblivious model is evaluated on in situ visualization of molecular dynamics and turbulent flow simulations to study its impact on the simulation when run in different proximity configurations. The evaluation demonstrates that the simulation-oblivious model provides a compelling option to achieve minimal impact in situ visualization. Finally, the data transfer costs of the restructuring and simulation-oblivious data transport models are compared with each other and existing restructuring approaches to evaluate the performance cost of restructuring.

5.3.1 A Spatially Aware Data Restructuring Model for Particle Simulations

The restructuring-based data transfer model for particle simulations treats the simulation as a data server that can respond to spatial queries from visualization clients (Fig. 5.4). This model allows visualization clients to dictate how data are distributed over the visualization

ranks, and to query additional neighbor data from the simulation if needed.

5.3.1.1 On-Demand Visualization Client Connection

To support on-demand connection between the visualization and simulation without losing MPI's support for high-speed networks and shared memory data transfers, the restructuring model uses facilities in MPI for creating communicators that connect separate processes together, `MPI_Comm_accept` and `MPI_Comm_connect`, along with a socket handshake (Fig. 5.3). After initializing the simulation, a background thread is launched on rank 0 of the simulation that opens a socket to listen for incoming client connections on a specified port. Visualization clients can connect to the simulation by first opening an MPI port to establish a communicator on. Each time the visualization client wishes to query data, rank 0 of the client connects to the simulation's socket and sends over its MPI port information. The MPI port information received on the simulation is pushed onto a list of pending data requests to be sent the next time step when it is ready.

After each time step, the simulation checks if any clients have requested data. For each data request, the simulation checks if an MPI communicator has already been established with the client by looking up its MPI port name in a hash map of established communicators. If a communicator was already established between the simulation and client, the processes proceed to handle the data query as described in the following section. If not, the processes must first establish an MPI communicator to be used for data transfers between them. To do so, the simulation connects back to the client's MPI port by calling `MPI_Comm_connect`; the clients are waiting in a matching call to `MPI_Comm_accept` for the simulation to connect back to them.

5.3.1.2 Data Query

Each query begins with rank 0 of the simulation broadcasting the global bounds of the simulation domain to the visualization client. The visualization client can then decompose the domain into blocks and assign its subregions to the visualization ranks as desired. Each client rank retrieves the data within their assigned subregions by sending one or more 3D bounding boxes to the simulation ranks. The simulation ranks loop through the client ranks and receive the desired box queries from each. Each simulation rank then determines which of its particles are contained in the box and sends them back to the client. To reduce memory

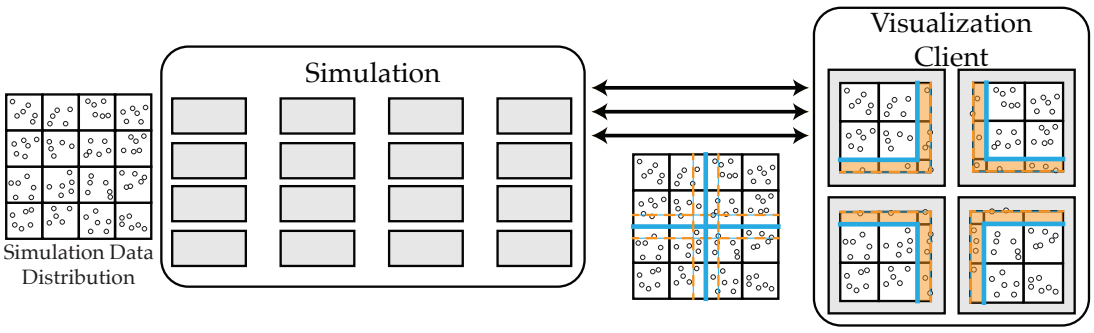


Fig. 5.4: Data are restructured from the simulation to the client's desired distribution during data transfer. Each client rank requests a spatial subregion from the simulation and receives back the contained particles. Here the visualization client partitions the domain among four ranks, of which each owns some subregion (blue lines), and each requires a ghost zone around the subregion (orange dashed lines and highlight). Each rank requests the entire bounding box of its subregion and the ghost zone from the simulation to fetch its data.

overhead, this determination is done by performing an in-place sort on the particles to partition the set of particles into two subarrays. The subset of particles that are contained in the query box can then be sent directly from this sorted buffer to the client.

Although this approach allows the visualization clients to set the data distribution as desired, the data query and restructuring process takes some time. The cost of this step scales with the number of simulation ranks and visualization ranks. The simulation must wait for this process to complete before continuing on to the next step, which translates to a direct performance impact on the simulation. Although constructing additional hierarchies on the data to accelerate spatial queries or using MPI asynchronous messaging to handle multiple requests at a time could improve performance, both approaches would require additional memory, to store either the acceleration structure or the temporary buffers being referenced by the asynchronous messages. Instead, it is possible to remove data restructuring from the in situ data transfer layer entirely, as discussed in the following section.

5.3.2 A Simulation-Oblivious Data Transport Model

The simulation-oblivious data model treats each simulation rank as an opaque block of bytes, the interpretation of which is left up to the simulation and client. Specifically, the model does not inspect, adjust, or restructure the input data distribution provided by the simulation, but instead simply forwards each rank's data to the assigned client. Each client

rank receives one simulation state from each simulation rank it is assigned to process data from, allowing for $M : N$ configurations (see Fig. 5.5).

If the simulation and visualization differ in their parallelism model (i.e., MPI-only vs. MPI + threads) or scalability, an $M : N$ configuration can be used to run each in its ideal configuration. In a $1 : 1$ configuration, each client is assigned a single simulation rank's data. When $M > N$ each client is assigned data from $\frac{M}{N}$ simulation ranks, any remainder is assigned evenly among the clients. After the data have been transferred, the client ranks can restructure the data if desired, or use the distinct set of states directly. Although this oblivious model supports only $M \geq N$, since it does not support splitting a rank's data to restructure it, it is typically the case that the simulation is run at the same or higher level of parallelism than the visualization. It is important to note that, depending on the simulation and visualization configuration, it may not be possible for each individual client rank to combine the received set of simulation states into a single convex region.

Although a somewhat primitive data model, this simplified view provides some desirable benefits over restructuring approaches. Removing the restructuring process from the data transfer step reduces computational cost, allowing the simulation-oblivious approach to achieve high network utilization and to reduce the amount of time the simulation spends performing data transfers. This approach also eases integration into simulations with more exotic mesh types or primitives that may not be supported by the standard restructuring strategies. From a practical standpoint, this approach does not rely on complex distributed data models or libraries, and can be easily deployed on a variety of systems, making it useful for lightweight in situ integrations, or as the underlying data transport layer for in full-featured systems (e.g., SENSEI [10]). Moreover, this simulation-oblivious approach does not preclude the use of data restructuring on the client. After the data have been transferred, the client ranks can make use of a distributed data query system [63, 68, 326] to restructure the data as desired without impacting the simulation.

5.3.2.1 Portable Communication Between the Simulation and Client

As in the restructuring based data transfer model, the simulation-oblivious model uses a socket handshake to establish an MPI communicator between the simulation and visualization processes (Fig. 5.3). To improve portability, the simulation-oblivious model

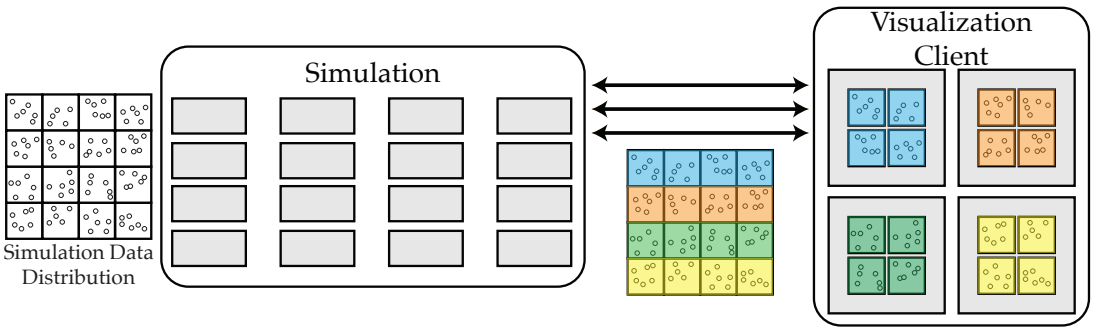


Fig. 5.5: A $16:4$ mapping of simulation data to visualization clients. With $M > N$ each rank is assigned $\frac{M}{N}$ simulation states, with any remainder distributed among the clients. Data are not restructured during transfer, and each client receives $\frac{M}{N}$ distinct simulation states to process.

provides an additional socket-based fallback intercommunicator. When attempting to establish the MPI intercommunicator, both the simulation and client test if the `MPI_Open_port` API is available. If the API is not available, as is the case on Theta at Argonne National Laboratory, the processes fall back to a socket-based intercommunicator; otherwise, an MPI intercommunicator is established as described previously.

To establish the socket-based intercommunicator, each client opens a socket and listens for connections from the simulation. Rank 0 of the simulation broadcasts rank 0 of the client's hostname and port number to the other ranks, which each connect to client rank 0 and receive the hostname and port numbers for the other clients. The simulation ranks then connect to the remaining clients to establish an all-to-all socket intercommunicator. When setting up a connection, the simulation rank sends its rank number to the client, which sends back its own rank number. Each socket in the communicator on a rank is indexed by the rank number of the rank on the other end to provide a send and receive API identical to MPI where messages are sent to or received from a specific rank by its rank number. Finally, to avoid flooding each client with incoming connections from a large simulation run, the simulation ranks' connection requests are rate limited. Without rate limiting, the OS would see the large number of incoming connections as a network flooding attack. The socket intercommunicator also enables running the clients on entirely different hardware and software stacks or HPC resources. Performing the in situ visualization on an entirely distinct cluster can be valuable for sensitive or time-critical applications, as discussed by Ellsworth et al. [74].

5.3.2.2 $M : N$ Asynchronous Data Query

After establishing the intercommunicator, the client can request to receive data from the simulation. The simulation probes for incoming messages from rank 0 of the client after each time step when calling `libISProcess`. The client can request to receive a new time step or to disconnect from the simulation. If a new message is received, it is broadcast to the other simulation ranks and processed collectively.

Data are transferred from the simulation ranks to the client ranks using the oblivious data model and $M : N$ mapping discussed above. The simulation-oblivious model first computes N groups of $\frac{M}{N} : 1$ simulation to client groupings to establish the mapping from simulation ranks to client ranks. Each such group transfers data to the client rank independently and in parallel to the others. When using the MPI intercommunicator, data are transferred using point-to-point communication. When using the socket intercommunicator, data are transferred using the socket connection established on each simulation rank to the assigned client. In contrast to the data query step of the restructuring approach (Section 5.3.1.2), the simulation-oblivious approach avoids global all-to-all communication or synchronization and scales well with the number of simulation and client ranks.

```
// Set the world bounds (per rank local/ghost bounds identical)
void libISSetWorldBounds(libISSimState *state, const libISBox3f box);

// Convenience method to set a 3D regular grid field
void libISSetField(libISSimState *state, const char *fieldName,
    const uint64_t dimensions[3], const libISDType type, const void *data);

// Convenience method to set an array of local and optional ghost particles
void libISSetParticles(libISSimState *state, const uint64_t numParticles,
    const uint64_t numGhostParticles, const uint64_t particleStride,
    const void *data);

// Set an arbitrary 1D buffer of data
void libISSetBuffer(libISSimState *state, const char *bufferName,
    const uint64_t size, const void *data);

// Call after each time step to send data to any clients
void libISProcess(const libISSimState *state);
```

Listing 7: The simulation interface is used to configure a simulation state object, which stores pointers to the rank's local data. Convenience methods are provided for regular 3D fields and particles; arbitrary 1D buffers of data can be sent via the buffer API.

5.3.2.3 The Simulation Interface

To allow integration into a wide range of simulations, the presented simulation-oblivious method is implemented in a library that exposes a C interface, libIS-sim, which is also accessible through a Fortran wrapper. Simulations begin listening for clients by calling `libISInit` and passing the port number to listen on. To make data available to clients, each rank configures a `libISSimState` that stores pointers to the simulation data and metadata describing it (Listing 7). The interface provides convenience wrappers for setting 3D regular grid fields and particles, along with a generic API to pass arbitrary 1D buffers of data. Internally, 3D fields and particles are treated the same as 1D buffers, since the data are not inspected or restructured when sent to clients. Simulations using more complex data representations, such as unstructured meshes, octree AMR, or block-structured AMR, can use raw buffers to send data to clients. When sending raw buffers, the clients must be tailored to the simulation and know how to interpret and process the incoming data. The `libISProcess` function is called collectively by the simulation ranks to send data to any clients that have requested to receive the latest time step.

5.3.2.4 The Client Interface

Similarly, to support visualization software that is commonly implemented in C++, clients link a C++ library, libIS-client, that implements the simulation-oblivious data transfer model (Listing 8). Clients first connect to the simulation by calling `connect`, after which they can query data using the blocking or asynchronous API. Once the desired analysis has completed, the client can disconnect and exit. The client can also check if the simulation has quit by passing an optional parameter to the API calls, or explicitly checking `sim_connected`. Each client receives a vector of simulation states, containing the $\frac{M}{N}$ simulation ranks the client was assigned to receive data from. Each state contains the 3D and 1D buffers sent by the simulation, associated with their name, along with any particles.

5.3.2.5 Integration Within Existing In Situ Systems

The SENSEI project [10] was begun with the goal of addressing the portability and reusability challenges introduced by the growing number of in situ frameworks. SENSEI provides a unified interface for simulations that can map to various in situ library backends, including tightly coupled, data staging, and loosely coupled modes (e.g., ParaView Cata-

lyst [78], VisIt LibSim [302], ADIOS [177]). A SENSEI in situ system consists of a *bridge*, *data adaptor*, and *analysis adaptor*. The first two are tailored to the simulation and responsible for converting its data representation to the VTK representation used by SENSEI. The VTK representation is then passed to an analysis adaptor, which runs the desired in situ visualization. The library implementing the simulation-oblivious approach also provides an analysis adaptor and an in situ visualization execution application, to transparently provide loosely coupled in situ visualization in SENSEI. Together, these act as a shim between the simulation and the original SENSEI analysis adaptors specified by the user (Fig. 5.6). The original analysis adaptors are run by the visualization execution application, which queries the data from the simulation using the simulation-oblivious data transfer model.

The analysis adaptor takes the VTK data from SENSEI, constructs the equivalent `libISSimState`, and calls `libISProcess` to send the data to the provided in situ visual-

```
struct SimState {
    libISBox3f world, local, ghost;
    // The rank that sent the data
    int simRank;
    // The 3D fields and 1D buffers sent by the simulation
    std::unordered_map<std::string, Buffer> buffers;
    // The particles sent by the simulation, if any
    Particles particles;
};

// Connect to the simulation listening on rank 0 at host:port
void connect(const std::string &host, const int port,
    MPI_Comm ownComm, bool *sim_quit = nullptr);

// Query the next time step, blocking until it is ready
std::vector<SimState> query(bool *sim_quit = nullptr);

// Asynchronously query the next time step.
// The future can be monitored for completion
std::future<std::vector<SimState>> query_async(bool *sim_quit = nullptr);

// Disconnect from the simulation
void disconnect();

// Check if the simulation has terminated
bool sim_connected();
```

Listing 8: The client interface is used to connect to a running simulation and query data from it. Each client will receive $\frac{M}{N}$ simulation states, containing data from the simulation ranks it is assigned to receive data from.

