A Hybrid In Situ Approach for Cost Efficient Image Database Generation

Valentin Bruder, Matthew Larsen, Thomas Ertl, Hank Childs, and Steffen Frey

Abstract—The visualization of results while the simulation is running is increasingly common in extreme scale computing environments. We present a novel approach for in situ generation of image databases to achieve cost savings on supercomputers. Our approach, a hybrid between traditional inline and in transit techniques, dynamically distributes visualization tasks between simulation nodes and visualization nodes, using probing as a basis to estimate rendering cost. Our hybrid design differs from previous works in that it creates opportunities to minimize idle time from four fundamental types of inefficiency: *variability*, limited *scalability*, *overhead*, and *rightsizing*. We demonstrate our results by comparing our method against both inline and in transit methods for a variety of configurations, including two simulation codes and a scaling study that goes above 19K cores. Our findings show that our approach is superior in many configurations. As in situ visualization becomes increasingly ubiquitous, we believe our technique could lead to significant amounts of reclaimed cycles on supercomputers.

Index Terms—Visualization, High performance computing, In situ

1 Introduction

Scientific visualization is a key approach for understanding the complicated data sets produced by computational simulations on supercomputers. These simulations produce massive data sets, with meshes containing billions or even trillions of cells per time step, and so the visualization process is typically parallelized to complete on interactive time scales. Further, this visualization process often occurs on the same supercomputer that performed the simulation, obviating the need to relocate simulation data [1]–[4].

Leading-edge supercomputers are quite expensive, meriting significant investigation into optimizing their usage. Several new supercomputers are built annually with hardware procurements costs upwards of $200M, and their true costs rising higher over time, including energy costs, staffing, and upkeep. Each job running on a supercomputer shares these costs. As a result, one important way to optimize a supercomputer's usage is to optimize individual jobs, i.e., having a job complete using fewer node hours. Such a speedup frees up node-hours for running other jobs, enabling the supercomputer to perform more calculations in its finite life than it could otherwise. If a speedup can be aggregated over all jobs, then the result can be profound, potentially creating millions of dollars of extra node-hours for additional computations.

This work considers the topic of optimizing jobs on supercomputers in the context of in situ visualization, i.e., visualizing data as it is generated. Traditionally, visualization on supercomputers has used post hoc processing, i.e., simulations save their data to disk and dedicated visualization programs later read that data. However, the post hoc paradigm is increasingly ineffective on supercomputers, as the ability to generate data on each new generation of supercomputer is increasing much faster than the ability to store and load data. In turn, I/O load times for visualization are becoming unacceptably large, as are I/O times for simulations performing frequent storage. In situ visualization avoids this issue, since visualization can occur without utilizing the file system. As a result, in situ processing is increasingly being adopted on leading-edge supercomputers, and has the potential to become the dominant paradigm in the future [5]–[7].

Despite the growing preference for in situ visualization, there is still much room for improvement in terms of cost and efficiency. Regarding cost, in situ visualization will, despite saving on I/O, still require significant computational resources—in situ routines sometimes use 10% or more of the simulation's resources. Of course, the exact proportion of time between visualization and simulation varies, based on the nature of the simulation and the data it produces, the type of visualization algorithm, the frequency visualization occurs (i.e., every cycle or less often), and

- *V. Bruder is with Daimler Truck AG, Germany.*
- *M. Larsen is with Luminary Cloud, Inc., CA, USA.*
- *T. Ertl is with with the University of Stuttgart, Germany.*
- *H. Childs is with the University of Oregon, OR, USA.*
- *S. Frey is with the University of Groningen, The Netherlands. E-mail: s.d.frey@rug.nl*

Manuscript received May 26, 2021; revised Feb. 08, 2022.

other factors. The second observation, that in situ visualization can be inefficient, is discussed in Section 3.

With this work, we avoid inefficiencies via a new in situ visualization approach that is a hybrid of traditional in situ approaches. Our method specifically considers an important technique for in situ visualization: the generation of image databases of volume renderings in the style of the Cinema project [8]. That said, our technique is applicable to any in situ visualization setting that can be split into many small tasks. Our findings indicate that we can regularly save on the order of 7.5% of the combined simulation and visualization time. We believe this speedup is very impactful for this setting—the motivating research assumption behind our work is that in situ visualization will become ubiquitous on supercomputers, and that optimizing its performance can potentially lead to millions of dollars of reclaimed cycles.

