

Low-Overhead In Situ Visualization Using Halo Replay

Jeff Ames*
Duke University

Silvio Rizzi
Argonne National Laboratory

Joseph Insley
Argonne National Laboratory
Northern Illinois University

Saamil Patel
Argonne National Laboratory

Benjamín Hernández
Oak Ridge National Laboratory

Erik W. Draeger
Lawrence Livermore National Laboratory

Amanda Randles
Duke University

ABSTRACT

In situ visualization and analysis is of increasing importance as the compute and I/O gap further widens with the advance to exascale capable computing. Yet, *in situ* methods impose resource constraints leading to the difficult task of balancing simulation code performance and the quality of analysis. Applications with tightly-coupled *in situ* visualization often achieve performance through spatial and temporal downsampling, a tradeoff which risks not capturing transient phenomena at sufficient fidelity. Determining *a priori* visualization parameters such as sampling rate is difficult without time and resource intensive experimentation. We present a method for reducing resource contention between *in situ* visualization and stencil codes on heterogeneous systems. This method permits full resolution replay through recording halos and the communication-free reconstruction of interior values uncoupled from the main simulation. We apply this method in the computational fluid dynamics (CFD) code HARVEY [1] on the Summit supercomputer. We demonstrate minimal-overhead, *in situ* visualization relative to simulation alone, and compare the Halo Replay performance to tightly-coupled *in situ* approaches.

Index Terms: Data reduction—scientific visualization—locality; parallelism—stencils

1 INTRODUCTION

Due to the growing performance gap between processing and I/O, the development of performant *in situ* algorithms is considered a top challenge in extreme-scale analytics [2]. While the scale and complexity of scientific applications grows rapidly, stagnating I/O capabilities preclude detailed exploration of massive simulation datasets through conventional *post hoc* analysis [3]. An example workload gaining traction in biomedical sciences is the massively-parallel simulation of hemodynamics in the human circulatory system with the lattice Boltzmann method (LBM) [4]–[6], where micron scale simulations can require petabytes of memory for system state at a given time step [7]. At such a resolution in complex arterial models, simulating flow over a single cardiac cycle using LBM requires millions of time steps, each with a memory footprint exceeding a petabyte. On modern systems such as Summit, with I/O bandwidth of 2.5 terabits per second, *post hoc* analysis of simulations of this scale is infeasible without down-sampling and risk of missing data of clinical significance. For extreme scale applications like this and the massive datasets they generate *in situ* methods are necessary for analysis at high spatial and temporal fidelity.

Despite progress both in the development of *in situ* methods and in understanding their trade-offs and scaling behavior, determining appropriate *in situ* algorithms for applications with different resource requirements is often not clear-cut [8]–[10]. While *in situ* methods can be orders of magnitude faster than their *post hoc* counterparts,

they often carry overhead with time spent in analysis or visualization routines dwarfing time spent in simulation code [11]. Average iteration time can be reduced through less frequent analysis; however, the trade-off in temporal resolution can be unacceptable for scientific applications where events of interest are short-lived [11], [12]. The transition of top supercomputers to heterogeneous nodes presents a further challenge to balancing *in situ* analysis and simulation workloads [13].

Since there is a large variance in the requirements of extreme-scale scientific applications, there is need to explore new *in situ* techniques for specific workloads to guide researchers’ design decisions. Stencil (nearest-neighbor) codes are an important class of algorithms which form the core of extreme-scale applications in fields such as weather modeling [14], image processing [15], and aeronautics [16]. Therefore improvements in *in situ* analysis targeting stencils have wide applications. We aim to develop a method for *in situ* analysis for stencil codes with minimal overhead to the simulation, that efficiently uses resources on heterogeneous systems and reduces risk of analysis failing to capture events at sufficient fidelity. The novel method we propose has the following characteristics.

- A scalable recording scheme that can rewind simulation state and playback at full resolution over the entire or desired subset of the domain. This allows repeating analysis at increased fidelity without restarting the full scale simulation from a checkpoint.
- Process-independent replay which enables *in transit* work flows where a subdomain is reconstructed and visualized at full resolution with a more cost-effective setup.
- Low overhead to the simulation through utilization of both the CPU and GPU on heterogeneous architectures. Decoupling of simulation and analysis data allows detailed visualization of a subdomain without blocking the simulation.

We describe how this method can be implemented in parallel stencil codes and measure the overhead in a test application using the 19-point 3D LBM stencil. We apply the method to high-resolution *in situ* visualization of vascular flows in HARVEY. The method has several attractive properties in the context of CFD. For visualization at high temporal frequency it has lower overhead to the simulation than other *in situ* schemes and makes more efficient use of limited HPC resources. For simulations at the micron scale involving complex geometries, events can be transient and missed during initial *in situ* analysis. This method allows users to efficiently rewind and catch such events. Further it enables migrating a simulation subdomain to a different system and reproducing results at full resolution. This is useful for complex analysis routines which may be accelerated by custom hardware or when the user requires low-latency rendering on local hardware, such as when viewing results with a Virtual Reality headset.

To evaluate the method’s scaling we measure performance on the Summit supercomputer at scales up to 1024 nodes consisting of 6144 GPUs and 43008 CPU cores. We explore its use cases and tradeoffs through comparison to other *in situ* paradigms using the performance metrics of overhead to the simulation and the rate of visualization.

*e-mail: jeff.ames@duke.edu

2 PREVIOUS WORK

Research into *in situ* analysis has focused on several important areas including the development of production-quality *in situ* frameworks and interfaces [3], [11], [17], data reduction [12], [18], [19], and topology-aware mapping [20].

Here we discuss the advantages and limitations of previous *in situ* approaches.

2.1 In Situ Frameworks

HPC applications in a range of fields face the challenge of visualizing increasingly massive and complex datasets, which has spurred the development of general purpose *in situ* visualization tools such as ParaView Catalyst[3] and VisIt libsim [17], supporting a variety of *in situ* workflows [21]. Prior work has used Catalyst or libsim as *in situ* visualization engines. For example, PHASTA a Navier-Stokes solver based on a stabilized finite element method used Catalyst to scale their interactive visualization to 1,048,576 MPI tasks with a *in situ* overhead of 5 seconds per time step [11].

In situ tools like Catalyst and libsim allow tight coupling between a visualization engine and simulation code. For applications without optimized integration with Catalyst or libsim, *in situ* performance is often limited by memory and communication. Spatial and temporal downsampling are often used to lower the impact of *in situ* code on simulation run time [11]. For some applications the trade-off between speed and analysis fidelity is unacceptable, motivating our approach to decouple the run time of *in situ* routines from that of the main simulation.

2.2 Feature Extraction

Transformation and filtering of simulation data impacts the costs of *in situ* analysis. Capturing state as a series of images reduces I/O demands but may be problematic for massive simulations with transient or unpredictable events, as camera position and capture frequency must be predefined. Other strategies reduce I/O demands through *in situ* data reduction and feature extraction. These techniques capture data likely to be relevant at greater frequency or detail, such as isosurfaces in CFD [12]. In the method proposed by Hamilton *et al.*, checkpoint data is written to burst buffers, and velocity data is extracted only in areas of high vorticity, reducing data written to disk by an order of magnitude [18]. A limitation of this approach is feature extraction thresholds must set in advance which risks culling potentially interesting data.