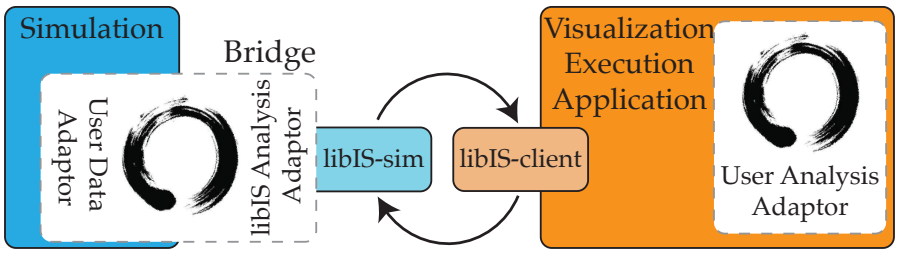


Fig. 5.6: The architecture of the loosely coupled in situ visualization execution system for SENSEI using the presented simulation-oblivious data transfer model.

ization execution application. The in situ visualization execution application uses the library to query data from the simulation, constructs the equivalent VTK representation, and passes it on to the desired analysis adaptors. The conversion back to VTK's data representation uses VTK's zero-copy arrays, avoiding additional data copies within the in situ analysis execution application.

Although running the SENSEI visualization with the simulation-oblivious approach in a $1:1$ configuration is straightforward, doing so for an $M > N$ configuration is less so. SENSEI's existing data model expects one region per rank, i.e., a tightly coupled or a data restructuring use case, requiring some special treatment for $M > N$ configurations. For example, an analysis adaptor that computes a global reduction over the data, such as the min or max, or a histogram, may not produce the correct result if each client rank ran a `for` loop over its assigned regions. To avoid requiring significant modifications to the visualization code, data are passed to the user's analysis adaptor as a `vtkMultiBlockDataset`, giving the appearance of a single simulation rank with $\frac{M}{N}$ distinct blocks of data.

5.3.3 Evaluation

The scalability and portability of the proposed simulation-oblivious model is evaluated using two simulations, LAMMPS (Section 5.3.3.1) and Poongback (Section 5.3.3.2), on two HPC systems. The evaluation is run on Stampede2 at the Texas Advanced Computing Center (TACC) and Theta at Argonne National Laboratory. Both systems contain roughly similar Intel Xeon Phi Knight's Landing (KNL) nodes: KNL 7250 on Stampede2 and KNL 7230 on Theta. Stampede2 contains an additional partition of Intel Skylake Xeon (SKX) nodes. Although the KNL nodes are similar on the two systems, the network architecture differs significantly. Stampede2 employs a fat-tree topology Omnipath network, whereas

Theta uses a 3-level Dragonfly topology Cray Aries network. On Stampede2 MPI uses the Omnipath network, which can provide up to 100Gbps of bandwidth; however, sockets use the 1Gbps ethernet network. Theta does not provide an ethernet network. Instead, sockets use the Aries network and can achieve a peak bandwidth of 14Gbps.

Section 5.3.3.1.1 conducts a weak scaling study to evaluate the performance portability of the simulation-oblivious model using both the MPI and socket intercommunicators on Stampede2 and Theta. The weak scaling study is run using LAMMPS and a test client application that queries data repeatedly. The simulation-oblivious model is also evaluated on an example use case of in situ image database generation for LAMMPS (Section 5.3.3.1.2) and Poongback (Section 5.3.3.2.6) on Stampede2, to measure the rendering performance of the client and the visualization's impact on simulation performance. Finally, Section 5.3.3.3 compares the data transfer performance achieved by the two restructuring and simulation-oblivious data transfer models against each other and the existing restructuring model used in ADIOS.

The in situ image database rendering benchmark is an example of using the proposed approach in combination with an OSPRay-based [291] rendering application to render image databases, similar to those used in Cinema [7]. The image database rendering application discussed in Section 5.1 is modified to query data using the simulation-oblivious data transfer model. The benchmark repeatedly queries a time step and renders a camera orbit around the data for a specified number of time steps.

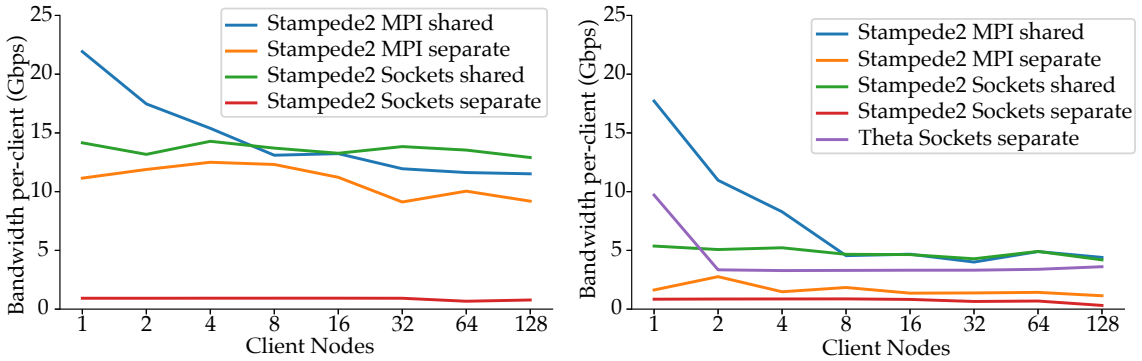
5.3.3.1 Molecular Dynamics

LAMMPS [225, 242] is a large-scale molecular dynamics simulation code, which is run MPI-parallel. The simulation-oblivious data transfer method is integrated into LAMMPS through existing mechanisms for coupling LAMMPS with other codes [243]. This integration is done by building a wrapper application that behaves as the regular LAMMPS executable, with the difference that before the simulation starts, the wrapper initializes the data transfer library interface and installs a callback to call `libISProcess` after each time step. After each time step, each simulation rank sends its local and ghost particles to its assigned client.

5.3.3.1.1 Performance portability and scalability. The weak scaling benchmark uses the Lennard Jones benchmark problem included with LAMMPS, which is replicated to store 131k particles per simulation rank. The benchmark measures the bandwidth achieved when querying data using an example client that simply prints out the received metadata. The benchmarks are run in a 16 : 1 configuration, where each client receives approximately 2.1M particles, which amounts to roughly 45MB of data per client after including ghost zones. Each group of 16 simulation ranks is placed on one node, and the client is run with one rank per node.

The benchmark is run on 1 to 128 client nodes, corresponding to 16 : 1 to 2048 : 128 configurations, using the SKX nodes on Stampede2 (Fig. 5.7a) and the KNL nodes on Stampede2 and Theta (Fig. 5.7b). In the shared node configuration, the client and simulation are run across the entire set of nodes, using the same resources. In the separate node configuration, the allocation of nodes is split in half between the simulation and client. To evaluate performance portability between systems that do or do not support connecting over MPI, the benchmark is run using the MPI and socket intercommunicators in the shared and separate configurations.

As shown in Fig. 5.7, the independent communication mode employed by the simulation-oblivious approach weak scales well, and can nearly saturate the 1Gbps ethernet network when using sockets on Stampede2. When using MPI over Omnipath on Stampede2, the simulation-oblivious approach is not network bound, and averages 11% network utilization



(a) Bandwidth per client on Stampede2 SKX.

(b) Bandwidth per client on KNL.

Fig. 5.7: Bandwidth per client rank achieved in the weak scaling benchmark. The independent communication strategy employed by the simulation-oblivious data transfer model achieves good weak scaling. Shared node runs are able to leverage shared memory for improved bandwidth.

on SKX and 2% on KNL. When using sockets on Stampede2, the approach averages 88% network utilization on SKX and 74% on KNL. When using sockets on Theta, the approach averages 30% network utilization. The relatively low network utilization achieved with MPI could potentially be resolved by parallelizing the data transfer from the simulation ranks to their clients. Although each group of simulation ranks and clients communicates in parallel, the simulation ranks within a group send their data to the client in serial.

When run in the shared node configuration, shared memory can be used for higher bandwidth by both the MPI and sockets intercommunicators, and both intercommunicators are found to achieve similar performance. The shared node configuration avoids the practical challenges of queuing and running a second job on the HPC system and is likely to be a common use case for in situ visualization. The ability of both MPI and sockets to use shared memory for higher bandwidth is especially promising for performance portability.

5.3.3.1.2 In situ image database rendering. The image database rendering example uses the Rhodopsin benchmark input and runs LAMMPS in the same 16 : 1 configuration as before. OSPRay uses an MPI + Threads model and is run with a single rank per node, whereas LAMMPS is run MPI-parallel with 16 ranks per node. The Rhodopsin benchmark stores 32k particles per simulation rank, resulting in each client rank receiving 512k particles in the 16 : 1 configuration, which, after accounting for ghost zones, corresponds to roughly

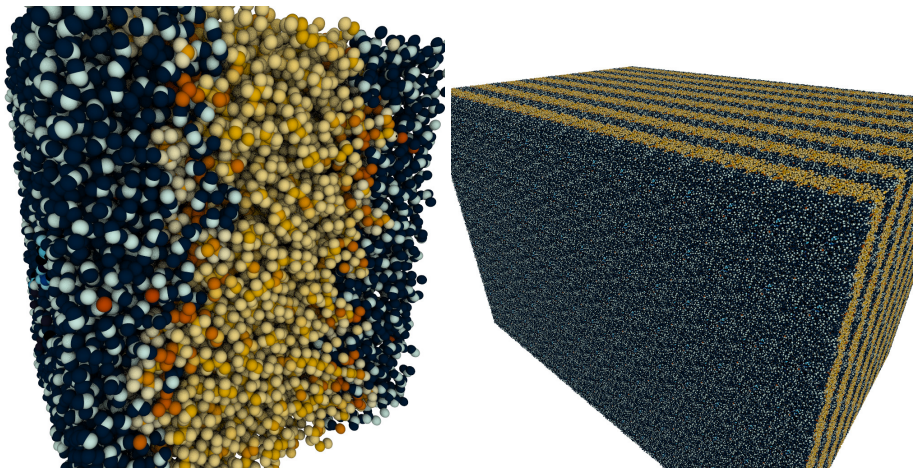
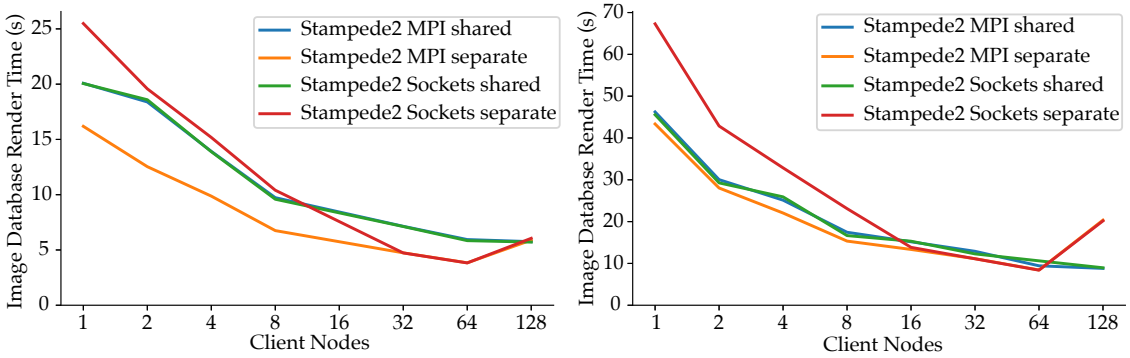


Fig. 5.8: The LAMMPS Rhodopsin benchmark rendered with ambient occlusion. Left: the simulation data on a single rank. Right: replicated in the weak scaling benchmark for 1024 ranks. By using the ghost particles already computed by LAMMPS, the in situ renderer is able to compute ambient occlusion effects using only the local data available on each client rank.

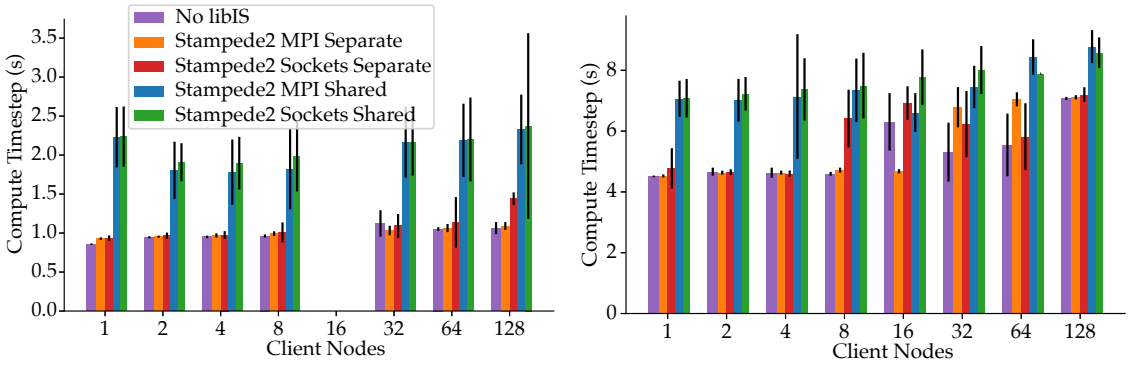
20MB of data per client. The additional ghost particles are used to compute ambient occlusion to provide better depth cues (see Fig. 5.8). The benchmark renders an orbit of 80 camera positions around the data, and does so for 50 time steps. OSPRay's asynchronous rendering support is used to reduce the total time to render the set of by rendering eight images in parallel (also see Section 5.1). The benchmark measures the total time to render all 80 frames each time step in the shared and separate configurations using both intercommunicators, with a 1024^2 image size (Fig. 5.9).

At low node counts, the difference observed in the total render time between the MPI and socket intercommunicators is unexpectedly large. The different communicators affect only data transfer performance, which is not timed in the rendering benchmark, and thus should achieve similar rendering performance. At higher node counts, the MPI and socket modes converge to similar render times, with the separate node configuration achieving slightly better performance than the shared node one, as expected. The shared node configuration runs both the simulation and renderer on the same nodes, and some performance degradation of both is to be expected. On KNL, each configuration achieves roughly similar performance, potentially due to the larger number of available cores for the renderer, and the relatively weaker per-core performance compared to SKX. The overall render time decreases in the benchmark as it is scaled up, which is a result of each client's local data projecting to a smaller region of the image as the data set is scaled up.



(a) Stampede2 SKX. **(b)** Stampede2 KNL.

Fig. 5.9: Camera orbit render times on SKX and KNL for each time step. The image database rendering task scales well, with the shared node configuration slightly impacted by the simulation. A performance decrease is observed at 128 clients in the separate configuration, which may be due to a different network placement for the 256 node job impacting compositing performance.



(a) Stampede2 SKX.

(b) Stampede2 KNL.

Fig. 5.10: The impact of the image database rendering client on LAMMPS performance. Bars show the average time to compute each time step in the different modes, with standard deviation shown as error bars. As expected, the shared node configuration significantly impacts performance, whereas the separate node configuration has relatively little effect. On 16 SKX nodes the LAMMPS simulation became unstable, and we were unable to run benchmarks in that configuration. The plots share the same legend.

5.3.3.1.3 Impact on simulation performance. Fig. 5.10 displays the impact on simulation performance for each configuration by comparing the time taken to compute each time step with and without the mini-cinema client running. In the separate node configuration, the simulation and client are run on distinct nodes, and do not contend for resources. Thus, the simulation is impacted only by the time spent sending data to the clients. On the Stampede2 SKX nodes, the simulation-oblivious approach has a negligible impact when using MPI or sockets, although an outlier is encountered at 128 nodes where a larger impact when using sockets is observed. On the KNLs, the simulation is 4% slower on average when using MPI, with a similar impact observed for sockets, although an outlier is observed at 16 nodes where a greater impact when using sockets is measured. The greater impact on KNL can be attributed to the lower single-core performance compared to SKX, which increases the time spent transferring data to the clients.

The shared node configuration runs the simulation and client on the same nodes, potentially leading to significant resource contention and impacting simulation performance. On SKX, simulation takes 82% longer when using MPI and 104% longer when using sockets. On KNL, the simulation takes 44% longer when using MPI and 44% longer when using sockets. The reduced impact on KNL is likely attributable to the larger number of cores available, which may reduce resource contention for processors, and the availability of

MCDRAM, which in this configuration is large enough to hold the data of both LAMMPS and mini-cinema.