2 Related Work

This section is organized into four main areas of in situ work: instantiations, hybrid approaches, approaches focusing on cost savings, and elastic approaches (that adapt resource usage over time).

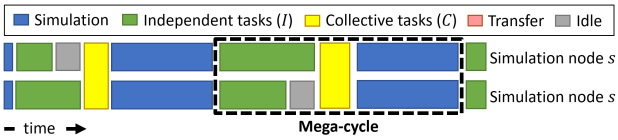
In situ instantiations: There are many possible instances of in situ processing, varying over division of resources, access to data, and other factors [9]. That said, two instances are used most commonly. In the first instance, sometimes referred to as inline in situ, the simulation code and visualization routines run on the same compute nodes, accessing the same memory and alternating usage of a node's cores. In this setting, visualization routines are typically integrated into the simulation via a library, and the simulation code invokes this library whenever visualization is required, effectively giving up control of the compute resources until the visualization routine returns from its function call. Popular products that use this form include Catalyst [10], Ascent [11], and LibSim [12]. In the second instance, sometimes referred to as in transit in situ, the simulation code and visualization routines use distinct resources, which we refer to as "simulation nodes" and "visualization nodes." In this setting, the simulation code typically sends its data to the visualization nodes via network communication, and the visualization nodes will keep its own separate copy of the simulation data. Popular products that use this form include Damaris [13], SENSEI [14], and ADIOS [15]. Our work is different than these as we consider a hybrid between inline and in transit. We compare to both of these approaches in our evaluation.

Hybrid in situ approaches: To date, there have been few true hybrid in situ approaches that blend between inline and in transit. Notably, Bennett et al. [16] used a hybrid in situ approach for S3D combustion simulations, using inline computations to reduce data such that modest in transit resources could be used to complete analysis tasks. In another notable work, Zheng et al. [17] introduced "PreDatA," where compute nodes could do "local processing" before sending data to in transit nodes, although most of the calculations took place on in transit nodes.

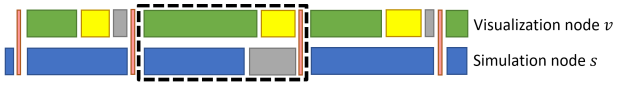
In situ approaches focusing on cost savings and improving performance: Several works have focused on addressing inherent inefficiencies with in situ processing. With Flexpath [18], the authors focus on saving transfer costs by reducing data movements or optimizing the data placement based on network topology and other performance influencing factors. Damaris [19] considered the issue of variability, while Kress et al. considered using in transit to reduce scalability in two separate studies [20], [21]. Our work can be viewed as a continuation of the Kress work, but with adding support for rightsizing, and, to a lesser extent, variability and overhead.

There are also several works on assessing resource usage of inline in situ and in transit analysis and visualization tasks, including quantitative formulations [22], [23]. Friesen et al. [24] discuss in situ experiments for the two instances with the Nyx simulation code and two in situ analysis suites while mainly considering overall execution time performance. Other works focus on general workflow optimization and orchestration [25], [26] or sub-sampling of simulation data to be processed on a local machine [27]. While our approach explicitly optimizes for efficient usage of node seconds, other works instead primarily focus on minimizing the time to solution (e.g., by dynamically allocating additional resources). Further, they use either in transit or inline processing, but not both in a hybrid fashion. In the case of in transit, rightsizing is not considered although several works acknowledge the problem.

Elastic in situ: There have been more works exploring elastic in situ [28], i.e., resource adaptation over the execution between simulation and visualization. Goldrush [29] identified when simulation resources were idle and used them to perform analysis tasks, and Landrush [30] extended this idea to use idle cycles on GPUs. Melissa [31] supports a design where a server processes data from multiple independent simulation groups that connect dynamically. Dirand's TINS system [32] approached the problem from a task-based perspective, with resources being allocated for analytics when such tasks emerged. LOOM [33] is a framework for tightly coupled in situ visualization that interweaves tasks to reduce idle times of simulation threads. While our approach is also elastic, it differs from these previous works in our focus: achieving cost savings by dynamically arranging execution to minimize fundamental in situ inefficiencies.