2.3 Compression

By compressing simulation data while it resides in memory, I/O burden is reduced both when writing to disk, allowing the simulation code to resume when storage commits, and later when reading from disk, enabling analysis to load and operate on data that would otherwise exceed memory capacities. As such, there is interest in *in situ* compression techniques. Achieving 80% compression rates on Tokamak simulation data, the In-situ Sort And-B-spline Error-bounded Lossy Abatement (ISABELA) has both communication-free compression, random access decompression and bounded error [22]. Recent techniques for floating-point data compression such as ZFP have superior speed and compression to ISABELA [19]. These state of the art techniques are used for compressing full domain checkpoints though they can operate on higher dimensional structures and compress temporal data, such as the Halo Buffer presented in our work. Even with high compression rates, data generation in massive scale applications overwhelms I/O capabilities of modern systems without downsampling. To avoid downsampling our approach decreases data size instead through storing lower dimensionality process communication data and reconstructing the full data set on-demand.

2.4 In situ Methods in Lattice Boltzmann codes

Prior work has incorporated *in situ* visualization in sparse geometry lattice Boltzmann-based applications. An implementation in the CFD software HemeLB used parallel raytracing to deliver interactive performance at up to 20 million fluid points running on 2048 cores [23]. The HemeLB team later detailed plans for an updated *in situ* visualization pipeline which would reduce the amount of data processed using level of detail methods [24]. The WaLBerla team integrated with VisIt for computational steering [25]. To our knowledge no LBM code has integrated a low overhead *in situ* visualization strategy which takes advantage of LBM's computational structure or has demonstrated performance at scales involving billions of fluid points.

2.5 Limitations of In situ Approaches

While enabling higher fidelity analysis than *post hoc* approaches, current *in situ* methods share important limitations at large scales. To achieve low overhead they require parameters such as camera placement, threshold values, or sampling rate to be determined *a priori* [26]. Preliminary data exploration informs these decisions; modifying parameters such as resolution or camera placement requires rerunning the simulation which is time consuming and expensive [27]. Data reduction is often performed on dense data where few time steps fit in memory preventing further reduction through temporal compression. Data size is kept tractable through spatial and temporal downsampling or aggressive thresholding. Data is irreversibly transformed and if transient or granular events are missed, supercomputing time is required to rerun the simulation. Lastly the data operated on by the simulation and *in situ* code is tightly coupled. While this enables high performance relative to *post hoc*, the simulation and analysis proceed in lockstep and thus detailed, computationally expensive analysis bottlenecks the simulation [11].

3 OVERVIEW OF HALO REPLAY METHOD FOR STENCIL CODES

We present an algorithm which permits full resolution data exploration over a rewindable window. It is designed for stencil codes that perform a halo exchange between time steps. Each process records halos received from neighbors allowing interior values for previous time steps to be recomputed, or replayed, locally on-demand. Here we focus on a Replay method designed for heterogeneous systems where the main computational kernel is executed on the GPU, though we have also adapted the method to work on homogeneous systems. Halo Replay reduces memory requirements by recording the data necessary for efficient reconstruction, similarly to how time-travel debugging tools and MPI reproducible replay enable full recovery of large application state through selective recording of nondeterministic, nonlocal, and costly operations [28]–[33].

3.1 Stencil Codes

Stencils are a common pattern found in scientific and engineering applications. Each time step involves a sequence of sweeps over the domain where points are updated by applying a function on neighbor elements inside the shape of the stencil. The arithmetic intensity of stencils tends to be low and performance memory bound [34]. Yet as points are updated independently, stencil codes are amenable to massive speedup through parallelization. When parallelized, processes are assigned different regions of the spatial domain and update their owned points in parallel. After a time step, synchronization with neighbor processes through a halo exchange is necessary.

As discussed in section 3.3 our method exploits this necessary transfer by storing halos in memory for on-demand reconstruction of interior points. Importantly, besides the halo exchange, advancing the neighborhood stencil a time step involves only local data. When halos are known in advance and accessible from persistent store no communication is needed. Given a history of halo points, the

interior region can be reconstructed independently from the rest of the simulation. Even if the full domain requires petabytes of memory distributed across millions of cores, a subdomain can be locally reconstructed on a single core.

3.2 Terminology

Here we clarify terms used in the description of the proposed Halo Replay method.

Main simulation task: the thread of an MPI process driving the principal simulation and coordinating MPI communication with neighbors to exchange halo values. The main task updates the simulation on the GPU. It may spawn child threads during Asynchronous memory transfers between the CPU and GPU or non-blocking MPI communication. After receiving halo values from its neighbors, the main task adds them to the Halo Buffer for the Replay task’s consumption.

Replay task: the thread of the same MPI process which reconstructs its own copy of the simulation data without any MPI communication using the Halo Buffer. It is able to advance a time step if the corresponding halo values exist in the Halo Buffer and the local state of the previous time step has been generated or received from the main simulation task. The Replay task updates the simulation on the CPU. It may spawn child threads for parallel execution of simulation kernels. All *in situ* routines are called on the Replay task.

Halo Buffer: Data structure storing the received halo values for a window of time steps. The Replay task has read-only access to the Halo Buffer.

3.3 Halo Replay Algorithm

The proposed algorithm, illustrated in Figure 1, may be separated into two main stages: Halo Record, where halo values from neighbor processes are received into the Halo Buffer, and Replay, where starting from a subdomain checkpoint the simulation is stepped forward loading values from the Halo Buffer instead of receiving them from neighbor processes.

Record Stage

Halo Record is performed on the main simulation task and involves only a minor modification to the original simulation loop code so that halos over a series of time steps may be stored. In the original code each process allocates a buffer sized to fit all received halo values for a single time step and at each time step the buffer is refilled with new received halo values. In the version with Record, the Halo Buffer allocates a 2-D array where each row holds the halo values for different time steps. At each time step a pointer into the Halo Buffer is incremented so in the next time step halos are received the next row in the array. The Record stage does not require any data transfers not already performed by the original simulation code but it does have additional memory requirements described in section 3.4.

Replay Stage

Here we describe a version in which Replay occurs in parallel with the main simulation however it is also possible to communicate or persist the halo values for *in transit* or *post hoc* Replay and analysis.

Each MPI process runs a main simulation task and a Replay task in parallel on separate OpenMP threads. Both threads have shared access to the Halo Buffer data structure. As shown in Figure 1 the only data shared between tasks is a buffer storing halo values and any checkpoints which the Replay task can use as a launch point.

The Replay task (Figure 1) advances its own simulation data using values from the local Halo Buffer which does not require MPI communication nor blocking synchronization with the main simulation task. Visualization routines are executed *in situ* from the Replay task using its version of the simulation data, allowing

the main simulation to proceed without being bottlenecked by the visualization.

While the main simulation task continues unblocked, the Replay task can never advance past the main simulation task. During the Record stage when the most recent halo values are received, variables are updated for the Halo Buffer’s head pointer and its corresponding time step so that the Replay task knows if halos are available for a certain time step. The variables for the head pointer and time step are updated atomically to protect against race conditions with the Replay task which has read-only access to these values.

In our observations the atomic updates add no overhead to the main simulation when executed every time step, however it is also possible to perform the atomic updates at less frequent intervals as the Replay task does not proceed in lockstep with the main simulation. The Replay task must not receive a false positive when checking if halos exist for time step t , but a false negative due to delaying the atomic update of the latest time step only causes the Replay task to wait and has no effect on correctness.