5.3.3.2 Direct Numerical Simulation of Turbulent Channel Flow

The turbulent channel flow example uses Poongback [165–167, 186] to simulate a large-scale turbulent channel fluid flow and the proposed simulation-oblivious data model to transfer data to the mini-cinema clients as before. Poongback is a computational fluid dynamics (CFD) solver for direct numerical simulation (DNS) of incompressible turbulent channel flows written in Fortran. The simulation generates a 3D volume data set and runs MPI-parallel. Poongback is written in Fortran, and integrates the in situ library using the provided Fortran wrapper. Minimal modifications to the simulation code are required to integrate in situ visualization through the library. First, before the Poongback simulation begins, it initializes the in situ interface and configures the simulation state on each rank with the bounds of its local volume data. After each time step, Poongback calls the `libISProcess` Fortran wrapper to send the data on to any clients requesting the current time step. Poongback stores its data row-major, and does not require a transpose to be done after receiving data on the client.

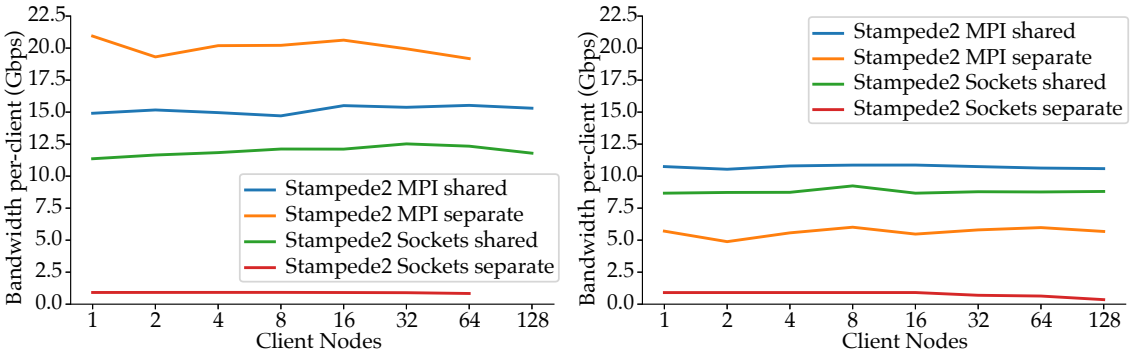
5.3.3.2.4 Evaluation setup. As done for LAMMPS, the evaluation is performed by transferring data from the simulation, Poongback, to the mini-cinema rendering client for a number of time steps. The benchmark records the network utilization of the data transfer, rendering performance of the client, and the impact on simulation performance. The Poongback benchmarks are run on Stampede2 SKX and KNL nodes, using a weak scaling benchmark for Poongback. To create the weak scaling benchmark, the simulation dimensions are increased based on the number of clients, maintaining roughly 1.4GB of volume data per client (Table 5.1). The simulation and clients are run in a $32:1$ configuration. Poongback is run MPI-parallel with 32 ranks per node, whereas the mini-cinema clients are run using MPI + Threads with one rank per node. The benchmarks are run in both the shared and separate configurations and use the MPI and socket intercommunicators. When using the shared node configuration, the simulation and application are run on the same nodes, whereas in the separate configuration half the nodes are assigned to the simulation and half to the client. The benchmarks are run up to an aggregate of 128 SKX nodes and

Table 5.1: Weak scaling configurations for the Poongback benchmark, targeting roughly 1.4GB of volume data per client rank.

Client Ranks	Total Voxels	Total Volume Size (GB)
1	384×768×576	1
2	768×768×576	3
4	768×768×1152	5
8	1536×768×1152	11
16	1536×768×2304	22
32	3072×768×2304	43
64	3072×768×4608	87
128	6144×768×4608	174

256 KNL nodes. The shared node runs are scaled up from a 32 : 1 configuration up to a 4096 : 128 one on SKX and KNL. The separate node runs are scaled up from a 32 : 1 configuration up to a 2048 : 64 one on SKX and a 4096 : 128 one on KNL.

5.3.3.2.5 Performance portability and scalability. Fig. 5.11 displays the average data transfer bandwidth achieved on each client over 50 time steps. When using MPI in the separate node configuration, data are transferred over the 100Gbps Omnipath network, and the simulation-oblivious approach achieves an average of 20% network utilization on SKX and 6% on KNL. When using sockets in the separate node configuration, data are transferred over the 1Gbps ethernet network, and the approach achieves 90% network utilization on SKX and 78% on KNL. As discussed previously, parallelizing the data transfer within each set of simulation ranks and client could improve network utilization when using MPI, especially for the larger aggregate data transfer performed for the Poongback



(a) Bandwidth per client on Stampede2 SKX. **(b)** Bandwidth per client on Stampede2 KNL.

Fig. 5.11: Bandwidth per client rank achieved in the Poongback weak scaling benchmark. The independent communication strategy employed by simulation-oblivious data transfer model achieves good weak scaling, and performs well when transferring the larger portion of data sent by each Poongback rank.

simulation.

In the shared node configuration, each set of 32 : 1 ranks is run on the same node, ensuring communication is local to each node. As a result, data can be transferred using shared memory instead of over the network to achieve higher bandwidth. A notable exception is on SKX (Fig. 5.11a), where higher bandwidth is achieved in the separate configuration than in the shared one when using MPI. This result is counter to the results observed on KNL with Poongback and on KNL and SKX with LAMMPS, and warrants further investigation.

Overall, higher bandwidth is achieved on SKX than KNL, likely due to the higher per-core performance of SKX. Similar to the results observed on LAMMPS in Section 5.3.3.1.1, MPI achieves higher bandwidth than sockets, as MPI is able to leverage the faster Omnipath network. However, in the shared configuration both can leverage shared memory and the gap between the two narrows. In terms of overall weak scalability, the independent data transfer strategy employed by the simulation-oblivious data transfer model scales well with the number of clients, with per client bandwidth remaining nearly constant across each scaling run.

5.3.3.2.6 In situ image database rendering. For each of the 50 time steps queried during the benchmark, the mini-cinema client renders an 80-position camera orbit around the data set (Fig. 5.12). Each image is rendered at a 1024^2 resolution, with one sample per pixel and a volume sampling rate of one sample per voxel. In the largest run with 128 client ranks, the application queries and renders a total of 8.7TB of volume data over the course of the 50 time steps. The mini-cinema client is run with a single image in flight to reduce the impact of the client in the shared node configuration, and in the same mode in the separate configuration for consistency.

The time taken to render each camera orbit is shown in Fig. 5.13. In contrast to the results observed with LAMMPS, the overall rendering time increases as additional client ranks are added. The difference in rendering performance is likely due to the differing data distributions of Poongback and LAMMPS. Compared to LAMMPS, where each simulation rank has a cube of data, Poongback partitions its data using a pencil decomposition [166]. After multiple pencil-pencil data transposes, the set of regions assigned to each client is a group of these x-pencils along the axis of flow, so that each pencil spans the entire x-axis

of the data set. The set of regions assigned to each client is a group of these x-pencils, where each pencil spans the entire x axis of the data set. The set of regions projects to a large number of pixels for most of the viewpoints in the orbit. The number of pixels covered does not reduce significantly as more ranks are added, leading to a significant amount of rendering and compositing work. One measure that could be taken to alleviate the compositing work is to merge the 32 regions assigned to each client rank into a single OSPRay rendering region, allowing them to be rendered and composited as a single brick, instead of 32 separate ones. This optimization would reduce the compositing work by a factor of 32, allow for faster local rendering, and provide a meaningful performance improvement.

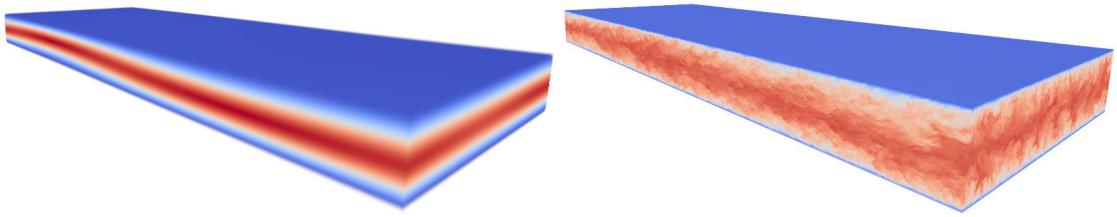
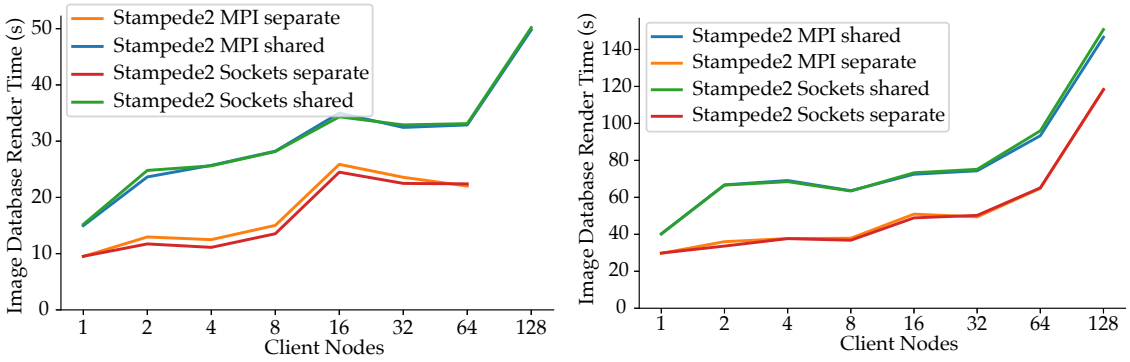


Fig. 5.12: Images of the Poongback turbulent channel-flow simulation rendered using the mini-cinema visualization client. Both volumes are $6144 \times 768 \times 4608$ voxels in double-precision floating-point values (174GB). Left: The simulation state at time step 0, the initial condition used in the weak scaling benchmarks. Right: A checkpoint of the same simulation (fully developed turbulence).



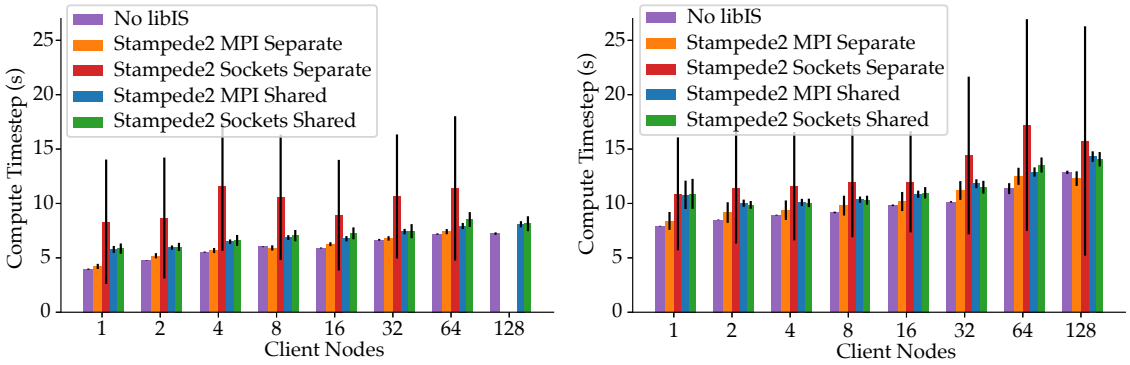
(a) Stampede2 SKX.

(b) Stampede2 KNL.

Fig. 5.13: Camera orbit render times on SKX and KNL for each time step in the weak scaling benchmark. Better overall rendering performance is achieved on SKX, and in the separate node configurations, as the shared node configuration oversubscribes the nodes. In contrast to the LAMMPS results, the renderer does not scale as well to higher node counts, potentially due to the differences in data distribution of Poongback and the resulting differences in the compositing workload.

We find similar results as those observed with LAMMPS when comparing the relative rendering performance of the different configurations and communicators. Although we find little impact of the communication method used (MPI vs. sockets) on render time, we find a significant impact on performance when using the shared configuration. The shared configuration can lead to contention between the simulation and client, and impacts the performance of both.

5.3.3.2.7 Impact on simulation performance. Fig. 5.14 shows the impact of running the mini-cinema client in different configurations on simulation performance. The impact incurred by the data transfer is primarily determined by the ratio of compute time and data transfer time for each time step. Depending on how large this ratio is, the simulation becomes either compute bound or communication bound. Thus, the impact on simulation time is related to the network performance measured in Section 5.3.3.2.5. The standard deviation of the overhead is much higher when using the sockets intercommunicator in the separate node configuration, which is likely attributable to varying network performance when using sockets. MPI and local sockets are able to achieve higher bandwidth with greater consistency to transfer the data faster and reduce impact on the simulation. When not rendering multiple frames in parallel, the rendering client has little impact on the simulation in the shared configuration, as can be seen by comparing the performance impact of the shared modes on Poongback and LAMMPS, where the client was configured



(a) Stampede2 SKX. **(b)** Stampede2 KNL.

Fig. 5.14: The impact of the image database rendering client on Poongback performance. Bars show the average time to compute each time step in the different modes, with standard deviation shown as error bars. Overall, the time-consuming socket-based communication in the separate configuration incurs the most overhead. By not rendering multiple frames in parallel, the renderer's impact on the simulation is reduced in the shared configurations.

to render multiple frames in parallel.

When using the socket communicator in the separate mode, the simulation becomes communication bound, and must wait for the data transfer to complete before advancing, thereby impacting performance. The sockets separate configuration has the greatest impact on simulation performance of the different modes, increasing compute time by 78% on SKX and 34% on KNL. The impact is greater on SKX as faster nodes lead to a 1.7× faster simulation, exacerbating the impact of the simulation becoming bound by the data transfer. In the other configurations, the impact is less severe. The MPI separate mode increases simulation time by 4% and 6%, on SKX and KNL, respectively; the MPI shared mode by 18% and 14%; and the sockets shared mode by 22% and 17%.

5.3.3.3 Data Transfer Performance of Restructuring and Simulation-Oblivious Approaches

To measure the performance impact of restructuring the data during transfer, this section compares the data transfer performance of the proposed restructuring and simulation-oblivious approaches with each other and the restructuring technique used in ADIOS2 [94, 177]. Whereas the presented restructuring technique is based on a spatial decomposition, ADIOS treats the particle attributes as 1D arrays without spatial information. When using ADIOS, the particle distribution on each visualization client will be more similar to the simulation-oblivious approach, which also does not consider the spatial distribution of the data. ADIOS does support spatially driven restructuring for 2D or 3D grid data.

ADIOS repurposes its existing I/O API to support in situ visualization, allowing users to simply change the I/O "engine" being used by the simulation and visualization. ADIOS adopts a lightweight data model, processing 1D, 2D, or 3D arrays of primitive types that are passed to the I/O engine. ADIOS supports restructuring the data during transfer by providing an interface similar to reading subregions of a shared file. Each client specifies a starting offset within an array and number of elements to be read, as if reading a file. ADIOS will then make the requested data available on that rank. Although this "file-based" API allows for a transparent transition from a post hoc visualization pipeline, it requires some form of data restructuring to be performed in an in situ scenario, potentially adding overhead to the data transfer.

The replicated LAMMPS Lennard Jones benchmark used in Section 5.3.3.1.1 is used to

compare the bandwidth achieved by each method. The benchmark generates 131k particles on each simulation rank and is run in a 16 : 1 configuration of the simulation and client, as used in Section 5.3.3.1.1. Each particle attribute (x, y, z, type) is passed to ADIOS as a 1D global array. Clients request the subregion of this array corresponding to their assigned set of simulation ranks, using an assignment similar to the proposed simulation-oblivious approach. The ADIOS benchmarks use the "SST" data transfer engine, which supports $M : N$ data transfer and will make use of RDMA enabled networks where available.