(a) Inline: No dedicated visualization resources, alternate between simulation, independent tasks, and collective tasks. Idle times are caused by variability.



(b) In transit: Independent and collective tasks are done on dedicated resources, idle times caused by rightsizing.

(c) Hybrid: Dynamic distribution of independent tasks between visualization and simulation nodes. Collective tasks are processed on visualization nodes only.

Fig. 1: Gantt diagrams of the two conventional in situ processing schemes (a, b) and our hybrid approach (c).

3 Motivation

This section considers how in situ approaches suffer from inefficiency. For background, in situ visualization generally occurs within "mega-cycles," which perform both simulation and visualization. Specifically, a mega-cycle consists of advancing a simulation from some cycle n to another cycle $n+m$ as well as visualizing the data from the previous mega-cycle (i.e., cycle n). Further, within a mega-cycle, there are two different types of visualization tasks to perform: (1) tasks executing independently from each other (e.g., rendering images of a data partition), and (2) collective tasks executing on all pieces of data at once (e.g., compositing partial result into one final image). Fig. 1 shows how these tasks are scheduled within a mega-cycle for the three in situ approaches using notional Gantt charts.

The remainder of this section is organized as follows: Sec. 3.1 describes four types of inefficiency, Sec. 3.2 considers how the traditional in situ processing types (inline and in transit) suffer from different kinds of inefficiency, and Sec. 3.3 describes the opportunities for a hybrid approach to reduce inefficiency.

3.1 In Situ Inefficiencies

Inefficiencies with in situ processing stem from two main categories: running in parallel and running on separate resources. Further, each of these two categories has two distinct types of inefficiency.

The two inefficiencies from running in parallel are:

(i) **Variability**: certain operations execute for variable amounts of time, and the nodes that run for longer create a bottleneck that leads other nodes to sit idle. In particular, the cost of rendering images varies significantly across nodes depending on the input data and rendering parameters (e.g., transfer function for volume rendering), especially when using acceleration techniques like empty space skipping [34]. The workload for the visualization of a node's data partition typically also shifts as the simulation progresses.

(ii) **Scalability**: certain operations exhibit limited scalability, and running them at scale causes all nodes to run inefficiently. In particular, the compositing of partial images (sub-images) into one final image frequently exhibits poor scalability [35].

These inefficiencies are related, but distinct. In particular, an algorithm can suffer from delays due to scalability even if every compute node has the same amount of work to perform. Further, an algorithm with no parallel coordination is very scalable, but it can suffer from delays due to variability if some compute nodes have much more work to perform than others.

The two inefficiencies from running on separate resources (i.e., dedicated visualization nodes) are:

(iii) **Overhead**: transferring data from the simulation nodes to the visualization nodes causes delays in multiple ways: the simulation must take time for encoding and sending, the visualization routines must receive and decode the data, and the network has extra traffic.

(iv) **Rightsizing**: visualization tasks rarely exactly align with the number of visualization nodes. If there are too many nodes for the desired tasks, then the visualization nodes sit idle. If there are too few visualization nodes for the desired tasks, then either the simulation nodes will need to block and wait for them to complete or tasks need to be dropped.

3.2 Traditional In Situ: Inline and In Transit

For ***inline*** (Fig. 1a), there are no visualization nodes, so the only inefficiencies are from running in parallel are **(i) variability** and **(ii) scalability**. That said, these two inefficiency types are often significant for inline: since all nodes participate in rendering and compositing the effects of scalability and variability typically get worse as scale increases.

In transit (Fig. 1b), provides an opportunity to save on inline's inefficiencies, but its use of separate resources creates new inefficiencies. In terms of savings, **(ii) scalability** inefficiency can be reduced by scheduling collective tasks on the visualization nodes, which are typically smaller in number than the simulation nodes. Further, **(i) variability** can potentially be addressed by identifying simulation nodes with independent tasks that have higher cost and reassigning some of those tasks to visualization nodes. That said, in

transit can suffer from issues due to **(iii) overhead** and **(iv) rightsizing**, for the reasons discussed in Sec. 3.1.