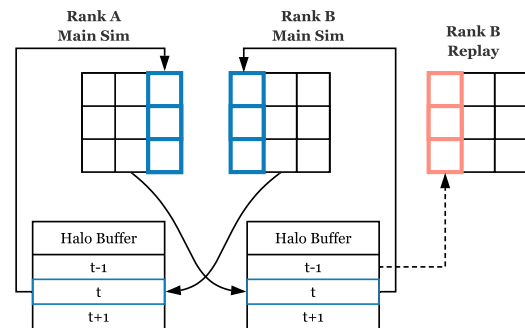


Figure 1: (Left) in Halo Record received halos are written into a row of the Halo Buffer for that time step. (Right) the Replay task advances its simulation independently, fetching halos for the next time step from the Halo Buffer instead of communicating with neighbor processes.

3.4 Theoretical Cost

3.4.1 Memory Requirements

Like other *in situ* schemes where analysis is executed by the CPU, the Halo Replay method requires CPU memory for a copy of the simulation data. The Halo Replay method needs additional memory for the Halo Buffer.

Each row in the Halo Buffer stores values received by a process at a different time step, leading to a memory footprint equal to the received data size for one time step multiplied by the rewind window size (i.e. row count). While not explored here, the Halo Buffer’s memory footprint may be reduced through out-of-core streaming and compression. For this study’s experiments the ratio of simulated points to communicated points is on the order of 100:1, and we use a rewind window of 100 time steps so that the Halo Buffer’s memory footprint roughly equals that of the simulation data. On Summit nodes the ratio of DRAM to the GPU’s High Bandwidth Memory (HBM) capacity is over 5x (Table 5.1). HBM and not DRAM limits our simulation size, so the memory requirements of a 100 time step window are always within DRAM capacity.

3.4.2 Main Simulation Run Time Overhead

Though requiring more memory, saving to the Halo Buffer does not impose additional work or data transfers on the main simulation task. Overhead only occurs due to resource contention from the Replay task. The expected low overhead is predicated on the main simulation spending little time in routines where resources are shared

with the Replay task. Reducing resource contention depends on process subdomains having a low surface-to-volume ratio, as the main simulation competes for CPU and interconnect bandwidth mainly for communication. Systems such as Summit with large per-node memory capacity encourage simulations with low surface-to-volume ratios so that less time is spent transferring data over the lower bandwidth node interconnects and these simulations would see especially low overhead from Replay.

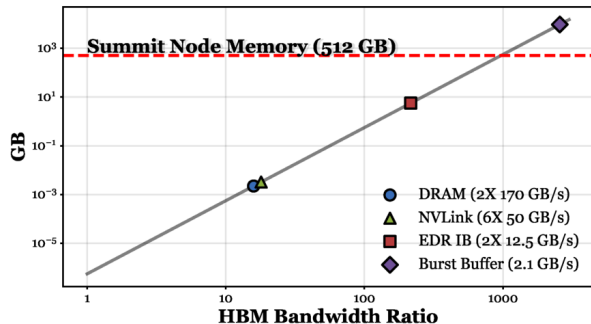


Figure 2: Per-node simulation size for halo generation rate to equal bandwidth of interfaces on Summit. When the generation rate exceeds the bandwidth of an interface, halo transfer becomes a bottleneck. Assumes a GPU-based algorithm with an arithmetic intensity of 1, cubic subdomain of N points and $6N^{\frac{2}{3}}$ halos.

In Figure 2 we analyze for a memory-bound algorithm how surface-to-volume ratio relates to halo generation rate and the ratios necessary to avoid bottlenecks when halos are generated faster than they can be transferred over interfaces such as NVLink and InfiniBand. In the figure, processes are assigned cubic subdomains of volume N and have an iteration time proportional to N divided by the GPU memory bandwidth. Each iteration generates $6N^{\frac{2}{3}}$ halos over the subdomain’s surface. We see that as long as the simulation size remains large enough to remain bound by GPU memory bandwidth, simulation updates can be overlapped with transfer of halos reducing impact from resource contention.

The overhead Ω of the main simulation due to Replay and analysis is a function of time the main simulation spends in GPU kernel updates, T_{GPU} relative to shared resource code, T_S and the overhead, ω , of the shared resource code from contention with Replay and analysis.

$$\Omega = \frac{\max(T_{GPU}, \omega T_S)}{\max(T_{GPU}, T_S)} \quad (1)$$

We illustrate this relationship in Figure 3 and show that even large overhead in shared resource code does not affect main simulation run time when the GPU has sufficient work.

4 APPROACH

We implemented Halo Replay in HARVEY and integrated with SENSEI to interface with ParaView Catalyst’s *in situ* visualization capabilities. SENSEI was chosen due to its flexibility in interfacing with multiple *in situ* infrastructures and to prepare a foundation for implementing additional *in situ* workflows. Catalyst was used for visualization due to its large library of visualization and analysis routines and for its convenient Python scripting abilities. Here we describe in further detail the components specific to our implementation of *in situ* visualization using Halo Replay.

4.1 Lattice Boltzmann Method

As our testbed stencil we use the LBM, a numerical scheme which approximates the Navier-Stokes equations. While LBM codes can

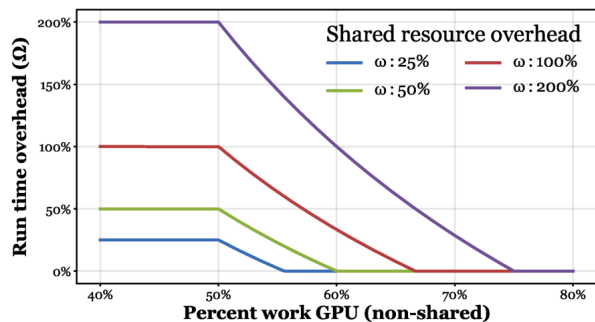


Figure 3: Simplified model of overhead to main simulation run time when sharing resources with a parallel task (Equation 1). The percent non-shared GPU work shown on the x-axis is $\frac{T_{GPU}}{T_{GPU} + T_S}$. Overlapping GPU work hides the run time cost of the shared-resource work. Overhead to shared-resource work, $\omega > 0$, results in run time overhead, $\Omega > 0$, only when $\omega T_S > T_{GPU}$. Increased surface-to-volume increases percentage of shared-resource work. Replay/analysis competition increases shared resource overhead.

exhibit near ideal scaling, they have an arithmetic intensity of $\mathcal{O}(1)$ and performance is highly dependent on memory bandwidth [35]. The LBM has applicability in a wide range of fields including aerospace, automotive, and manufacturing and its computational structure is representative of stencil codes and numerical methods with performance depending critically on instruction and data-level parallelism [16], [35]–[37].

HARVEY is an application that uses the LBM to enable personalized blood flow simulations with the aim of aiding clinical decision making [6]. HARVEY is specialized for simulations involving sparse, complex vascular geometries, domains where efficient memory access is especially challenging. For studying patient-specific hemodynamics, simulating at resolutions of at least $20 \mu\text{m}$ are typically necessary [7], and extreme-scale simulations will involve trillions of simulated points [5], [38]. In the D3Q19 formulation of the LBM used in HARVEY each simulated 3D lattice point has 19 double precision discrete velocity components and thus requires at least 19×8 bytes of memory (with further memory requirements for associated data structures).