In both the shared and separate configurations, ADIOS achieves similar data transfer speeds, averaging 0.5Gbps per client on SKX and KNL when run with 16 clients on Stampede2 (a 256 : 16 configuration). The proposed spatially aware data restructuring method achieves poorer performance, achieving just 0.2Gbps in the shared configuration and 0.09Gbps in the separate configuration. In contrast, the proposed simulation-oblivious approach averages 13Gbps and 11Gbps, respectively on SKX, and 5Gbps and 1.5Gbps, respectively on KNL (see Fig. 5.7). The potential need for more global communication among ranks to perform data redistribution comes at a cost compared to the independent model used in the simulation-oblivious approach, though is convenient for applications. Furthermore, the global, serial data query process and the acceleration structure-less approach to performing spatial queries in the spatially aware data restructuring approach (Section 5.3.1) comes with a significant performance impact.

5.4 Summary

The need for scalable in situ visualization continues to grow as simulations increase in scale, requiring new scalable solutions for existing in situ use cases and new flexible, low-overhead techniques to minimize the impact of the visualization on the simulation. This chapter has presented work to enable scalable in situ visualization across tightly and loosely coupled configurations. Rendering is a common task required in visualization, and thus in situ visualization. The Distributed FrameBuffer [281] can also be used in situ to provide scalable high-quality rendering, as demonstrated through a Cinema-style mini-app. Ongoing work to integrate data staging based in situ visualization methods within the adaptive, spatially aware two-phase I/O pipeline can transparently provide in situ visualization to simulations using the I/O strategy, while also providing a number of

benefits over existing approaches. Finally, two data transfer methods were proposed to enable low memory or compute overhead loosely coupled in situ visualization.

The Distributed FrameBuffer [281] provides a compelling option to enable flexible, scalable, and high-quality in situ visualization in tightly and loosely coupled use cases. The core scalable rendering component provided by the DFB is applicable to accelerate a broad range of in situ rendering applications, from static images or movies to massive image databases or interactive rendering. Morever, the flexibility provided by the DFB with regard to the data distribution is especially useful to support a wide range of simulations. Applications with more complex data layouts, which standard compositing libraries may not be able to render in place, can be virtually partitioned using regions to allow rendering them in place. Loosely coupled applications can also leverage hybrid-parallel rendering for load balancing to improve performance.

Passing data from the simulation to in situ visualization tasks follows a similar model as passing data to an I/O library for output. This similarity has made the I/O library a popular path to integrate in situ visualization, because it can be done without modifying the simulation code. A number of approaches using this model to enable tightly coupled, data staging, and loosely coupled in situ visualization have been proposed and effectively demonstrated [17, 63, 68, 94, 177, 284, 326, 327]. Compared to existing approaches, which are either not spatially aware or not adaptive, integrating in situ visualization into the spatially aware two-phase I/O pipeline proposed in this work provides a number of benefits. The in situ visualization tasks run on each aggregator are provided a spatially coherent and load balanced data distribution via the I/O pipeline, leading to improved performance for visualization tasks. Moreover, tasks run on the aggregators can use the BAT data layout to accelerate local and remote data access.

Finally, this chapter proposed two models for low-overhead data transfer for loosely coupled and on-demand in situ visualization. In situ visualization using these data transfer models can be run on the same nodes as the simulation, different nodes, or an entirely different HPC resource, as desired. The proposed spatially aware data restructuring approach for particle data treats the simulation as a data server that visualization tasks can make spatial queries to. However, performing this spatial data restructuring comes at significant cost. The proposed simulation-oblivious approach does not restructure the

data during transfer, instead providing each visualization rank a set of simulation states to process. By skipping restructuring, the simulation-oblivious approach can quickly transfer data to the visualization clients to minimize the time the simulation must spend performing data transfers. As the simulation-oblivious approach does not inspect the data, it can also be easily integrated into simulations that use custom mesh or data layouts and specially tailored visualization pipelines. Moreover, the simulation-oblivious approach does not preclude the use of data restructuring after the data have been transferred to provide a better data distribution to visualization tasks. This restructuring could be performed using a similar adaptive and spatially aware approach as employed in the adaptive, spatially aware two-phase I/O pipeline.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The continuing growth in computing power available on HPC systems has enabled increasingly higher fidelity simulations, which in turn produce ever increasing amounts of data. This large volume of output data poses a challenge to existing strategies for I/O and visualization that are required to gain scientific insight from such massive simulations. This dissertation proposes methods to address key challenges arising in the simulation visualization pipeline when faced with this massive volume of data, laying the groundwork for an efficient end-to-end simulation visualization pipeline. To address these challenges, this dissertation adopts a holistic view of the simulation visualization pipeline and proposes work spanning it, from I/O and data layout to post hoc and in situ visualization. At a high level, the proposed methods address these challenges by adopting adaptive and multiresolution processing throughout the simulation visualization pipeline.

The growing size of data computed by simulations requires scalable, high throughput I/O strategies that can quickly output the data so that the simulation can advance to the next time step. However, as discussed in this dissertation, it is not sufficient to output just a raw dump of the data, as the raw data dump will likely not provide efficient read access for post hoc visualization tasks. Chapter 3 proposes new strategies for spatially aware adaptive aggregation of particle data for scalable I/O, along with a low-overhead, visualization-focused data layout for particle data that can be constructed quickly during I/O. The proposed I/O strategies and data layout work together to quickly output massive particle data sets directly in a visualization-focused data layout, eliminating the need for a postprocess data conversion, without sacrificing I/O performance or portability. The aggregation tree adaptive I/O strategy proposed in this work is able to improve I/O performance by up to 2.5$\times$ on nonuniform particle distributions, by adjusting how ranks are grouped together during two-phase I/O to improve load balance. The proposed BAT

data layout supports progressive multiresolution reads with spatial and attribute filtering with little memory overhead, and is specifically designed to enable fast parallel construction during the I/O pipeline.

Post hoc visualization tasks face a range of challenges when faced with visualizing massive data sets. On one hand, scalable distributed rendering methods are needed to render the full-resolution data on HPC systems, whereas other methods are needed to enable visualization in immersive VR environments and other computationally constrained environments, such as the browser, to improve accessibility. Chapter 4 presents work on post hoc visualization methods that adopt adaptive and multiresolution computation and data layouts to enable visualization of massive data sets across HPC systems, VR, and the web browser. These works are key to enable scientists to gain insight from the massive data sets output by current and upcoming simulations. The DFB enables scalable full-resolution rendering of massive data on HPC systems through a flexible tile-based image processing pipeline. The DFB enables scalable image-, data-, and hybrid-parallel rendering within the same compositing framework to support a wide variety of distributed renderers. A VR neuron tracing tool, VRNT, was proposed to enable immersive exploration of large data sets in VR. VRNT is built on a multiresolution data layout and on-demand page-based processing system to ensure a high-quality VR experience for neuroanatomists. The intuitive and direct interaction provided by VRNT was found to improve neuron tracing performance and understanding through a pilot study conducted with neuronatomists from the Angelucci lab. Neuroanatomists are able to use the tool comfortably for long periods on large data sets as a result of VRNT's on-demand page-based processing system, which ensures the tool meets the high rendering demands of VR. Finally, a new GPU-driven isosurface extraction algorithm for block-compressed volumes, BCMC, was proposed to enable interactive visualization of massive data sets in constrained environments. BCMC significantly reduces the memory required to isosurface massive volumes by decompressing and caching blocks as needed through a GPU-driven LRU cache, and performs all computation on the GPU to accelerate isosurface extraction. BCMC enables interactive isosurface extraction on massive volumes in constrained environments, such as the web browser, improving access to visualization of massive scientific data sets.

In situ visualization methods face similar challenges as post hoc methods with respect

to rendering scalability and similar challenges as parallel I/O with respect to load balancing and data layout. In situ visualization is also faced with its own unique challenges. This dissertation focuses on the need to minimize the impact of the visualization on the simulation, a key concern for large-scale simulations. Chapter 5 demonstrates how the work proposed in this dissertation on scalable post hoc visualization, adaptive load balancing for I/O, and data layouts can be applied to in situ visualization. In situ rendering of massive data at scale is a challenging problem, and is made even more challenging when a large number of images must be rendered, e.g., as needed for explorable extracts methods. The scalable rendering provided by the DFB for post hoc visualization is equally applicable to accelerate in situ rendering, as demonstrated in this chapter. The adaptive spatially aware I/O pipeline proposed in this dissertation can also be used as a data staging in situ execution environment, by running the in situ visualization on the aggregators. In situ visualization run in this manner can take advantage of the load balancing provided by the aggregation tree and use the BAT data layout to accelerate local and remote data queries. Moreover, Chapter 5 presents new low-overhead strategies for loosely coupled in situ visualization that minimize the impact of the visualization on the simulation. The new data transfer strategies presented provide significant flexibility to allow running the simulation and visualization in their ideal configurations, supporting $M : N$ configurations, asynchronous and on-demand execution, and flexible placement of the visualization processes. Although the spatially aware approach discussed incurs a high computational cost during data transfer, the simulation-oblivious approach does not, minimizing the impact of the visualization on the simulation. The simulation-oblivious approach achieves high bandwidth data transfers to the visualization clients, which are then free to adjust the data distribution as desired.

6.1 Exciting Avenues for Future Research

Although this dissertation has laid the groundwork for an efficient end to end simulation visualization pipeline, much work remains to be done within this broad effort to enable science at exascale.

Chapter 3 presented new strategies for adaptive spatially aware I/O and low-overhead visualization-focused data layouts. Although the improved load balancing provided by the aggregation tree achieves a significant improvement in I/O performance, it does not

consider partitioning the data within a rank, limiting the granularity of load balancing that can be achieved. However, it is also not feasible to aggregate the entire data set to a single rank to construct a perfectly balanced aggregation strategy. To construct a more balanced aggregation strategy, it may be possible to subdivide ranks with large amounts of data into "virtual" ranks before constructing the aggregation tree, or to exchange some form of reduced representation of the data distribution on each rank. With regard to the BAT data layout, the assumption of spatial coherence in the attributes is a main limitation of the low-overhead attribute filtering approach. If some attributes are not spatially coherent, the bitmaps in the tree will be less useful, impacting query filtering performance. This limitation could potentially be addressed by adopting more advanced binning schemes [313] or additional hierarchies on these attributes. Better and more consistent error improvement for attributes could be provided through a more advanced LOD sampling scheme, e.g., using the bitmaps [267]. Such an approach would allow the layout to provide stronger quality vs. error guarantees to users. However, building a layout that provides strong quality guarantees can be computationally expensive, and could in turn degrade I/O performance.

A key limitation of compositing-based rendering algorithms, such as the data- and hybrid-parallel renderers built on the DFB in Section 4.1, is that they do not support global illumination effects that require data from different nodes. Local lighting effects such as ambient occlusion can be achieved by duplicating subsets of the data to provide ghost zones; however, shadows or full path tracing on the distributed data are not possible without a more advanced distributed rendering system. Recent work [1, 133, 201, 212, 262, 269] has begun investigating approaches based on moving rays or data, or both, to enable secondary effects, an effort that dates back to early work by Reinhard et al. [234]. However, the cost of moving data or rays over the network is high, and such approaches are typically noninteractive. Investigating strategies for high-performance secondary ray effects on distributed data is a challenging area of research, and would enable high-fidelity imagery for distributed rendering and in situ visualization. Such strategies would also have applications in production film rendering, where scene sizes have been growing rapidly.

Improving access to interactive visualization of massive data sets remains an ongoing challenge as simulation sizes grow or new visualization platforms are introduced. New

consumer VR systems based on self-contained head-mounted displays, such as the Oculus Quest 2 [204], reduce the barrier to entry for VR and provide an easily accessible immersive environment for visualization. However, even though the HMD is driven by what amounts to a high-end mobile phone, the framerate and resolution demands to provide a comfortable and immersive VR experience are no less strict. Enabling visualization of massive data sets in such a computationally constrained yet demanding environment requires new visualization techniques that adopt adaptive and multiresolution processing throughout all aspects of the visualization pipeline, from computation to rendering. Moreover, such techniques will need to adopt incremental strategies for updating data and performing computation to avoid dropping frames, as done in VRNT. Similar challenges are faced when considering additional techniques for visualizing massive data in the browser. For example, BCMC is built around exploiting the sparsity of typical isosurfaces to reduce the amount data that must be decompressed; however, this is less applicable to volume rendering, where the entire data set can be semitransparent and thus must be processed. Visualization algorithms that can work directly on variable resolution and precision data layouts, such as the layout proposed by Hoang et al. [120], should be explored to enable massive data visualization in these constrained environments and even on high-end workstations.

It would also be worthwhile to investigate methods for automatic tuning and placement of in situ visualization tasks. Rather than relying on the scientist or visualization developer to decide where and how to run the tasks, a runtime system used by both the simulation and visualization could make such a decision based on the data access patterns of the visualization and its estimated or measured execution time. The runtime system could then execute the visualization in a tightly coupled, data staging, or loosely coupled configuration, as appropriate to improve overall performance. Integrating in situ visualization aware scheduling into the asynchronous many-task runtime systems used by simulations could provide better overall performance and portability, easing the adoption of in situ visualization in practice.

REFERENCES

[1] G. Abram, P. Navrátil, P. Grosset, D. Rogers, and J. Ahrens, "Galaxy: Asynchronous ray tracing for large high-fidelity visualization," in *Proc. 2018 IEEE Symp. Large Data Anal. and Vis.*, 2018, pp. 72–76.

[2] L. Acciai, P. Soda, and G. Iannello, "Automated neuron tracing methods: An updated account," *Neuroinformatics*, vol. 14, no. 4, pp. 353–367, 2016.

[3] M. Aftosmis, M. Berger, and G. Adomavicius, *A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries*. Reno, NV, United States: Amer. Inst. of Aeronaut. & Astronaut., 2000.

[4] M. Agus, D. Boges, N. Gagnon, P. J. Magistretti, M. Hadwiger, and C. Calí, "GLAM: Glycogen-derived lactate absorption map for visual analysis of dense and sparse surface reconstructions of rodent brain structures on desktop systems and virtual environments," *Comput. & Graph.*, vol. 74, pp. 85–98, 2018.

[5] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu, "Scientific discovery at the exascale, a report from the DOE ASCR 2011 workshop on exascale data management, analysis, and visualization," Dept. of Energy Office of Advanced Scientific Computing Research, Washington, DC, United States, Tech. Rep. 1372701, 2011.

[6] J. Ahrens, B. Geveci, and C. Law, "ParaView: An end-user tool for large-data visualization," in *Vis. Handbook*, C. D. Hansen and C. R. Johnson, Eds. Burlington, VT: Butterworth-Heinemann, 2005, pp. 717–731.

[7] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in *SC '14: Proc. Int. Conf. for High Perform. Comput., Networking, Storage and Anal.* IEEE Press, 2014, pp. 424–434.

[8] A. Alsegård, "Interactive out-of-core rendering and filtering of one billion stars measured by the ESA gaia mission," M.S. Thesis, Dept. of Science and Technology, Media and Information Technology, Linköping Univ., Linköping, 2018.

[9] N. Apthorpe, A. Riordan, R. Aguilar, J. Homann, Y. Gu, D. Tank, and H. S. Seung, "Automatic neuron detection in calcium imaging data using convolutional networks," in *Advances in Neural Inf. Process. Syst. 29*, 2016.

[10] U. Ayachit, A. Bauer, E. P. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. E. Jansen, B. Loring, Z. Lukic, S. Menon *et al.*, "Performance analysis, design considerations,

and applications of extreme-scale in situ infrastructures," in *SC '16: Proc. Int. Conf. for High Perform. Comput., Networking, Storage and Anal.*, 2016, pp. 921–932.

[11] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "In situ methods, infrastructures, and applications on high performance computing platforms," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 577–597, 2016.

[12] A. C. Bauer, B. Geveci, and W. Schroeder, *The ParaView Catalyst User's Guide v2.0*. Clifton Park, NY: Kitware, 2015.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput., Networking, Storage and Anal.*, 2012, p. 11.

[14] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," in *Proc. 1970 ACM SIGFIDET (Now SIGMOD) Workshop Data Description, Access and Control*, 1970, pp. 107–141.

[15] N. Bell, J. Hoberock, and C. Rodrigues, "THRUST: A productivity-oriented library for CUDA," in *Program. Massively Parallel Processors*, D. B. Kirk and W.-m. W. Hwu, Eds. Burlington, MA: Morgan Kaufmann, 2017, pp. 475–491.

[16] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi *et al.*, "ASC ATDM level 2 milestone# 5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratory, Albuquerque, NM, United States, Tech. Rep. SAND-2015-8312, 2015.

[17] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, and V. Pascucci, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC '12: Proc. Int. Conf. High Perform. Comput., Networking, Storage and Anal.*, 2012, pp. 1–9.

[18] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[19] M. J. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *J. Comput. Phys.*, vol. 82, no. 1, pp. 64–84, 1989.

[20] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Comput. Phys.*, vol. 53, no. 3, Mar 1984.

[21] 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 J. Scientific Comput.*, vol. 38, no. 5, pp. S101–S122, 2016.

[22] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in GPU-based large-scale volume visualization," *Comput. Graph. Forum*, vol. 34, no. 8, pp. 13–37, 2015.

[23] T. Biedert and C. Garth, "Contour tree depth images for large data visualization," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2015, p. 10.

[24] T. Biedert, P. Messmer, T. Fogal, and C. Garth, "Hardware-accelerated multi-tile streaming for realtime remote visualization," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2018.

[25] T. Biedert, K. Werner, B. Hentschel, and C. Garth, "A task-based parallel rendering component for large-scale visualization applications," in *Proc. Eurographics Symp. Parallel Graph. and Vis.* The Eurographics Association, 2017, p. 9.

[26] J. Bigler, A. Stephens, and S. G. Parker, "Design for parallel interactive ray tracing systems," in *Proc. 2006 IEEE Symp. Interactive Ray Tracing*, 2006, pp. 187–196.

[27] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *Proc. 2014 ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 385–396.

[28] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. Cambridge, MA: MIT Press, 1990.

[29] D. D. Bock, W.-C. A. Lee, A. M. Kerlin, M. L. Andermann, G. Hood, A. W. Wetzel, S. Yurgenson, E. R. Soucy, H. S. Kim, and R. C. Reid, "Network anatomy and in vivo physiology of visual cortical neurons," *Nature*, vol. 471, no. 7337, pp. 177–182, 2011.

[30] D. Boges, M. Agus, R. Sicat, P. J. Magistretti, M. Hadwiger, and C. Calì, "Virtual reality framework for editing and exploring medial axis representations of nanometric scale neural structures," *Comput. & Graph.*, vol. 91, pp. 12–24, 2020.

[31] D. Boges, C. Calì, P. J. Magistretti, M. Hadwiger, R. Sicat, and M. Agus, "Immersive environment for creating, proofreading, and exploring skeletons of nanometric scale neural structures," in *Proc. Smart Tools and Apps for Graph. - Eurographics Italian Chapter Conf.* The Eurographics Association, 2019, p. 10.

[32] S. Boorboor, S. Jadhav, M. Ananth, D. Talmage, L. Role, and A. Kaufman, "Visualization of neuronal structures in wide-field microscopy brain images," *IEEE Trans. Vis. and Comput. Graph.*, vol. 25, no. 1, pp. 1018–1028, 2019.

[33] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *IEEE Trans. Vis. and Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, 2011.

[34] A. Bria, G. Iannello, and H. Peng, "An open-source Vaa3D plugin for real-time 3D visualization of terabyte-sized volumetric images," in *Proc. Biomed. Imag. (ISBI), 2015 IEEE 12th Int. Symp.* IEEE, 2015, pp. 520–523.

[35] K. L. Briggman, M. Helmstaedter, and W. Denk, "Wiring specificity in the direction-selectivity circuit of the retina," *Nature*, vol. 471, no. 7337, pp. 183–188, 2011.

[36] C. Brownlee, J. Patchett, L.-T. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C. Hansen, "A study of ray tracing large-scale scientific data in parallel visualization applications," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2012, p. 10.

[37] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms, "Scalable parallel I/O on a blue gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling," in *Proc. 2014 22nd Euromicro Int. Conf. Parallel, Distrib., and Network-Based Process.*, 2014, pp. 107–111.

[38] C. Burstedde, L. C. Wilcox, and O. Ghattas, "P4est : Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM J. Scientific Comput.*, vol. 33, no. 3, pp. 1103–1133, 2011.

[39] S. Byna, M. Chaarawi, Q. Koziol, J. Mainzer, and F. Willmore, "Tuning HDF5 subfiling performance on parallel file systems," in *Proc. Cray User Group*, 2017, p. 9.

[40] 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 *SC '12: Proc. Int. Conf. High Perform. Comput., Networking, Storage and Anal.* IEEE, 2012, pp. 1–12.

[41] 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," in *Proc. Cray User Group*, 2013, p. 9.

[42] C.-Y. Chan and Y. E. Ioannidis, "Bitmap index design and evaluation," *SIGMOD Rec.*, vol. 27, no. 2, pp. 355–366, 1998.

[43] ——, "An efficient bitmap encoding scheme for selection queries," *SIGMOD Rec.*, vol. 28, no. 2, pp. 215–226, 1999.

[44] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow, A. Choudhury, Y. Sabharwal, S. Singhal, and J. J. Parker, "Looking under the hood of the IBM blue gene/Q network," in *Proc. Int. Conf. High Perform. Comput., Networking, Storage and Anal.*, ser. SC '12, 2012.

[45] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier, S. Dutta, J. M. Favre, T. Fogal, S. Frey, C. Garth, B. Geveci, W. F. Godoy, C. D. Hansen, C. Harrison, B. Hentschel, J. Insley, C. R. Johnson, S. Klasky, A. Knoll, J. Kress, M. Larsen, J. Lofstead, K.-L. Ma, P. Malakar, J. Meredith, K. Moreland, P. Navrátil, P. O'Leary, M. Parashar, V. Pascucci, J. Patchett, T. Peterka, S. Petruzza, N. Podhorszki, D. Pugmire, M. Rasquin, S. Rizzi, D. H. Rogers, S. Sane, F. Sauer, R. Sisneros, H.-W. Shen, W. Usher, R. Vickery, V. Vishwanath, I. Wald, R. Wang, G. H. Weber, B. Whitlock, M. Wolf, H. Yu, and S. B. Ziegeler, "A terminology for in situ visualization and analysis systems," *The Int. J. High Perform. Comput. Appl.*, vol. 34, no. 6, pp. 676–691, 2020.

[46] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Ruebel, M. Durant, J. Favre, and P. Navrátil, "VisIt: An end-user tool for visualizing and analyzing very large data," Lawrence Berkeley National Laboratory, Berkeley, CA, United States, Tech. Rep., 2012.

[47] H. Childs, K.-L. Ma, H. Yu, B. Whitlock, J. Meredith, J. Favre, S. Klasky, N. Podhorszki, K. Schwan, and M. Wolf, "In situ processing," Lawrence Berkeley National Laboratory, Berkeley, CA, United States, Tech. Rep., 2012.

[48] J. Chou, K. Wu, and Prabhat, "FastQuery: A parallel indexing system for scientific data," in *Proc. 2011 IEEE Int. Conf. Cluster Comput.* IEEE, 2011, pp. 455–464.

[49] 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 *Proc. 2011 Int. Conf. for High Perform. Comput., Networking, Storage and Anal.* IEEE, 2011, pp. 1–11.

[50] K. Chung, J. Wallace, S.-Y. Kim, S. Kalyanasundaram, A. S. Andalman, T. J. Davidson, J. J. Mirzabekov, K. A. Zalocusky, J. Mattis, A. K. Denisin *et al.*, "Structural and molecular interrogation of intact biological systems," *Nature*, vol. 497, no. 7449, pp. 332–337, 2013.

[51] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno, "Optimal isosurface extraction from irregular volume data," in *Proc. 1996 Symp. Volume Vis.* IEEE Press, 1996, pp. 31–38.

[52] M. Ciżnicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierała, and J. Palczyński, "Efficient isosurface extraction using marching tetrahedra and histogram pyramids on multiple GPUs," in *Proc. Parallel Process. and Appl. Math.*, 2012.

[53] P. Coffman, F. Tessier, P. Malakar, and G. Brown, "Parallel I/O on theta with best practices," Talk at ALCF Simulation, Data, and Learning Workshop, 2018.

[54] R. H. Cohen, W. P. Dannevik, A. M. Dimits, D. E. Eliason, A. A. Mirin, Y. Zhou, D. H. Porter, and P. R. Woodward, "Three-dimensional simulation of a Richtmyer-Meshkov instability with a two-scale initial perturbation," *Phys. Fluids*, vol. 4, no. 10, pp. 3692–3709, 2002.

[55] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for AMR applications design document," Lawrence Berkeley National Lab, Berkeley, CA, United States, Tech. Rep., 2000.

[56] A. W. Cook, W. Cabot, and P. L. Miller, "The mixing transition in Rayleigh–Taylor instability," *J. Fluid Mechanics*, vol. 511, pp. 333–362, 2004.

[57] C. Crassin, F. Neyret, and S. Lefebvre, "Interactive gigavoxels," INRIA, Le Chesnay-Rocquencourt, France, Research Report RR-6567, 2008.

[58] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proc. 2009 Symp. Interactive 3D Graph. and Games*. ACM, 2009, pp. 15–22.

[59] C. Cruz-Neira, J. Leigh, M. Papka, C. Barnes, S. M. Cohen, S. Das, R. Engelmann, R. Hudson, T. Roy, L. Siegel *et al.*, "Scientists in wonderland: A report on visualization applications in the CAVE virtual reality environment," in *Proc. 1993 IEEE Res. Properties in Virtual Reality Symp.* IEEE, 1993, pp. 59–66.

[60] C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart, "The CAVE: Audio visual experience automatic virtual environment," *Commun. ACM*, vol. 35, no. 6, pp. 64–72, 1992.

[61] "NVIDIA: CUDA marching cubes example." [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/index.html

[62] C. Dachsbacher, C. Vogelgsang, and M. Stamminger, "Sequential point trees," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 657–662, 2003.

[63] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *Proc. 2014 14th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Comput.* IEEE, 2014, pp. 246–255.

[64] 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*, vol. 21, no. 5, 1993.

[65] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, "Memory sharing for interactive ray tracing on clusters," *Parallel Comput.*, vol. 31, no. 2, pp. 221–242, 2005.

[66] W. Denk, J. Strickler, and W. Webb, "Two-photon laser scanning fluorescence microscopy," *Sci.*, vol. 248, no. 4951, pp. 73–76, 1990.

[67] S. Dias, K. Bora, and A. Gomes, "CUDA-based triangulations of convolution molecular surfaces," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput. - HPDC '10*. Chicago, IL: ACM Press, 2010, pp. 531–540.

[68] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," *Cluster Comput.*, vol. 15, no. 2, pp. 163–181, 2012.

[69] J. Dubois and J.-B. Lekien, "Highly efficient controlled hierarchical data reduction techniques for interactive visualization of massive simulation data," in *Proc. EuroVis 2019 - Short Papers*, 2019, p. 5.

[70] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel, "High-speed marching cubes using HistoPyramids," *Comput. Graph. Forum*, vol. 27, no. 8, pp. 2028–2039, 2008.

[71] T. Dykes, A. Hassan, C. Gheller, D. Croton, and M. Krokos, "Interactive 3D visualization for theoretical virtual observatories," *Monthly Notices Roy. Astronomical Soc.*, vol. 477, no. 2, pp. 1495–1507, 2018.

[72] S. Eilemann and R. Pajarola, "Direct send compositing for parallel sort-last rendering," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2007, pp. 29–36.

[73] D. Ellsworth, H. Good, and T. Brice, "Distributing display lists on a multicomputer," in *Proc. 1990 Symp. Interactive 3D Graph.*, 1990, pp. 147–154.

[74] D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom, "Concurrent visualization in a production supercomputing environment," *IEEE Trans. Vis. and Comput. Graph.*, vol. 12, no. 5, pp. 997–1004, 2006.

[75] J. Elseberg, D. Borrmann, and A. Nüchter, "One billion points in the cloud – an octree for efficient processing of 3D laser scans," *ISPRS J. Photogrammetry and Remote Sens.*, vol. 76, pp. 76–88, 2013.

[76] K. Engel, "CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs," in *Proc. 2011 IEEE Symp. Large Data Anal. and Vis.* IEEE, 2011, pp. 123–124.

[77] N. Fabian, "In situ fragment detection at scale," in *Proc. IEEE Symp. Large Data Anal. and Vis.*, 2012, pp. 105–108.

[78] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Proc. 2011 IEEE Symp. Large Data Anal. and Vis.*, 2011, pp. 89–96.

[79] A. Febretti, A. Nishimoto, T. Thigpen, J. Talandis, L. Long, J. D. Pirtle, T. Peterka, A. Verlo, M. Brown, D. Plepys, D. Sandin, L. Renambot, A. Johnson, and J. Leigh, "CAVE2: A hybrid reality environment for immersive simulation and information analysis," in *Proc. The Eng. Reality Virtual Reality 2013*, M. Dolinsky and I. E. McDowall, Eds., vol. 8649. SPIE, 2013, pp. 9–20.

[80] O. Fernandes, S. Frey, F. Sadlo, and T. Ertl, "Space-time volumetric depth images for in-situ visualization," in *Proc. 2014 IEEE 4th Symp. Large Data Anal. and Vis.* IEEE, 2014, pp. 59–65.

[81] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974.

[82] T. Fogal and J. Krüger, "Tuvok, an architecture for large scale volume rendering," in *Proc. Vision, Model., and Vis.* The Eurographics Association, 2010, p. 8.

[83] ——, "An approach to lowering the in situ visualization barrier," in *Proc. First Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.* ACM Press, 2015, pp. 7–12.

[84] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger, "Freeprocessing: Transparent in situ visualization via data interception," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2014, pp. 49–56.

[85] T. Fogal, A. Schiewe, and J. Krüger, "An analysis of scalable GPU-based ray-guided volume rendering," in *Proc. 2013 IEEE Symp. Large-Scale Data Anal. and Vis.*, 2013, pp. 43–51.

[86] R. Fraedrich, S. Auer, and R. Westermann, "Efficient high-quality volume rendering of SPH data," *IEEE Trans. Vis. and Comput. Graph.*, vol. 16, no. 6, pp. 1533–1540, 2010.

[87] R. Fraedrich, J. Schneider, and R. Westermann, "Exploring the millennium run–scalable rendering of large-scale cosmological datasets," *IEEE Trans. Vis. and Comput. Graph.*, vol. 15, no. 6, pp. 1251–1258, 2009.

[88] S. Frey and T. Ertl, "Load balancing utilizing data redundancy in distributed volume rendering," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2011, pp. 51–60.

[89] W. Fulmer, T. Mahmood, Z. Li, S. Zhang, J. Huang, and A. Lu, "ImWeb: Cross-platform immersive web browsing for online 3D neuron database exploration," in *Proc. 24th Int. Conf. Intell. User Interfaces*, 2019, pp. 367–378.

[90] K. Gao, W.-k. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham, "Using subfiling to improve programming flexibility and performance of parallel shared-file I/O," in *Proc. 2009 Int. Conf. Parallel Process.* Vienna, Austria: IEEE, 2009, pp. 470–477.

[91] W. Ge, R. Sankaran, and J. H. Chen, "Development of a CPU/GPU portable software library for Lagrangian-Eulerian simulations of liquid sprays," *Int. J. Multiphase Flow*, vol. 128, p. 103293, 2020.