4.2 Enabling *In situ* Visualization with SENSEI

We use the SENSEI framework for interfacing with ParaView Catalyst’s visualization API. SENSEI (or Scalable Environment for Scientific Explorations In Situ) is a platform that aims to improve code portability and reusability by decoupling simulations from the analysis routines. SENSEI has been demonstrated at similar scales we target in this work [11]. The design philosophy of SENSEI is *write once, use everywhere* [39]. This enables a simulation code to be instrumented once, and then have access to any of the visualization and analysis backends that are available through SENSEI. It provides a common instrumentation interface by leveraging VTK, which is widely used in visualization infrastructures such as VisIt/libsim and ParaView/Catalyst, as its common data model. Using this instrumentation interface, a *Data Adapter* is implemented on the simulation side to map simulation data to the VTK data model. This data conversion can often be facilitated through the use of memory pointers, without requiring expensive data copies. Once defined, SENSEI uses this *Data Adapter* to pass data on to an *Analysis Adapter*. Customized *Analysis Adapters* can be implemented to consume the VTK data through one of the supported analysis backend *in situ* infrastructures, such as Catalyst, libsim, the ADIOS IO framework [40], or Python.

5 EVALUATION METHODOLOGY

5.1 Hardware

All experiments presented run on Summit, which has a peak performance of 148.6 petaFLOPS and as of June 2019 is ranked first on the Top 500 list [41]. Summit is a 4,608 node machine with dual socket IBM POWER9 22-core 4-way simultaneous multithreading (SMT) processors and six NVIDIA Volta GPUs per node. Table 5.1 summarizes many of the system’s key characteristics. Particularly relevant are the large memory capacity and the memory bandwidths as those for memory bound stencil applications those are highly predictive of maximum simulation size and performance respectively.

Table 1: Summit System Characteristics. Summit’s architecture centers on *fat nodes*, each with huge memory capacity, compute capability. Node-local burst buffers provide scalable, high speed I/O. The design enables fast intranode data movement, and minimizes performance variability from system usage

CPU	2X POWER9
Cores/CPU	22
Memory	512 GB
Memory Bandwidth	170 GB/s/CPU
GPU	6X Volta V100
GPU Memory	96 GB HBM2
GPU Memory Bandwidth	900 GB/s/GPU
NVLink Bandwidth	50 GB/s/GPU
Burst Buffer Write	2.1 GB/s
Interconnect	2X 12.5 GB/s EDR

5.1.1 Configuration

Code was compiled using GCC 6.4 and CUDA 9.2. SENSEI version 2.1.1 and ParaView 5.5 were statically linked with the HARVEY executable.

All experiments were performed with 4 hardware threads per core (SMT4) and GPU Multi-Process Service allowing GPUs to be shared between processes on the same node. Unless otherwise stated experiments used 16 processes per node with all GPUs and CPU cores per node evenly assigned to processes. GPUs were oversubscribed since the version of HARVEY’s load balancer used in the experiments achieves better load balance when the total number of tasks is a power of 2, allowing for testing at larger scales before load imbalance strongly impacted results. The number of processes per node was chosen by optimizing only main simulation performance to avoid unfair representation of Replay performance and overhead.

5.1.2 Visualization on Summit

Here we provide details on the rendering techniques used on Summit to and their implications for visualization performance and overhead introduced to the main simulation. Hardware accelerated OpenGL has been recently supported in heterogenous clusters [42] though it was not fully supported on Summit at the time of this study. As an alternative we used the CPU based OSMesa interface and the multithreaded `llvmpipe` driver [43]. The use of CPU rendering would be expected to reduce resource contention with the main simulation running on the GPU. Further rendering performance is likely slower than with hardware acceleration though as described in section 5.2.1 the 1500×1500 resolution of the rendered images results in rendering being highly communication bound.

5.2 Measurements and Comparisons

We evaluate overhead and performance of the Halo Replay algorithm through direct comparison to versions of HARVEY without any *in situ* features and versions implementing the common Blocking and Asynchronous *in situ* schemes [21], [44].

Control: All *in situ* related features disabled.

Replay: Halos are recorded to buffer and Replay task performs simulation data reconstruction and visualization.

Blocking: Replay and Halo Buffer are disabled. At predefined time steps the main simulation task calls the visualization routine after transferring the simulation data from GPU to CPU memory. The simulation halts until the *in situ* routine completes.

Asynchronous: Similar to Blocking except the GPU simulation can proceed during visualization. Memory is allocated for a single extra copy of the simulation data, and if visualization has not completed before the GPU simulation reaches the next time step to visualize it waits for visualization to complete.

5.2.1 Geometries and Visualization Parameters

The input aorta geometry is described in Table 5.2.1. With 2.36% of the domain consisting of simulated points, the aorta is an example of the sparse geometries seen in hemodynamics simulations of the circulatory system. As the main artery in the human body, patient-specific simulations of blood flow in the aorta are of clinical interest [45] and aorta geometries are frequently used in testing CFD solvers [7], [46], [47]. Figure 4 shows an example generated frame of flow velocity in the simulated aorta. The visualization is comprised of two slices along the ascending and descending aorta and cover a large cross sectional area of the domain. The visualized property is velocity magnitude. Visualization parameters are chosen to encourage communication related resource contention between the main simulation and the visualization routines. The camera parameters and slices are selected so that the majority of MPI tasks own data in the output image set. The image resolution of 1500×1500 was chosen as compositing is communication bound at lower resolutions [48] and frequent communication offers more opportunities to compete with the main simulation.

Table 2: Resolutions and simulation sizes for aorta geometry used in experiments. Weak scaling runs with process counts not included here keep the average simulated points per process constant.

Processes	Res (μm)	Sim Points	Grid
32	63.5	119M	2010 \times 827 \times 3153
256	31.7	957M	4021 \times 1655 \times 6307
2048	15.9	7B	8043 \times 3311 \times 12614
16384	7.9	61B	16086 \times 6622 \times 25228

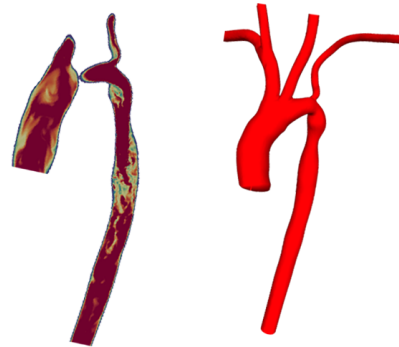


Figure 4: (Left) a frame of the velocity magnitude visualized in the experiments. It is comprised of slices covering a large cross sectional area of the domain. (Right) the aorta geometry used as input.

5.2.2 Overhead in Weak Scaling

We examine performance on up to 16384 MPI processes and 131072 OpenMP threads distributed across 1024 nodes consisting of 6144 V100s and 43008 Power9 SMT4 cores. Table 5.2.1 shows the input sizes for a subset of the scales, with the number of simulation points per process constant for all process counts. Input sizes for scales not shown also have the same number of points per process. To assess the performance impact of Replay we compare the average main simulation iteration time over 10000 time steps to the Control version of HARVEY with all *in situ* features disabled. The Replay implementation in parallel renders 25 frames over a 100 time step window. The number of main simulation iterations and the size of the Replay window were chosen so that the main simulation always reached its time step target before the Replay task to ensure that the main simulation incurs overhead the entirety of the run. Additionally the Replay parameters were set so that the Replay task rendered its final image shortly after the main simulation reached its time step target to capture overhead from the differing resource demands of Replay simulation and visualization.

To separate the performance impact of Replay and visualization from storing to the Halo Buffer, we include a comparison to a version of HARVEY which writes to the Halo Buffer without performing Replay or analysis.

5.2.3 Overhead in Strong Scaling

Summit's *fat nodes* encourage using a smaller number of memory intensive processes. The design is intended to limit scaling bottlenecks from inter-node communication and route more communication through high bandwidth intra-node communication methods [49]. This system design is well suited for Replay through Halo Buffer persistence as it allows nodes to be assigned subdomains with small surface area to volume ratios. However the system design makes strong scaling experiments only relevant over a small range of nodes if one would like to make realistic use of available resources. Despite less efficient resource use as node count increases, strong scaling experiments offer insights for the Halo Replay algorithm as we can observe overhead from increase surface area to volume ratio using only a few system nodes.