[92] T. A. Gillette, K. M. Brown, and G. A. Ascoli, "The DIADEM metric: Comparing multiple reconstructions of the same neuron," *Neuroinformatics*, vol. 9, no. 2-3, pp. 233–245, 2011.

[93] E. Gobbetti, F. Marton, and J. A. Iglesias-Guitián, "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *Vis. Comput.*, vol. 24, no. 7, pp. 797–806, 2008.

[94] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The adaptable input output system. A framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[95] Google, "Snappy." [Online]. Available: https://github.com/google/snappy

[96] ——, "Tilt brush," 2016. [Online]. Available: https://www.tiltbrush.com/

[97] L. Gosink, J. Shalf, K. Stockinger, Kesheng Wu, and W. Bethel, "HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices," in *Proc. 18th Int. Conf. Scientific and Statistical Database Manage.*, 2006, pp. 149–158.

[98] L. Gosink, J. Anderson, E. Bethel, and K. Joy, "Query-driven visualization of time-varying adaptive mesh refinement data," *IEEE Trans. Vis. and Comput. Graph.*, vol. 14, no. 6, pp. 1715–1722, 2008.

[99] M. Gross, C. Lojewski, M. Bertram, and H. Hagen, "Fast implicit KD-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields," in *Proc. Ninth IASTED Int. Conf. Comput. Graph. and Imag.*, 2007, pp. 67–74.

[100] A. V. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. Hansen, "TOD-tree: Task-overlapped direct send tree image compositing for hybrid MPI parallelism and GPUs," *IEEE Trans. Vis. and Comput. Graph.*, vol. 23, no. 6, pp. 1677–1690, 2016.

[101] A. P. Grosset, A. Knoll, and C. Hansen, "Dynamically scheduled region-based image compositing," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2016, pp. 79–88.

[102] A. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. D. Hansen, "TOD-tree: Task-overlapped direct send tree image compositing for hybrid MPI parallelism," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2015, pp. 67–76.

[103] H. Guo, T. Peterka, and A. Glatz, "In situ magnetic flux vortex visualization in time-dependent Ginzburg-Landau superconductor simulations," in *Proc. 2017 IEEE Pacific Vis. Symp.* IEEE, 2017, pp. 71–80.

[104] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, 1984.

[105] 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 Astron.*, vol. 42, pp. 49–65, 2016.

[106] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, "Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach," *IEEE Trans. Vis. and Comput. Graph.*, vol. 18, no. 12, pp. 2285–2294, 2012.

[107] M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski, "Advanced illumination techniques for GPU volume raycasting," SIGGRAPH Asia Tutorials, 2008.

[108] R. Haimes, "pV3 - A distributed system for large-scale unsteady CFD visualization," in *Proc. 32nd Aerosp. Sciences Meeting and Exhibit*, 1994, pp. 1–9.

[109] M. Han, I. Wald, W. Usher, N. Morrical, A. Knoll, and V. Pascucci, "A virtual frame buffer abstraction for parallel rendering of large tiled display walls," in *Proc. IEEE VIS Short Papers*, 2020, pp. 11–15.

[110] M. Han, I. Wald, W. Usher, Q. Wu, F. Wang, V. Pascucci, C. D. Hansen, and C. R. Johnson, "Ray tracing generalized tube primitives: Method and applications," *Comput. Graph. Forum*, vol. 38, no. 3, pp. 467–478, 2019.

[111] G. Harel, J.-B. Lekien, and P. P. Pébaÿ, "Two new contributions to the visualization of AMR grids: I. interactive rendering of extreme-scale 2-dimensional grids II. novel selection filters in arbitrary dimension," CEA, France, Tech. Rep., 2017.

[112] ——, "Visualization and analysis of large-scale, tree-based, adaptive mesh refinement simulations with arbitrary rectilinear geometry," CEA, France, Tech. Rep., 2017.

[113] K. Harms, T. Leggett, B. Allen, S. Coghlan, M. Fahey, C. Holohan, G. McPheeters, and P. Rich, "Theta: Rapid installation and acceptance of an XC40 KNL system," *Concurrency and Computation: Pract. and Experience*, vol. 30, no. 1, p. e4336, 2018.

[114] E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen, "Direct numerical simulation of turbulent combustion: Fundamental insights towards predictive models," *J. Phys.: Conf. Ser.*, vol. 16, pp. 65–79, 2005.

[115] "HDF5 home page," http://www.hdfgroup.org/HDF5/.

[116] A. Heirich, E. Slaughter, M. Papadakis, W. Lee, T. Biedert, and A. Aiken, "In situ visualization with task-based parallelism," in *Proc. In Situ Infrastructures Enabling Extreme-Scale Anal. and Vis.* New York, NY: ACM Press, 2017, pp. 17–21.

[117] K. Heitmann, T. D. Uram, H. Finkel, N. Frontiere, S. Habib, A. Pope, E. Rangel, J. Hollowed, D. Korytov, P. Larsen, B. S. Allen, K. Chard, and I. Foster, "HACC cosmological simulations: First data release," *Astrophysical J. Suppl. Ser.*, vol. 244, p. 17, 2019.

[118] F. Hernell, P. Ljung, and A. Ynnerman, "Local ambient occlusion in direct volume rendering," *IEEE Trans. Vis. and Comput. Graph.*, vol. 16, no. 4, pp. 548–559, 2010.

[119] D. Hoang, P. Klacansky, H. Bhatia, P.-T. Bremer, P. Lindstrom, and V. Pascucci, "A study of the trade-off between reducing precision and reducing resolution for data analysis and visualization," *IEEE Trans. Vis. and Comput. Graph.*, vol. 25, no. 1, pp. 1193–1203, 2019.

[120] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P.-T. Bremer, and V. Pascucci, "Efficient and flexible hierarchical data layouts for a unified encoding of scalar field precision and resolution," *IEEE Trans. Vis. and Comput. Graph.*, vol. 27, no. 2, pp. 603–613, 2021.

[121] W. Hong, F. Qiu, and A. Kaufman, "GPU-based object-order ray-casting for large datasets," in *Proc. Volume Graph. 2005*. IEEE, 2005, pp. 177–185.

[122] M. Hopf and T. Ertl, "Hierarchical splatting of scattered data," in *Proc. 14th IEEE Vis.* IEEE Computer Society, 2003, p. 8.

[123] M. Howison, A. Adelmann, E. W. Bethel, A. Gsell, B. Oswald, and Prabhat, "H5hut: A high-performance I/O library for particle-based simulations," in *Proc. 2010 IEEE Int. Conf. On Cluster Comput. Workshops and Posters (CLUSTER WORKSHOPS)*. IEEE, 2010, pp. 1–8.

[124] M. Howison, E. W. Bethel, and H. Childs, "MPI-hybrid parallelism for volume rendering on large, multi-core systems," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2010, pp. 1–10.

[125] W. M. Hsu, "Segmented ray casting for data parallel volume rendering," in *Proc. 1993 Symp. Parallel Rendering*, 1993, pp. 7–14.

[126] "HTC vive," www.vive.com/us/product/, Apr. 2016.

[127] Y. Huang, J. Peng, C.-C. J. Kuo, and M. Gopi, "A generic scheme for progressive point cloud coding," *IEEE Trans. Vis. and Comput. Graph.*, vol. 14, no. 2, pp. 440–453, 2008.

[128] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "The quantized kd-tree: Efficient ray tracing of compressed point clouds," in *Proc. 2006 IEEE Symp. Interactive Ray Tracing*. Salt Lake City, UT, USA: IEEE, 2006, pp. 105–113.

[129] S. Ibrahim, T. Stitt, M. Larsen, and C. Harrison, "Interactive in situ visualization and analysis using Ascent and Jupyter," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2019, pp. 44–48.

[130] M. Ikits and D. Brederson, "The visual haptic workbench," in *The Viualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Burlington, VT: Butterworth-Heinemann, 2005, pp. 431–447.

[131] I. T. Iliev, G. Mellema, K. Ahn, P. R. Shapiro, Y. Mao, and U.-L. Pen, "Simulating cosmic reionization: How large a volume is large enough?" *Monthly Notices Roy. Astronomical Soc.*, vol. 439, no. 1, pp. 725–743, 2014.

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

[133] T. Ize, C. Brownlee, and C. D. Hansen, "Real-time ray tracer for visualizing massive models on a cluster," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2011, pp. 61–69.

[134] H. Jacinto, R. Kéchichian, M. Desvignes, R. Prost, and S. Valette, "A web interface for 3D visualization and interactive segmentation of medical images," in *Proc. 17th Int. Conf. 3D Web Technol.*, 2012, pp. 51–58.

[135] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," *login Usenix Mag.*, vol. 44, no. 3, pp. 12–16, 2019.

[136] B. Jeong, P. A. Navrátil, K. P. Gaither, G. Abram, and G. P. Johnson, "Configurable data prefetching scheme for interactive visualization of large-scale volume data," in *Proc. Vis. and Data Anal.*, 2012, pp. 204–217.

[137] 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: Proc. 2003 ACM/IEEE Conf. Supercomputing*, 2003.

[138] I. T. Joliffe, *Principal Component Analysis*. New York, NY: Springer, 1986.

[139] D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski, "A survey of volumetric illumination techniques for interactive volume rendering: A survey of volumetric illumination techniques," *Comput. Graph. Forum*, vol. 33, no. 1, pp. 27–51, 2014.

[140] S. Jourdain, U. Ayachit, and B. Geveci, "ParaViewWeb: A web framework for 3D visualization and data processing," *Int. J. Comput. Inf. Syst. and Ind. Manage. Appl.*, vol. 3, pp. 870–877, 2011.

[141] R. Kaehler, T. Abel, and H.-C. Hege, "Simultaneous GPU-assisted raycasting of unstructured point sets and volumetric grid data," in *Proc. Eurographics/IEEE VGTC Symp. Volume Graph.*, 2007, pp. 49–56.

[142] A. Kageyama and T. Yamada, "An approach to exascale visualization: Interactive viewing of in-situ visualization," *Comput. Phys. Commun.*, vol. 185, no. 1, pp. 79–85, 2014.

[143] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Global Address Space Program. Models*. ACM, 2014, pp. 1–11.

[144] T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees," in *Proc. Fourth ACM SIGGRAPH/Eurographics Conf. High-Perform. Graph.* Eurographics Association, 2012, pp. 33–37.

[145] 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 *Proc. IEEE Symp. Large Data Anal. and Vis.* IEEE, 2011, pp. 65–72.

[146] Kitware, "VTK.js." [Online]. Available: https://github.com/Kitware/vtk-js

[147] A. Knoll, I. Wald, S. Parker, and C. Hansen, "Interactive isosurface ray tracing of large octree volumes," in *Proc. IEEE Symp. Interactive Ray Tracing*, 2006, pp. 115–124.

[148] J. Kress, S. Klasky, N. Podhorszki, J. Choi, H. Childs, and D. Pugmire, "Loosely coupled in situ visualization: A perspective on why it's here to stay," in *Proc. First Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2015, pp. 1–6.

[149] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Comparing the efficiency of in situ visualization paradigms at scale," in *Proc. High Perform. Comput.*, 2019, pp. 99–117.

[150] S. Kumar, C. Christensen, J. A. Schmidt, P.-T. Bremer, E. Brugger, V. Vishwanath, P. Carns, H. Kolla, R. Grout, J. Chen, M. Berzins, G. Scorzelli, and V. Pascucci, "Fast multiresolution reads of massive simulation datasets," in *Proc. Supercomputing*, vol. 8488. New York, NY: Springer, 2014, pp. 314–330.

[151] 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 *SC '14: Proc. Int. Conf. for High Perform. Comput., Networking, Storage and Anal.* IEEE, 2014, pp. 413–423.

[152] S. Kumar, D. Hoang, S. Petruzza, J. Edwards, and V. Pascucci, "Reducing network congestion and synchronization overhead during aggregation of hierarchical data," in *Proc. 2017 IEEE 24th Int. Conf. High Perform. Comput. (HiPC)*, 2017, pp. 223–232.

[153] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, and M. Berzins, "Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation," in *Proc. Supercomputing Frontiers*, 2018, pp. 219–240.

[154] S. Kumar, S. Petruzza, W. Usher, and V. Pascucci, "Spatially-aware parallel I/O for particle data," in *Proc. 48th Int. Conf. Parallel Process. - ICPP 2019*. Kyoto, Japan: ACM Press, 2019, pp. 1–10.

[155] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. E. Papka, J. Chen, and V. Pascucci, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *Proc. Int. Conf. High Perform. Comput., Networking, Storage and Anal.* IEEE Computer Society Press, 2012, pp. 1–11.

[156] 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 *Proc. 2011 IEEE Int. Conf. Cluster Comput.* IEEE, 2011, pp. 103–111.

[157] B. Laha, D. A. Bowman, and J. J. Socha, "Effects of VR system fidelity on analyzing isosurface visualization of volume datasets," *IEEE Trans. Vis. and Comput. Graph.*, vol. 20, no. 4, pp. 513–522, 2014.

[158] B. Laha, K. Sensharma, J. D. Schiffbauer, and D. A. Bowman, "Effects of immersion on visual analysis of volume data," *IEEE Trans. Vis. and Comput. Graph.*, vol. 18, no. 4, pp. 597–606, 2012.

[159] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The ALPINE in situ infrastructure: Ascending from the ashes of strawman," in *Proc. In Situ Infrastructures Enabling Extreme-Scale Anal. and Vis.*, 2017, pp. 42–46.

[160] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, "Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes," in *Proc. First Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.* ACM Press, 2015, pp. 30–35.

[161] M. Larsen, A. Woods, N. Marsaglia, A. Biswas, S. Dutta, C. Harrison, and H. Childs, "A flexible system for in situ triggers," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis. - ISAV '18*. Dallas, Texas: ACM Press, 2018, pp. 1–6.

[162] "LAStools." [Online]. Available: https://github.com/LAStools/LAStools/

[163] 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

[164] M. Le Muzic, J. Parulek, A.-K. Stavrum, and I. Viola, "Illustrative visualization of molecular reactions using omniscient intelligence and passive agents," *Comput. Graph. Forum*, vol. 33, no. 3, pp. 141–150, 2014.

[165] M. Lee, R. Ulerich, N. Malaya, and R. D. Moser, "Experiences from leadership computing in simulations of turbulent fluid flows," *Comput. in Sci. Eng.*, vol. 16, no. 5, pp. 24–31, 2014.

[166] M. Lee, N. Malaya, and R. D. Moser, "Petascale direct numerical simulation of turbulent channel flow on Up, to 786K Cores," in *Proc. Int. Conf. High Perform. Computing, Networking, Storage and Analysis*, ser. SC'13, 2013.

[167] M. Lee and R. Moser, "Direct numerical simulation of turbulent channel flow up to $\mathrm{Re}_\tau \approx 5200$," *J. Fluid Mechanics*, vol. 11, no. 4, pp. 943–945, June 2015.

[168] S. Lefebvre, S. Hornus, and F. Neyret, "Octree textures on the GPU," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds. Boston, MA: Addison-Wesley, 2005, p. 13.

[169] J. K. Li and K.-L. Ma, "P4: Portable parallel processing pipelines for interactive information visualization," *IEEE Trans. Vis. and Comput. Graph.*, vol. 26, no. 3, pp. 1548–1561, 2018.

[170] ——, "P5: Portable progressive parallel processing pipelines for interactive data analysis and visualization," *IEEE Trans. Vis. and Comput. Graph.*, vol. 26, no. 1, pp. 1151–1160, 2019.

[171] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. and Comput. Graph.*, vol. 20, no. 12, pp. 2674–2683, 2014.

[172] B. Liu, G. J. Clapworthy, F. Dong, and E. Wu, "Parallel marching blocks: A practical isosurfacing algorithm for large data on many-core architectures," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 211–220, 2016.