To measure overhead from Replay we compare the average main simulation iteration time over 10000 time steps to the Control. The Replay version renders 25 frames over a 100 time step window. Strong scaling is performed from 16 to 128 processes using 1 to 8 nodes and 16 processes per node. All strong scaling experiments used the aorta geometry at 63 μm (Table 5.2.1).

5.2.4 Replay Thread Management

We analyze the effect of allocating a different number of threads to Replay on both main simulation and Replay performance. As overhead to the main simulation may be minimized simply by throttling the Replay task, the goal was to find a configuration which optimizes both objectives of main simulation iteration time and Replay iteration and visualization time. Each Summit node has 168 SMT cores and permits highly flexible assignment of threads to workloads. Thus we evaluate a limited subset of configurations, partly due to application-specific preference for power of two process counts.

Tests are performed on 64 MPI processes distributed across 4 nodes and up to 8 OpenMP threads per process, with one thread assigned to the main simulation and up to 7 assigned to Replay. As with the weak scaling experiment the average iteration time of the main simulation is measured over 10000 time steps and the Replay task generates 25 frames at a 1500×1500 resolution over a window of 100 time steps. These experiments use the aorta geometry at 63 μm resolution (Table 5.2.1).

5.2.5 Comparison of Main Simulation Overhead With *In situ* Schemes

To compare the overhead to the main simulation introduced by Halo Replay to the alternative *in situ* visualization schemes described in section 5.2 we perform the same strong scaling tests with all schemes measuring average main simulation iteration time over 10000 time steps. The Replay, Asynchronous, and Blocking implementations all generate images for the same time steps and run the main simulation for the same number of total time steps. Each renders 25 images over a 100 time step window. The Asynchronous and Blocking implementations continue the simulation after generating all images in the window until reaching time step 10000.

5.2.6 Comparison of Halo Replay and Asynchronous at Different Visualization Frequencies

The strengths of different *in situ* approaches depend heavily on the needs of the application and therefore it is important to understand the tradeoffs of each method and compare performance along different dimensions of interest. We compare the Replay and Asynchronous methods according to two metrics which may be of primary importance in different contexts.

- Overhead to the main simulation (or time to reach main simulation target time step)
- The time to complete rendering of all frames over the visualized time step window

We see in general the methods provides advantages along one of these dimensions over the other and vary the time step intervals between visualized frames to explore how the visualization frequency influences their respective strengths.

5.2.7 Cost of Visualization Stages

We measure the time spent in stages of the code necessary for visualizing a frame to understand their relative cost at different scales. We use the Blocking implementation as the performance of the Replay and Asynchronous versions may be influenced by the frequency of visualization and code executed in parallel which competes for resources. The stages of visualization are separated into:

- Transfer of simulation data from host to device,
- Calculation of visualized properties from the simulation data,
- Passing buffers holding visualized properties to SENSEI for visualization.

6 EXPERIMENTAL RESULTS

6.1 Overhead in Weak Scaling

Figure 5 shows the measured overhead in the weak scaling experiments. Writing to the Halo Buffer without Replay (Halo Record only) introduces no overhead to the main simulation. At all scales up to 8192 process the Halo Record only implementation incurs between <1 to 2% overhead with no clear correlation with increasing scale. At the highest process count of 16384 an overhead of 4% was observed. However at the highest process counts a variance in iteration time of up to 4% was observed between the control trials repeated on different days.

The low overhead of writing to the Halo Buffer is unsurprising as the Halo Record only implementation does not require additional memory transfers, only more memory to be allocated for storing a time sequence of halos. To ensure high device host bandwidth the Halo Buffer must be allocated in pinned memory. This is critical to performance as after halos are received through MPI communication, they are transferred to the GPU for updating the simulation and this transfer occurs at reduced bandwidth if using pageable host memory.

Overhead of Replay relative to Control remains between 0.5% and 2% up to 2048 processes. At 8192 and 16384 processes it grows to 9% and 17% respectively. Overhead is low at all scales relative to the time the Replay tasks spend in the visualization routines. For all

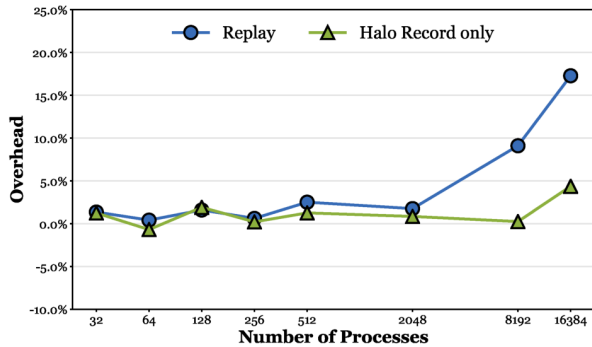


Figure 5: Weak scaling of overhead to main simulation loop time. **Replay** renders 25 frames. **Halo Record** only writes received halos to a buffer holding 100 time steps but disables Replay and rendering.

scales the Replay task spends 28% to 43% of the total simulation run time in the visualization routines that generate the 25 images. This would be minimum overhead to the main simulation if the *in situ* visualization were performed with the Blocking scheme. Due to the Blocking method’s additional costs of transferring all simulation data from the GPU to CPU for every rendered time step, the actual overhead of Blocking for the weak scaling experiments reaches 70%. Even greater overhead benefits relative to Blocking would be seen if higher resolution images were generated, as the time to render each frame would increase.

Visualization communication costs increase at at large scales [50] and thus we would expect this to lead to greater contention between the main simulation and visualization. Figure 6 displays the run time cost of *in situ* visualization on the Replay task in the weak scaling experiments. The visualization times are consistent with other reports [48], [50] of the scaling of the IceT compositing library used by Catalyst. Increased communication requirements of image compositing and collection is considered to be a major cause of longer render times at higher process counts [50]. While the visualization is performed on the Replay task and does not block the main simulation, the increased communication cost of visualization is likely to be a source of the overhead seen at larger scales. For Blocking *in situ* visualization the increased cost of communication directly contributes to increased run time by delaying the start of the next iteration.

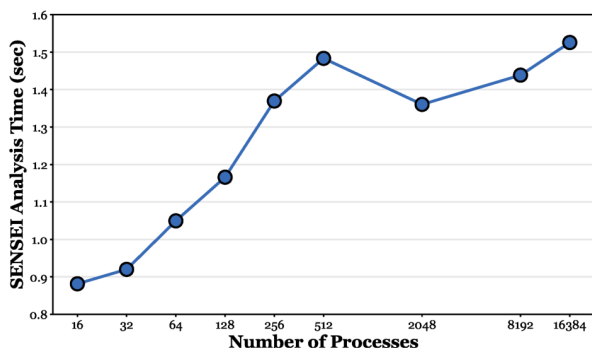


Figure 6: The mean visualization time on Replay tasks. We do not profile the rendering code so we use time spent in SENSEI/Catalyst routines as visualization cost. Growing visualization costs lead to resource contention with the main simulation and the overhead at high process counts seen in Figure 5.

6.2 Replay Thread Management

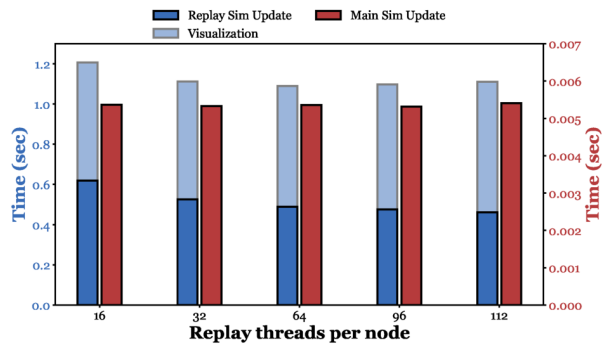


Figure 7: Effect on main simulation and *in situ* visualization performance of increasing the number of threads per Replay task.

We observe in Figure 7 that the main simulation performance shows little variability when different number of threads are allocated for Replay and visualization. There is a 1% increase in main simulation iteration time when the maximum tested number of 112 SMT cores per node are assigned to Replay threads. Greater variability is seen in the performance of the Replay tasks. As the Replay task uses an OpenMP kernel, the simulation update speed is slower when fewer threads are assigned to Replay, though performance plateaus at 64 threads per node (4 per process). The performance of the Replay task can be understood in the context of the sustainable memory bandwidth on Summit. The Replay task is bound by available memory bandwidth which saturates before utilizing all cores on the CPU and allocating additional threads to Replay once this bandwidth limit is reached will not improve performance. The STREAM benchmark [51] is the standard for measuring sustainable memory bandwidth. In our measurements (Table 6.2) memory bandwidth of the POWER9 saturates when using just 16 threads out of a total of 84 SMT units. For a node with two POWER9s we would therefore expect a memory bound algorithm such as the LBM to see little benefit using over 32 threads per node.

While there was little difference between configurations there are subtleties that could influence performance in expanded tests. For example, the main GPU based simulation performs several asynchronous operations such as non-blocking MPI communication and CUDA device host transfers. These operations have threading behavior that are not explicitly controlled by the program and their performance may be affected by how they are assigned to SMT units.

Table 3: Measured memory bandwidth (GB/s) of the POWER9 and Tesla V100 on a Summit node. CPU tests run on a single POWER9, SMT level 4 and variable OpenMP threads using STREAM [51] compiled with GCC. GPU HBM bandwidth measured with Babel-Stream [52].

SMT	4	8	16	32	84	GPU
Copy	53.5	87.3	111.9	108.2	106.8	806.9
Triad	68.1	113.1	132.9	127.1	123.2	846.9

6.3 Overhead in Strong Scaling

A similar trend in overhead is observed in Figure 8 as in the weak scaling experiments. Writing to the Halo Buffer introduces little to no overhead as we see overhead of the Halo only implementation remain below 1% and does not increase with scale. Overhead from

Replay and visualization remains below 2% up to 64 processes and increases to 6% at 128 processes. At these scales any overhead in the Replay implementation is unlikely to come from visualization as the communication requirements are small for such process counts. All processes have an increased surface area to volume ratio at larger scales and thus the main simulation spends more time in code limited by CPU memory bandwidth that is shared with the Replay task.

The negligible overhead of the Halo only implementation even when surface-to-volume ratio is large is promising since in certain use cases the user may want to Halos to allow for later Replay but is not currently interested in Replaying the simulation on all processes. The user can later explore the data of interest without incurring overhead from simultaneous Replay competing for resources.

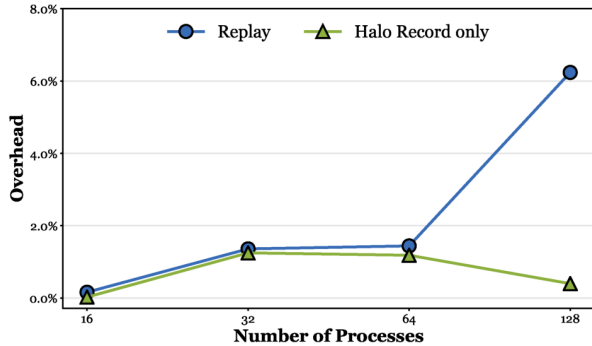


Figure 8: Strong scaling $63\mu\text{m}$ (120 million fluid points). 16 MPI processes per node, sharing 6 GPUs. Averages are computed over 10,000 iterations. At larger scales a greater portion of run time is spent in data transfer, competing with Replay for memory bandwidth.

6.3.1 Comparison of Main Simulation Overhead With *In situ* Schemes

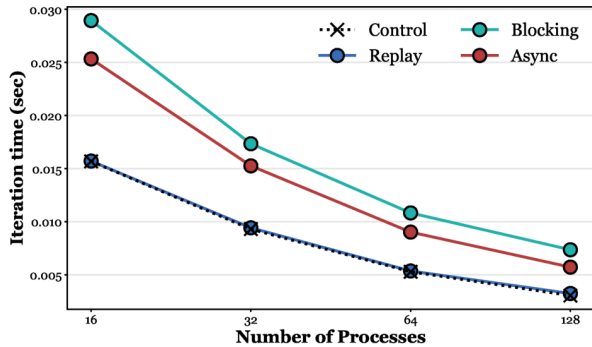


Figure 9: Main simulation performance comparison to Blocking and Asynchronous schemes for *in situ* visualization. Strong scaling $63\mu\text{m}$ (120 million fluid points). Averages are computed over 10,000 iterations.

The results in Figure 9 show the average main simulation iteration time over 10000 time steps for the different *in situ* schemes. At the lower process counts in this experiment the low communication cost of image compositing is a less important source of overhead than in the weak scaling experiments. In strong scaling experiments the surface-to-volume ratio increases with scale, and here we focus on how this property influences performance of the different *in situ* schemes. The Asynchronous version outperforms Blocking while both deviate significantly from the Control. Replay has significantly

smaller overhead as compared to the Control, ranging from less than 1% at the smallest process count to 6% at the largest. Though both the Asynchronous and Replay schemes do not block the main simulation to perform rendering, the larger overhead of the Asynchronous version is due to the blocking transfer of the full simulation data set from the GPU to the CPU. The Halo Replay version in contrast only needs to transfer the smaller halo dataset between the host and device, a process also required by the other versions, before allowing the simulation to proceed.

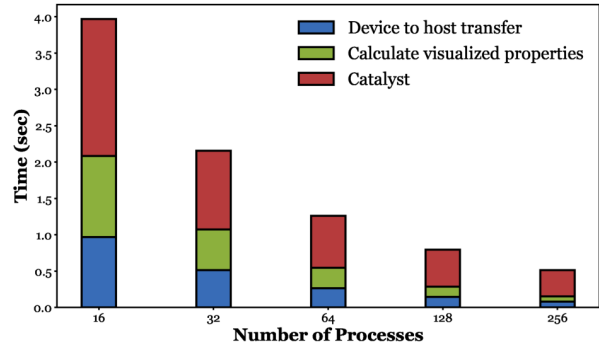


Figure 10: Strong scaling of time spent in stages required during visualized time steps for the Blocking implementation.

6.3.2 Cost of Visualization Stages

In Figure 10 we see strong scaling reduces the time spent in each stage with less speedup in rendering than the other phases. The device to host transfer and calculating visualized properties are process local operations while rendering requires communication which prevents it from scaling as efficiently. In these experiments the rendered frames are of size 1500×1500 . The relative time spent in the visualization stage would be larger for higher resolution images. For these tests we did not separately measure time spent in process local rendering and compositing. Accelerated rendering using OpenGL instead of Mesa, as recently made available on Summit, would dramatically decrease the cost of process local rendering, leading to different relative times in the stages displayed here.