[173] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks: HELLO ADIOS," *Concurrency and Computation: Pract. and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[174] Y. Liu, "The DIADEM and beyond," *Neuroinformatics*, vol. 9, no. 2-3, pp. 99–102, 2011.

[175] Y. Livnat, H.-W. Shen, and C. R. Johnson, "A near optimal isosurface extraction algorithm using the span space," *IEEE Trans. Vis. and Comput. Graph.*, vol. 2, no. 1, pp. 73–84, 1996.

[176] P. Ljung, C. Lundstrom, A. Ynnerman, and K. Museth, "Transfer function based adaptive decompression for volume rendering of large medical data sets," in *Proc. IEEE Symp. Volume Vis. and Graph.*, 2004, pp. 25–32.

[177] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.* IEEE, 2009, pp. 1–10.

[178] J. Loiseau, H. Lim, B. K. Bergen, N. D. Moss, and F. Alin, "FleCSPH: A parallel and distributed smoothed particle hydrodynamics framework based on FleCSI," in *Proc. 2018 Int. Conf. High Perform. Comput. & Simul. (HPCS)*, 2018, pp. 484–491.

[179] W. E. Lorenson and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, 1987.

[180] B. Loring, W. Bethel, M. Wofl, J. Kress, S. Rizzi, J. Gu, J. Logan, and N. Ferrier, "Improving performance and portability of M-to-N processing and data redistribution in in transit analysis and visualization," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2020, p. 11.

[181] L. Luo, E. M. Callaway, and K. Svoboda, "Genetic dissection of neural circuits," *Neuron*, vol. 57, no. 5, 2008.

[182] K.-L. Ma, "Runtime volume visualization for parallel CFD," in *Parallel Comput. Fluid Dyn. 1995*. North-Holland, 1996, pp. 307–314.

[183] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Comput. Graph. and Appl.*, vol. 14, no. 4, pp. 59–68, 1994.

[184] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit," *Comput. Phys. Commun.*, vol. 126, pp. 330–354, 2000.

[185] C. Magliaro, A. L. Callara, N. Vanello, and A. Ahluwalia, "Gotta trace 'em all: A mini-review on tools and procedures for segmenting single neurons toward deciphering the structural connectome," *Frontiers in Bioengineering and Biotechnology*, vol. 7, p. 202, 2019.

[186] N. Malaya, D. McDougall, C. Michoski, M. Lee, and C. S. Simmons, "Experiences porting scientific applications to the intel (KNL) xeon phi platform," in *Proc. Pract. and Experience in Adv. Res. Comput. 2017 Sustainability, Success and Impact*, 2017.

[187] S. Martin, H.-W. Shen, and P. McCormick, "Load-balanced isosurfacing on multi-GPU clusters," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2010, pp. 91–100.

[188] MBF Bioscience, "NeuroLucida 11.08."

[189] T. McDonald, W. Usher, N. Morrical, A. Gyulassy, S. Petruzza, F. Federer, A. Angelucci, and V. Pascucci, "Improving the usability of virtual reality neuron tracing with topological elements," *IEEE Trans. Vis. and Comput. Graph.*, vol. 27, no. 2, pp. 744–754, 2021.

[190] E. Meijering, "Neuron tracing in perspective," *Cytometry Part A*, vol. 77A, no. 7, pp. 693–704, 2010.

[191] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: A unified heterogeneous task scheduling and runtime system," in *Proc. SC Companion: High Perform. Comput., Networking Storage and Anal.* IEEE, 2012, pp. 2441–2448.

[192] M. M. Mobeen and L. Feng, "High-performance volume rendering on the ubiquitous WebGL platform," in *Proc. 2012 IEEE 14th Int. Conf. High Perform. Comput. and Communication & 2012 IEEE 9th Int. Conf. Embedded Softw. and Syst.*, 2012, pp. 381–388.

[193] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *IEEE Comput. Graph. and Appl.*, vol. 14, no. 4, pp. 23–32, 1994.

[194] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An image compositing solution at scale," in *Proc. 2011 Int. Conf. for High Perform. Comput., Networking, Storage and Anal.*, 2011, pp. 1–10.

[195] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, M. E. Papka, and S. Klasky, "Examples of in transit visualization," in *Proc. 2nd Int. Workshop Petascale Data Analytics: Challenges and Opportunities*. ACM, 2011, pp. 1–6.

[196] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the visualization toolkit for massively threaded architectures," *IEEE Comput. Graph. and Appl.*, vol. 36, no. 3, pp. 48–58, 2016.

[197] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Research, Armonk, NY, Tech. Rep., 1966.

[198] E. Murray, J. Cho, D. Goodwin, T. Ku, J. Swaney, S.-Y. Kim, H. Choi, Y.-G. Park, J.-Y. Park, A. Hubbert, M. McCue, S. Vassallo, N. Bakh, M. Frosch, V. Wedeen, H. Seung, and K. Chung, "Simple, scalable proteomic imaging for high-dimensional profiling of intact systems," *Cell*, vol. 163, no. 6, pp. 1500–1514, 2015.

[199] F. Mwalongo, M. Krone, G. Reina, and T. Ertl, "State-of-the-art report in web-based visualization," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 553–575, 2016.

[200] B. Nam and A. Sussman, "Improving access to multi-dimensional self-describing scientific datasets," in *Proc. 3rd IEEE/ACM Int. Symp. Cluster Comput. and the Grid*, 2003, pp. 172–179.

[201] P. A. Navrátil, D. Fussell, C. Lin, and H. Childs, "Dynamic scheduling for large-scale distributed-memory ray tracing," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2012, pp. 61–70.

[202] U. Neumann, "Parallel volume-rendering algorithm performance on mesh-connected multicomputers," in *Proc. 1993 IEEE Parallel Rendering Symp.* IEEE, 1993, pp. 97–104.

[203] "Oculus rift development kit 2," www.oculus.com/en-us/dk2/, July 2014.

[204] "Oculus quest 2," https://www.oculus.com/quest-2/, Oct. 2020.

[205] P. E. O'Neil, "MODEL 204 architecture and performance," in *High Perform. Transaction Syst.*, G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, D. Gawlick, M. Haynie, and A. Reuter, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 359, pp. 39–59.

[206] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea, "End-to-end i/o portfolio for the summit supercomputing ecosystem," in *Proc. Int. Conf. for High Perform. Comput., Networking, Storage and Anal.*, ser. SC '19. Association for Computing Machinery, 2019.

[207] B. W. O'shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing enzo, an AMR cosmology application," in *Adaptive Mesh Refinement-Theory and Applications*, ser. Lecture Notes in Computational Science and Engineering. Springer, 2005.

[208] R. Pajarola, "Efficient level-of-details for point based rendering," in *Proc. Sixth IASTED Int. Conf. Comput. Graph. and Imag., Honolulu*, 2003, pp. 141–146.

[209] R. Pajarola, M. Sainz, and R. Lario, "XSplat: External memory multiresolution point visualization," in *Proc. IASTED Int. Conf. Vis., Imag. and Image Process., Benidorm*, 2005, pp. 628–633.

[210] K. Palmerius, M. Cooper, and A. Ynnerman, "Haptic rendering of dynamic volumetric data," *IEEE Trans. Vis. and Comput. Graph.*, vol. 14, no. 2, pp. 263–276, 2008.

[211] V. D. Paola, A. Holtmaat, G. Knott, S. Song, L. Wilbrecht, P. Caroni, and K. Svoboda, "Cell type-specific structural plasticity of axonal branches and boutons in the adult neocortex," *Neuron*, vol. 49, no. 6, 2006.

[212] H. Park, D. Fussell, and P. Navrátil, "SpRay: Speculative ray scheduling for large data visualization," in *Proc. IEEE Symp. Large Data Anal. and Vis.*, 2018, pp. 77–86.

[213] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive ray tracing for isosurface rendering," in *Proc. Vis. '98*, 1998, pp. 233–238.

[214] S. G. Parker and C. R. Johnson, "SCIRun: A scientific programming environment for computational steering," in *Proc. 1995 ACM/IEEE Conf. Supercomputing*. ACM, 1995, pp. 52–72.

[215] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proc. 2001 ACM/IEEE Conf. Supercomputing*. IEEE, 2001, pp. 45–45.

[216] ——, "Hierarchical indexing for out-of-core access to multi-resolution data," in *Proc. Hierarchical and Geometrical Methods in Scientific Vis.* Berlin: Springer, 2003, pp. 225–241.

[217] M. Pauly, M. Gross, and L. P. Kobbelt, "Efficient simplification of point-sampled surfaces," in *Proc. Conf. Vis. '02*. IEEE Computer Society, 2002, pp. 163–170.

[218] P. P. Pébaÿ, G. Borghesi, H. Kolla, J. C. Bennett, and S. Treichler, "A novel shard-based approach for asynchronous many-task models for in situ analysis," in *Proc. In Situ Infrastructures Enabling Extreme-Scale Anal. and Vis.*, 2017, pp. 27–31.

[219] P. Pébaÿ, J. C. Bennett, D. Hollman, S. Treichler, P. S. McCormick, C. M. Sweeney, H. Kolla, and A. Aiken, "Towards asynchronous many-task in situ data analysis using legion," in *Proc. 2016 IEEE Int. Parallel and Distrib. Process. Symp. Workshops*. IEEE, 2016, pp. 1033–1037.

[220] H. Peng, Z. Ruan, F. Long, J. H. Simpson, and E. W. Myers, "V3D enables real-time 3D visualization and quantitative analysis of large-scale biological image data sets," *Nature Biotechnology*, vol. 28, no. 4, pp. 348–353, 2010.

[221] F. Perez and B. E. Granger, "IPython: A system for interactive scientific computing," *Comput. in Sci. & Eng.*, vol. 9, no. 3, pp. 21–29, 2007.

[222] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur, "A configurable algorithm for parallel image-compositing applications," in *Proc. Conf. High Perform. Comput. Networking, Storage and Anal.*, 2009, pp. 1–10.

[223] S. Petruzza, S. Treichler, V. Pascucci, and P.-T. Bremer, "BabelFlow: An embedded domain specific language for parallel analysis and visualization," in *Proc. IEEE Int. Parallel and Distrib. Process. Symp.*, 2018, pp. 463–473.

[224] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *Proc. Innovative Parallel Comput. (InPar), 2012*. IEEE, 2012, pp. 1–13.

[225] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, 1995.

[226] Prabhat, A. Forsberg, M. Katzourin, K. Wharton, and M. Slater, "A comparative study of desktop, fishtank, and cave systems for the exploration of volume rendered confocal data sets," *IEEE Trans. Vis. and Comput. Graph.*, vol. 14, no. 3, pp. 551–563, 2008.

[227] S. Prohaska, A. Hutanu, R. Kahler, and H.-C. Hege, "Interactive exploration of large remote micro-CT scans," in *Proc. IEEE Vis. 2004*, 2004, pp. 345–352.

[228] M. Raji, A. Hota, T. Hobson, and J. Huang, "Scientific visualization as a microservice," *IEEE Trans. Vis. and Comput. Graph.*, vol. 26, no. 4, pp. 1760–1774, 2018.

[229] M. Raji, A. Hota, and J. Huang, "Scalable web-embedded volume rendering," in *Proc. IEEE 7th Symp. Large Data Anal. and Vis.* IEEE, 2017, pp. 45–54.

[230] B. Rathke, I. Wald, K. Chiu, and C. Brownlee, "SIMD parallel ray tracing of homogeneous polyhedral grids," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2015, pp. 33–41.

[231] T. Rau, P. Gralka, O. Fernandes, G. Reina, S. Frey, and T. Ertl, "The impact of work distribution on in situ visualization: A case study," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2019, pp. 17–22.

[232] K. Reda, A. Knoll, K.-i. Nomura, M. E. Papka, A. E. Johnson, and J. Leigh, "Visualizing large-scale atomistic simulations in ultra-resolution immersive environments." in *Proc. IEEE Symp. Large-Scale Data Anal. and Vis.*, 2013, pp. 59–65.

[233] F. Reichl, M. Treib, and R. Westermann, "Visualization of big SPH simulations via compressed octree grids," in *Proc. IEEE Int. Conf. Big Data*. IEEE, 2013, pp. 71–78.

[234] E. Reinhard, A. Chalmers, and F. W. Jansen, "Hybrid scheduling for parallel rendering using coherent ray tasks," in *Proc. 1999 IEEE Symp. Parallel Vis. and Graph.* IEEE Computer Society, 1999, pp. 21–28.

[235] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath, "Large-scale co-visualization for LAMMPS using vl3," in *Proc. IEEE 5th Symp. Large Data Anal. and Vis.* IEEE, 2015, pp. 141–142.

[236] ——, "Large-scale parallel visualization of particle-based simulations using point sprites and level-of-detail," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2015, pp. 1–10.

[237] D. Rotem, K. Stockinger, and K. Wu, "Optimizing candidate check costs for bitmap indices," in *Proc. 14th ACM Int. Conf. Inf. and Knowl. Manage.*, 2005, pp. 648–655.

[238] S. Rusinkiewicz and M. Levoy, "QSplat: A multiresolution point rendering system for large meshes," in *Proc. 27th Annu. Conf. Comput. Graph. and Interactive Techn.*, 2000, pp. 343–352.

[239] T. Saad and J. C. Sutherland, "Wasatch: An architecture-proof multiphysics development environment using a domain specific language and graph theory," *J. Comput. Sci.*, vol. 17, pp. 639–646, 2016.

[240] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of PCs," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop Graph. Hardware*. The Eurographics Association, 2000, pp. 97–108.

[241] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Burlington, MA: Morgan Kaufmann, 2006.

[242] Sandia National Laboratories, "LAMMPS molecular dynamics simulator." [Online]. Available: lammps.sandia.gov

[243] ——, "Coupling LAMMPS to other codes," Accessed Jan. 2020. [Online]. Available: lammps.sandia.gov/doc/Howto_couple.html

[244] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Trans. Vis. and Comput. Graph.*, vol. 23, no. 1, pp. 341–350, 2017.

[245] A. Satyanarayan, K. Wongsuphasawat, and J. Heer, "Declarative interaction design for data visualization," in *Proc. 27th Annu. ACM Symp. User Interface Softw. and Technol.* New York, NY: ACM Press, 2014, pp. 669–678.

[246] K. Schatz, C. Muller, M. Krone, J. Schneider, G. Reina, and T. Ertl, "Interactive visual exploration of a trillion particles," in *Proc. IEEE 6th Symp. Large Data Anal. and Vis.*, 2016, pp. 56–64.

[247] C. Scheiblauer, "Interactions with gigantic point clouds," Ph.D. dissertation, Inst. of Comput. Graph. and Algorithms, Vienna Univ. of Technol., Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, 2014.

[248] L. Schmitz, L. F. Scheidegger, D. K. Osmari, C. A. Dietrich, and J. L. D. Comba, "Efficient and quality contouring algorithms on the GPU," *Comput. Graph. Forum*, vol. 29, no. 8, pp. 2569–2578, 2010.

[249] W. Schroeder, K. Martin, and B. Lorenson, *The Visualization Toolkit*, 4th ed. Clifton Park, NY: Kitware, 2006.

[250] W. Schroeder, R. Maynard, and B. Geveci, "Flying edges: A high-performance scalable isocontouring algorithm," in *Proc. IEEE 5th Symp. Large Data Anal. and Vis.* IEEE, 2015, pp. 33–40.

[251] M. Schütz, "Potree: Rendering large point clouds in web browsers," Ph.D. dissertation, Vienna Univ. of Technol., Vienna, Austria, 2016.

[252] M. Schütz, K. Krösl, and M. Wimmer, "Real-time continuous level of detail rendering of point clouds," in *Proc. IEEE Conf. Virtual Reality and 3D User Interfaces*, 2019, pp. 103–110.