6.3.3 Visualization Frequency

We compare performance of Replay to the Asynchronous implementation in terms of overhead to the main simulation and rate of visualization as applications may value these metrics differently. The more frequent visualization, the larger the advantage of Replay over Asynchronous in terms of main simulation speed. This is due to increased memory traffic as the device must transfer all its simulated points at a visualized time step. For Replay the main simulation iteration time is relatively invariant to visualization frequency. However performance degradation is seen for 64 and 128 processes when visualization occurs every frame as each process has fewer points to update and both the main simulation and visualization threads frequently compete for communication bandwidth.

When instead looking at the rate of visualization in Figure 11, the Asynchronous implementation outperforms Replay. The performance gap is largest when visualization is less frequent as the Replay implementation must compute all intermediate time steps between frames on the CPU. Only in the case when visualization occurs every frame does Replay narrowly outperform Asynchronous. The similar performance of Replay and Asynchronous when visualization occurs every frame is explained by the memory and interconnect bandwidths of a Summit node. Each socket has a combined bandwidth of 150 GB/s between the GPUs and the CPU, which is similar

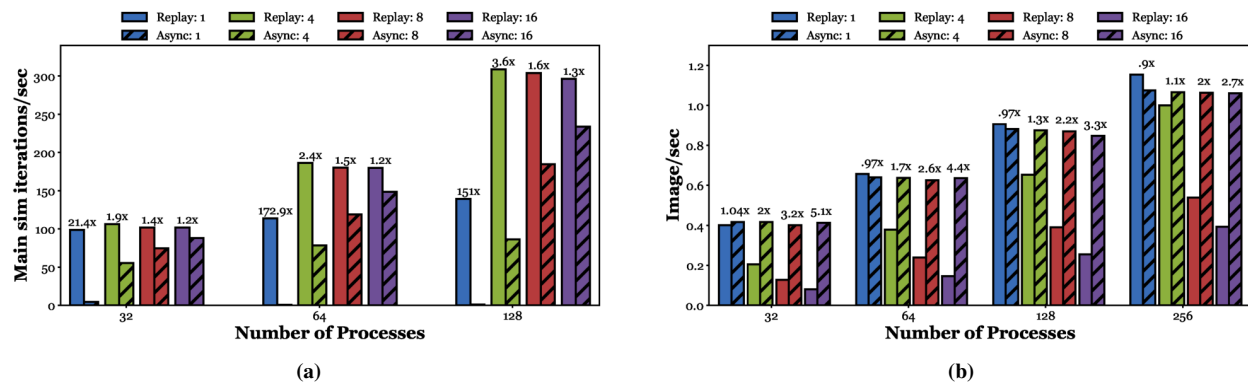


Figure 11: The tradeoff between visualization time step frequency on main simulation iteration time and rate of image generation. Performance is evaluated with intervals between visualization of 1, 4, 8 and 16 time steps. In each case 25 frames are generated and the visualization window is 25 multiplied by the time step interval. **(a)** The main simulation iteration rate over the duration it takes Halo Replay to visualize 25 frames. The numbers above the bars show the speedup of Replay compared to Asynchronous main simulation iteration time. Replay introduces less overhead to the main simulation, especially at higher frequencies of visualization. **(b)** The image generation rate over the visualized window. The numbers above the bars show the speedup of Asynchronous compared to Replay for image generation rate. For larger time step intervals between frames the speed advantage of Asynchronous over Replay, which must reconstruct intermediate time steps on the CPU.

to the CPU’s memory bandwidth of 170 GB/s (Table 5.1). Given the arithmetic intensity of LBM is close to 1, the rate at which all point values can be transferred over NVLink is similar to the rate the CPU can step forward the simulation.

7 DISCUSSION

7.1 Comparison to Other Methods

Halo Replay is an attractive option when low overhead to the main simulation is a more important concern than the analysis run time. There is a large variety of *in situ* workloads with different computational costs, resource requirements, and effects on simulation performance. Complex routines with a long or indeterminate execution time may result in intractable simulation run times and inefficient utilization of system resources when using traditional *in situ* schemes, as we saw that even the Asynchronous *in situ* scheme quickly becomes a bottleneck to the simulation when analysis is frequent and relatively costly. In such cases frequent, high fidelity analysis requires the simulation to stall after only a few time steps before the dense data enqueued for analysis overwhelms memory capacities. Halo Replay enables such analysis to execute independently of the simulation with a memory footprint proportional to surface area instead of volume.

For codes not fully utilizing the CPUs on heterogeneous systems, simultaneous Halo Replay can be an efficient use of compute resources. However when run time of analysis over a predefined interval is of prime importance, the Asynchronous scheme should be chosen over Replay. The total cost to visualize a frame with Replay grows with the number of time steps between frames while the Asynchronous scheme has a constant cost. However, unlike the experiments in this study, simultaneous Replay does not need to be running at all times. It can be used in conjunction with other *in situ* paradigms and enabled to rewind at full resolution for more detailed analysis. Halo Record’s negligible run time overhead offers this flexibility at no computational cost.

8 SUMMARY AND CONCLUSIONS

We have presented a method for *in situ* analysis of stencil codes that has low run time overhead and enables more flexible data exploration through memory efficient full resolution replay of previous time steps. In comparison to other methods it has significantly reduced run time overhead to the main simulation. It can be used

in combination with other schemes and with low overhead enables a full resolution rewind window to mitigate the risk of missing transient events of interest.

Potential avenues for optimization that would improve the usefulness of the proposed method include using reduced precision on the Replay task to speed reconstruction or lower memory bandwidth requirements. Larger rewind windows could be obtained through compression of halos and streaming to and from the burst buffer.

Further work will be required for optimizing the algorithm for future exascale systems based on homogeneous architectures. Homogeneous systems will require different techniques for reducing contention between simulation and analysis such as pipelining or staggering tasks with similar resource usage. The presented algorithm sets the stage for large-scale visualization and analysis at temporal and spatial fidelity that would carry prohibitive overhead using the common *in situ* schemes. It aims to be a valuable tool in the diverse set of *in situ* workflows necessary for the exascale era.

ACKNOWLEDGMENTS

We are thankful to David E. DeMarle of Kitware for his support and guidance. We would like to thank Daniel Puleri, Madhurima Vardhan, and Gregory Herschlag for their thoughtful discussions and feedback. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This work used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This work was performed under the auspices of the U.S. Department of Energy by LLNL under Contract DE-AC52-07NA27344. Research reported in this publication was supported by the Office of the Director, National Institutes of Health under Award Number DP5OD019876.

REFERENCES

- [1] A. P. Randles, V. Kale, J. Hammond, W. Gropp, and E. Kaxiras, “Performance analysis of the lattice boltzmann model beyond navier-stokes”, in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, 2013.