[253] M. Sedlmair, M. Meyer, and T. Munzner, "Design study methodology: Reflections from the trenches and the stacks," *IEEE Trans. Vis. and Comput. Graph.*, vol. 18, no. 12, pp. 2431–2440, 2012.

[254] 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: Proc. 2008 ACM/IEEE Conf. Supercomputing*, Nov 2008.

[255] T. Sherif, N. Kassis, M.-Ã. Rousseau, R. Adalat, and A. C. Evans, "BrainBrowser: Distributed, web-based neurological data visualization," *Frontiers in Neuroinformatics*, vol. 8, p. 89, 2015.

[256] R. Sicat, J. Li, J. Choi, M. Cordeil, W.-K. Jeong, B. Bach, and H. Pfister, "DXR: A toolkit for building immersive data visualizations," *IEEE Trans. Vis. and Comput. Graph.*, vol. 25, no. 1, pp. 715–725, 2019.

[257] A. Siegel, E. Draeger, J. Deslippe, A. Dubey, T. Evans, T. Germann, and W. Hart, "Early application results on pre-exascale architecture with analysis of performance challenges and projections WBS 2.2, milestone PM-AD-1080," Office of Advanced Scientific Computing Research, Washington, DC, United States, Tech. Rep., 2020.

[258] R. R. Sinha and M. Winslett, "Multi-resolution bitmap indexes for scientific data," *ACM Trans. Database Syst.*, vol. 32, no. 3, pp. 16–55, 2007.

[259] 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," Stanford, CA, United States, Tech. Rep., 2014.

[260] 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.

[261] S. Soltanian-Zadeh, K. Sahingur, S. Blau, Y. Gong, and S. Farsiu, "Fast and robust active neuron segmentation in two-photon calcium imaging using spatiotemporal deep learning," *Proc. Nat. Acad. Sciences*, vol. 116, no. 17, pp. 8554–8563, 2019.

[262] M. Son and S.-E. Yoon, "Timeline scheduling for out-of-core ray batching," in *Proc. High Perform. Graph.* New York, NY: ACM Press, 2017, pp. 1–10.

[263] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "Stampede 2: The evolution of an XSEDE supercomputer," in *Proc. Pract. and Experience in Adv. Res. Comput. 2017 Sustainability, Success and Impact*, ser. PEARC17. Association for Computing Machinery, 2017.

[264] K. Stockinger, J. Shalf, W. Bethel, and K. Wu, "DEX: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization," Lawrence Berkeley National Laboratory, Berkeley, CA, United States, Tech. Rep. LBNL–57023, 837248, 2005.

[265] K. Stockinger, J. Shalf, K. Wu, and E. W. Bethel, "Query-driven visualization of large data sets," in *Proc. IEEE Vis.*, 2005, pp. 167–174.

[266] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Trans. Vis. and Comput. Graph.*, vol. 8, no. 1, pp. 52–65, 2002.

[267] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. Ahrens, "Taming massive distributed datasets: Data sampling using bitmap indices," in *Proc. 22nd Int. Symp. High-Perform. Parallel and Distrib. Comput.* New York, NY: ACM, 2013, pp. 13–24.

[268] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *Proc. 24th Int. Symp. High-Perform. Parallel and Distrib. Comput.*, 2015, pp. 61–72.

[269] Tae-Joon Kim, Xin Sun, and Sung-Eui Yoon, "T-ReX: Interactive global illumination of massive models on heterogeneous computing resources," *IEEE Trans. Vis. and Comput. Graph.*, vol. 20, no. 3, pp. 481–494, 2014.

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

[271] D. Thompson, S. Jourdain, A. Bauer, B. Geveci, R. Maynard, R. R. Vatsavai, and P. O'Leary, "In situ summarization with VTK-m," in *Proc. In Situ Infrastructures Enabling Extreme-Scale Anal. and Vis.* New York, NY: ACM Press, 2017, pp. 32–36.

[272] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'hallaron, "From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing," in *Proc. 2006 ACM/IEEE Conf. Supercomputing*. ACM, 2006, pp. 91–106.

[273] U. U. Turuncoglu, B. Önol, and M. Ilicak, "A new approach for in situ analysis in fully coupled earth system models," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2019, pp. 6–11.

[274] W. Usher, J. Amstutz, C. Brownlee, A. Knoll, and I. Wald, "Progressive CPU volume rendering with sample accumulation," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2017, pp. 21–30.

[275] W. Usher, J. Amstutz, J. Günther, A. Knoll, G. P. Johnson, C. Brownlee, A. Hota, B. Cherniak, T. Rowley, J. Jeffers, and V. Pascucci, "Scalable CPU ray tracing for in situ visualization using OSPRay," in *In Situ Vis. for Comput. Sci. (in press)*, H. Childs, C. Garth, and J. Bennett, Eds., 2021, ch. 4.

[276] W. Usher, X. Huang, S. Petruzza, S. Kumar, S. R. Slattery, S. T. Reeve, F. Wang, C. R. Johnson, and V. Pascucci, "Adaptive spatially aware I/O for multiresolution particle data layouts," in *Proc. 35th IEEE Int. Parallel & Distrib. Process. Symp. (IPDPS)*, 2021, p. 10.

[277] W. Usher, P. Klacansky, F. Federer, P.-T. Bremer, A. Knoll, J. Yarch, A. Angelucci, and V. Pascucci, "A virtual reality visualization tool for neuron tracing," *IEEE Trans. Vis. and Comput. Graph.*, vol. 24, no. 1, pp. 994–1003, 2018.

[278] W. Usher, H. Park, M. Lee, P. Navrátil, D. Fussell, and V. Pascucci, "A simulation-oblivious data transport model for flexible in transit visualization," in *In Situ Vis. for Comput. Sci. (in press)*, H. Childs, C. Garth, and J. Bennett, Eds., 2021, ch. 7.

[279] W. Usher and V. Pascucci, "Interactive visualization of terascale data in the browser: Fact or fiction?" in *Proc. IEEE 10th Symp. Large Data Anal. and Vis.*, 2020, pp. 27–36.

[280] W. Usher, S. Rizzi, I. Wald, J. Amstutz, J. Insley, V. Vishwanath, N. Ferrier, M. E. Papka, and V. Pascucci, "libIS: A lightweight library for flexible in transit visualization," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2018, pp. 33–38.

[281] W. Usher, I. Wald, J. Amstutz, J. Günther, C. Brownlee, and V. Pascucci, "Scalable ray tracing using the distributed framebuffer," *Comput. Graph. Forum*, vol. 38, no. 3, pp. 455–466, 2019.

[282] W. Usher, I. Wald, A. Knoll, M. E. Papka, and V. Pascucci, "In situ exploration of particle simulations with CPU ray tracing," *Supercomputing Frontiers and Innovations*, vol. 3, no. 4, pp. 4–18, 2016.

[283] T. Vierjahn, A. Schnorr, B. Weyers, D. Denker, I. Wald, C. Garth, T. W. Kuhlen, and B. Hentschel, "Interactive exploration of dissipation element geometry," in *Proc. Eurographics Symp. Parallel Graph. and Vis.* The Eurographics Association, 2017, pp. 53–62.

[284] 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 *Proc. 2011 Int. Conf. for High Perform. Comput., Networking, Storage and Anal.* ACM, 2011, pp. 1–11.

[285] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and I/O acceleration on leadership-class systems," in *Proc. IEEE Symp. Large Data Anal. and Vis.* IEEE, 2011, pp. 9–14.

[286] A. Vlachos, "Advanced VR rendering," 2015, talk at the Game Developer's Conference.

[287] E. L. Vote, "A new methodology for archaeological analysis," Ph.D. dissertation, Brown Univ., Providence, RI, United States, 2001.

[288] I. Wald, C. Benthin, and P. Slusallek, "A flexible and scalable rendering engine for interactive 3D graphics," Saarland University, Saarbrüken, Germany, Tech. Rep., 2002.

[289] I. Wald, C. Brownlee, W. Usher, and A. Knoll, "CPU volume rendering of adaptive mesh refinement data," in *Proc. SIGGRAPH Asia 2017 Symp. Vis.*, 2017, pp. 1–8.

[290] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel, "Faster isosurface ray tracing using implicit KD-trees," *IEEE Trans. Vis. and Comput. Graph.*, vol. 11, no. 5, pp. 562–572, 2005.

[291] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil, "OSPRay – a CPU ray tracing framework for scientific visualization," *IEEE Trans. Vis. and Comput. Graph.*, vol. 23, no. 1, pp. 931–940, 2017.

[292] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka, "CPU ray tracing large particle data with balanced P-k-d trees," in *Proc. 2015 IEEE Scientific Vis. Conf. (SciVis)*, 2015, pp. 57–64.

[293] I. Wald, P. Slusallek, and C. Benthin, "Interactive distributed ray tracing of highly complex models," in *In Proc. Rendering Techn. 2001*, 2001, pp. 277–288.

[294] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient CPU ray tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 1–8, 2014.

[295] F. Wang, N. Marshak, W. Usher, C. Burstedde, A. Knoll, T. Heister, and C. R. Johnson, "CPU ray tracing of tree-based adaptive mesh refinement data," *Comput. Graph. Forum*, vol. 39, no. 3, pp. 1–12, 2020.

[296] F. Wang, I. Wald, and C. R. Johnson, "Interactive rendering of large-scale volumes on multi-core CPUs," in *Proc. IEEE 9th Symp. Large Data Anal. and Vis.*, 2019, pp. 27–36.

[297] F. Wang, I. Wald, Q. Wu, W. Usher, and C. R. Johnson, "CPU isosurface ray tracing of adaptive mesh refinement data," *IEEE Trans. Vis. and Comput. Graph.*, vol. 25, no. 1, pp. 1142–1151, 2019.

[298] K.-C. Wang, N. Shareef, and H.-W. Shen, "Image and distribution based volume rendering for large data sets," in *Proc. 2018 IEEE Pacific Vis. Symp.* IEEE, 2018, pp. 26–35.

[299] "WebAssembly." [Online]. Available: https://webassembly.org/

[300] "WebGPU." [Online]. Available: https://gpuweb.github.io/gpuweb/

[301] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner, "The structure of the nervous system of the nematode caenorhabditis elegans," *Philos. Trans. Roy. Soc. London B: Biological Sci.s*, vol. 314, no. 1165, pp. 1–340, 1986.

[302] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2011, pp. 101–109.

[303] J. Wilhelms and A. V. Gelder, "Octrees for faster isosurface generation," *ACM Trans. Graph.*, vol. 11, no. 3, pp. 201–227, 1992.

[304] M. Wimmer and C. Scheiblauer, "Instant points: Fast rendering of unprocessed point clouds," in *Proc. Symp. Point-Based Graph.*, 2006, pp. 129–136.

[305] H. K. T. Wong, J. Z. Li, F. Olken, D. Rotem, and L. Wong, "Bit transposition for very large scientific and statistical databases," *Algorithmica*, vol. 1, no. 1-4, pp. 289–309, 1986.

[306] 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," *Comput. Graph. Forum*, vol. 30, no. 3, pp. 1151–1160, 2011.

[307] J. Woodring, M. Petersen, A. Schmeiber, J. Patchett, J. Ahrens, and H. Hagen, "In situ eddy analysis in a high-resolution ocean climate model," *IEEE Trans. Vis. and Comput. Graph.*, vol. 22, no. 1, pp. 857–866, 2016.

[308] 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," *J. Phys.: Conf. Ser.*, vol. 180, p. 012053, 2009.

[309] K. Wu, W. Koegler, J. Chen, and A. Shoshani, "Using bitmap index for interactive exploration of large datasets," in *Proc. 15th Int. Conf. Scientific and Statistical Database Manage.* IEEE, 2003, pp. 65–74.

[310] 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, 2006.

[311] K. Wu, A. Shoshani, and K. Stockinger, "Analyses of multi-level and multi-component compressed bitmap indexes," *ACM Trans. Database Syst.*, vol. 35, no. 1, pp. 1–52, 2010.

[312] K. Wu, A. Shoshani, W.-M. Zhang, J. Lauret, and V. Perevoztchikov, "Grid collector: Using an event catalog to speed up user analysisin distributed environment," Lawrence Berkeley National Laboratory, Berkeley, CA, United States, Tech. Rep., 2004.

[313] K. Wu, K. Stockinger, and A. Shoshani, "Breaking the curse of cardinality on bitmap indexes," in *Proc. Scientific and Statistical Database Manage.*, B. Ludäscher and N. Mamoulis, Eds., vol. 5069. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 348–365.

[314] Q. Wu, W. Usher, S. Petruzza, S. Kumar, F. Wang, I. Wald, V. Pascucci, and C. D. Hansen, "VisIt-OSPRay: Toward an exascale volume visualization system," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2018, pp. 13–23.

[315] B. Yang, J. B. Treweek, R. P. Kulkarni, B. E. Deverman, C.-K. Chen, E. Lubeck, S. Shah, L. Cai, and V. Gradinaru, "Single-cell phenotyping within transparent intact tissue through whole-body clearing," *Cell*, vol. 158, no. 4, pp. 945–958, 2014.

[316] Y. C. Ye, T. Neuroth, F. Sauer, K.-L. Ma, G. Borghesi, A. Konduri, H. Kolla, and J. Chen, "In situ generated probability distribution functions for interactive post hoc visualization and analysis," in *Proc. IEEE 6th Symp. Large Data Anal. and Vis.* IEEE, 2016, pp. 65–74.

[317] Y. C. Ye, Y. Wang, R. Miller, K.-L. Ma, and K. Ono, "In situ depth maps based feature extraction and tracking," in *Proc. IEEE 5th Symp. Large Data Anal. and Vis.* IEEE, 2015, pp. 1–8.

[318] A. Ynnerman, T. Rydell, A. Persson, A. Ernvik, C. Forsell, P. Ljung, and C. Lundström, "Multi-touch table system for medical visualization," in *Proc. 36th Annu. Conf. Eur. Assoc. for Comput. Graph., EuroGraphics*, 2015, pp. 9–12.

[319] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Comput. Graph. and Appl.*, vol. 30, no. 3, pp. 45–57, 2010.

[320] H. Yu, C. Wang, and K.-L. Ma, "Massively parallel volume rendering using 2– 3 swap image compositing," in *SC '08: Proc. 2008 ACM/IEEE Conf. Supercomputing*, 2008, pp. 1–11.

[321] Y. Yucong, R. Miller, and K.-L. Ma, "In situ pathtube visualization with explorable images," in *Proc. Eurographics Symp. Parallel Graph. and Vis.*, 2013, p. 8.

[322] A. Zakai, "Emscripten: An LLVM-to-JavaScript compiler," in *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Languages and Appl. Companion*, 2011, pp. 301–312.

[323] H. C. Zanúz, B. Raffin, O. A. Mures, and E. J. Padrón, "In-transit molecular dynamics analysis with apache flink," in *Proc. Workshop In Situ Infrastructures for Enabling Extreme-Scale Anal. and Vis.*, 2018, pp. 25–32.

[324] D. Z. Zhang, Q. Zou, W. B. VanderHeyden, and X. Ma, "Material point method applied to multiphase flows," *J. Comput. Phys.*, vol. 227, no. 6, pp. 3159–3173, 2008.

[325] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar, "In-situ feature-based objects tracking for large-scale scientific simulations," in *Proc. 2012 SC Companion: High Perform. Comput., Networking Storage and Anal.* IEEE, 2012, pp. 736–740.

[326] 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 *Proc. IEEE Int. Symp. Parallel Distrib. Process.* IEEE, 2010, pp. 1–12.

[327] 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 *Proc. IEEE 27th Int. Symp. Parallel and Distrib. Process.* IEEE, 2013, pp. 320–331.

[328] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling embedded in-situ indexing with DeltaFS," in *Proc. Int. Conf. for High Perform. Comput., Networking, Storage, and Anal.*, 2018, pp. 1–15.