- [2] P. C. Wong, H.-W. Shen, C. R. Johnson, C. Chen, and R. B. Ross, "The top 10 challenges in extreme-scale visual analytics", *IEEE computer graphics and applications*, vol. 32, no. 4, 2012.
- [3] A. C. Bauer, B. Geveci, and W. Schroeder, *The paraview catalyst user's guide v2. 0. kitware*, 2015.
- [4] A. Peters, S. Melchionna, E. Kaxiras, J. Lätt, J. Sircar, M. Bernaschi, M. Bison, and S. Succi, "Multiscale simulation of cardiovascular flows on the ibm bluegene/p: Full heart-circulation system at red-blood cell resolution", in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2010.
- [5] C. Godenschwager, F. Schornbaum, M. Bauer, H. Köstler, and U. Rüde, "A framework for hybrid parallel flow simulations with a trillion cells in complex geometries", *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC '13*, no. November, 2013.
- [6] J. Gounley, M. Vardhan, and A. Randles, "A Computational Framework to Assess the Influence of Changes in Vascular Geometry on Blood Flow", 2017.
- [7] A. Randles, E. W. Draeger, T. Oettel, L. Krauss, and J. A. Gunnels, "Massively parallel models of the human circulatory system", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015.
- [8] K. Moreland, "The tensions of in situ visualization", *IEEE computer graphics and applications*, vol. 36, no. 2, 2016.
- [9] R. A. Oldfield, K. Moreland, N. Fabian, and D. Rogers, "Evaluation of methods to integrate analysis into a large-scale shock physics code", 2014.
- [10] 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 *International Conference on High Performance Computing*, Springer, 2019.
- [11] U. Ayachit, A. Bauer, E. P. Duque, G. Eisenhauer, N. Ferrer, J. Gu, K. E. Jansen, B. Loring, Z. Lukić, S. Menon, et al., "Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2016.
- [12] S. Li, N. Marsaglia, C. Garth, J. Woodring, J. Clyne, and H. Childs, "Data Reduction Techniques for Simulation, Visualization and Data Analysis", *Computer Graphics Forum*, vol. 37, no. 6, 2018.
- [13] O. Pearce, "Exploring utilization options of heterogeneous architectures for multi-physics simulations", *Parallel Computing*, vol. 87, 2019.
- [14] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "Stella: A domain-specific tool for structured grid methods in weather and climate models", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2015.
- [15] G. Roth, G. Roth, J. Mellor-crummey, J. Mellor-crummey, K. Kennedy, K. Kennedy, R. G. Brickner, and R. G. Brickner, "Compiling stencils in high performance fortran", in *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, 1997.
- [16] H. Aono, A. Gupta, D. Qi, and W. Shyy, "The lattice boltzmann method for flapping wing aerodynamics", in *40th Fluid Dynamics Conference and Exhibit*, 2010.
- [17] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data", in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, Oct. 2012.
- [18] S. Hamilton, R. Burns, C. Meneveau, P. Johnson, P. Lindstrom, J. Patchett, and A. S. Szalay, "Extreme event analysis in next generation simulation architectures", in *International Supercomputing Conference*, Springer, 2017.
- [19] P. Lindstrom, "Fixed-rate compressed floating-point arrays", *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, 2014.
- [20] P. Malakar, C. Knight, T. Munson, V. Vishwanath, and M. E. Papka, "Scalable In situ Analysis of Molecular Dynamics Simulations", *Proceedings of the Third Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV 2017)*, no. Md, 2017.
- [21] 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", *Computer Graphics Forum*, vol. 35, no. 3, 2016.
- [22] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data", in *European Conference on Parallel Processing*, Springer, 2011.
- [23] M. D. Mazzeo, S. Manos, and P. V. Coveney, "In situ ray tracing and computational steering for interactive blood flow simulation", *Computer Physics Communications*, vol. 181, no. 2, 2010.
- [24] F. Chen, M. Flatken, A. Basermann, A. Gerndt, J. Hetherington, T. Kruger, G. Matura, and R. W. Nash, "Enabling in situ pre- and post-processing for exascale hemodynamic simulations - A co-design study with the sparse geometry lattice-boltzmann code HemeLB", *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, 2012.
- [25] A. Daubler, "Interactive Visualization and Simulation of Fluids", vol. 10, 2014.
- [26] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows", *International Journal of High Performance Computing Applications*, vol. 32, no. 1, 2018.
- [27] 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", *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2015-Janua, no. January, 2014.

- [28] R. O’Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, “Engineering Record And Replay For Deployability: Extended Technical Report”, 2017. arXiv: 1705.05937.
- [29] C. Gottbrath and T. Technologies, “Reverse Debugging with the TotalView Debugger”,
- [30] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz, “Clock Delta Compression for Scalable Order-Replay of Non-Deterministic Parallel Applications”, 2015.
- [31] D. Geels, G. Altekari, S. Shenker, and I. Stoica, “Replay debugging for distributed applications”, *Proceedings of the annual conference on USENIX’06 Annual Technical Conference*, 2006.
- [32] J. Zhai, W. Chen, W. Zheng, and K. Li, “Performance Prediction for Large-Scale Parallel Applications Using Representative Replay”, *IEEE Transactions on Computers*, vol. 65, no. 7, 2016.
- [33] M. Noeth, F. Mueller, M. Schulz, and B. R. De Supinski, “Scalable compression and replay of communication traces in massively parallel environments”, *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM*, 2007.
- [34] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures”, *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008*, no. November, 2008.
- [35] S. Williams, L. Oliker, J. Carter, and J. Shalf, “Extracting ultra-scale Lattice Boltzmann performance via hierarchical and distributed auto-tuning”, 2011.
- [36] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde, “WaLBerla: HPC software design for computational engineering simulations”, *Journal of Computational Science*, vol. 2, no. 2, 2011.
- [37] R. Kotapati, A. Keating, S. Kandasamy, B. Duncan, R. Shock, and H. Chen, “The lattice-boltzmann-vles method for automotive fluid dynamics simulation, a review”, SAE Technical Paper, Tech. Rep., 2009.
- [38] A. Randles, E. W. Draeger, T. Opielstrup, L. Krauss, and J. A. Gunnels, “Massively parallel models of the human circulatory system”, 2015.
- [39] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, “The SENSEI Generic in Situ Interface”, in *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, ser. ISAV’16, Salt Lake City, Utah: IEEE Press, Nov. 2016.
- [40] 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”, *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, 2014.
- [41] *Top 500*, <http://www.top500.org>, Accessed: 2018-10-12.
- [42] J. E. Stone, P. Messmer, R. Sisneros, and K. Schulten, “High performance molecular visualization: In-situ and parallel rendering with egl”, in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016.
- [43] mesa3d, *Osmesa llvmpipe*, <https://www.mesa3d.org/llvmpipe.html>, 2019 (accessed April 9, 2019).
- [44] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, et al., “Combining in-situ and in-transit processing to enable extreme-scale scientific analysis”, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, 2012.
- [45] N. Stergiopoulos, D. Young, and T. Rogge, “Computer simulation of arterial flow with applications to arterial and aortic stenoses”, *Journal of biomechanics*, vol. 25, no. 12, 1992.
- [46] T. Tomczak and R. G. Szafran, “A new GPU implementation for lattice-Boltzmann simulations on sparse geometries”, *Computer Physics Communications*, no. November, 2018. arXiv: arXiv:1611.02445v2.
- [47] L. Axner, J. Bernsdorf, T. Zeiser, P. Lammers, J. Linxweiler, and A. G. Hoekstra, “Performance evaluation of a parallel sparse lattice Boltzmann solver”, *Journal of Computational Physics*, vol. 227, no. 10, 2008.
- [48] A. V. 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 Transactions on Visualization and Computer Graphics*, vol. 23, no. 6, 2017.
- [49] S. S. Vazhkudai, B. R. D. Supinski, A. S. Bland, A. Geist, J. Sexton, and J. Kahle, “The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems”, 2018.
- [50] K. Moreland, W. Kendall, T. Peterka, and J. Huang, “An image compositing solution at scale”, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’11*, 2011.
- [51] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers”, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.
- [52] T. Deakin and S. McIntosh-Smith, “GPU-STREAM: Benchmarking the achievable memory bandwidth of Graphics Processing Units”, *Supercomputing*, 2015.