3.3 Hybrid In Situ

Hybrid in situ (Fig. 1c) refers to using a mixture of inline and in transit techniques. With this work, we consider a specific form of hybrid in situ where simulation nodes perform all simulation work as well as some visualization work, while visualization nodes only do visualization. At the beginning of a mega-cycle, both simulation nodes and visualization nodes tackle visualization tasks. At some point, simulation nodes stop performing visualization tasks and resume the simulation, while visualization nodes concurrently process their visualization tasks.

This form of hybrid in situ creates opportunities for addressing all four inefficiencies:
(i) Variability. The visualization nodes can take work from the most overloaded simulation nodes, reducing delays due to bottlenecking.
(ii) Scalability. Non-scalable tasks can be run on the dedicated visualization nodes, saving time.
(iii) Overhead. Transfer times can be overlapped with doing visualization work on the simulation nodes.
(iv) Rightsizing. The work distribution between simulation and visualization nodes is dynamically adapted. If there are few visualization nodes, then they will receive only the work they can perform in their allotted time and the remainder can be performed on the simulation nodes. If there are more visualization nodes, then they can assume more work, and simulation nodes resume the simulation more quickly.

The main challenge for hybrid in situ is the distribution of independent tasks and collective tasks to simulation nodes and visualization nodes in a way that minimizes these inefficiencies. Addressing this challenge is a main focal point of our proposed method.

4 Hybrid In Situ Method for Image Database Generation

This section describes our hybrid in situ method for generating Cinema-style image databases. An image database consists of a collection of n renderings, each corresponding to a different camera position in our case. Creating each image involves two types of operations: (1) rendering sub-images across nodes $s \in S$ (independent tasks I_s), and (2) compositing sub-images to the final result (a collective task C). This means that there are $|S| \cdot n$ independent tasks and n collective tasks in each mega-cycle.

Our hybrid in situ system (Sec. 4.1) addresses the involved inefficiencies. Compositing suffers from (ii) scalability inefficiency, which we reduce by performing it only with the generally significantly lower number of visualization nodes. To address (iii) overhead, the system overlaps data transfer and visualization work. Rendering suffers from (i) variability inefficiency, as the rendering cost heavily depends on the data generated by simulation node $s \in S$ and the camera configuration associated with task $i \in I$. This is addressed together with (iv) rightsizing by distributing rendering tasks I such that idle time across all nodes is minimized. For this, we first estimate how long rendering and compositing will take (Sec. 4.2), and then schedule visualization work accordingly (Sec. 4.3).

4.1 System Overview

Fig. 2 gives a sequential overview of our system. Although our approach is not limited to a specific rendering or compositing technique, we use volume raycasting and direct-send compositing in our system. A mega-cycle starts with visualizing the simulation results from the previous iteration. First, we create an estimation of induced cost on the simulation nodes $s \in S$ (Sec. 4.2). Probing carries out and measures a subset of the render tasks $I'_s \subset I_s$ to estimate their cost. There is global synchronization between all nodes to exchange probing timings (this is the only instance of global synchronization in our approach). Compositing time is predicted using a simple performance model. This provides the basis for visualization load assignment, which consists of two phases (Sec. 4.3): first, each simulation node is assigned to one visualization node ($N : S \rightarrow V$), and then the remaining render tasks $I^*_s = I_s \setminus I'_s$ are distributed between a simulation node s and its visualization node $N(s)$. Simulation data is accordingly distributed to the visualization node that took over respective rendering tasks. Likewise, all render parts produced on s are moved to its assigned $N(s)$. These and all other data sending and receiving operations are asynchronous both on simulation and visualization nodes. This allows the system to effectively hide induced latency, i.e., simulation nodes can render while they are sending, and also visualization nodes can process tasks while they are receiving. After rendering, simulation nodes immediately continue with the simulation. Visualization nodes $v \in V$ perform classic direct send compositing with image parts rendered by them as well as associated simulation nodes s (i.e, with $v = N(s)$). Since the images are composited sequentially, it allows us to compress and write them to disk concurrently. Details on the implementation of our system, the integration into simulation codes, and design decisions can be found in the supplemental material.

4.2 Render and Compositing Time Predictions

Render Probing. At the beginning of each mega-cycle, all simulation nodes carry out probing rendering. For this, we randomly sample from all render

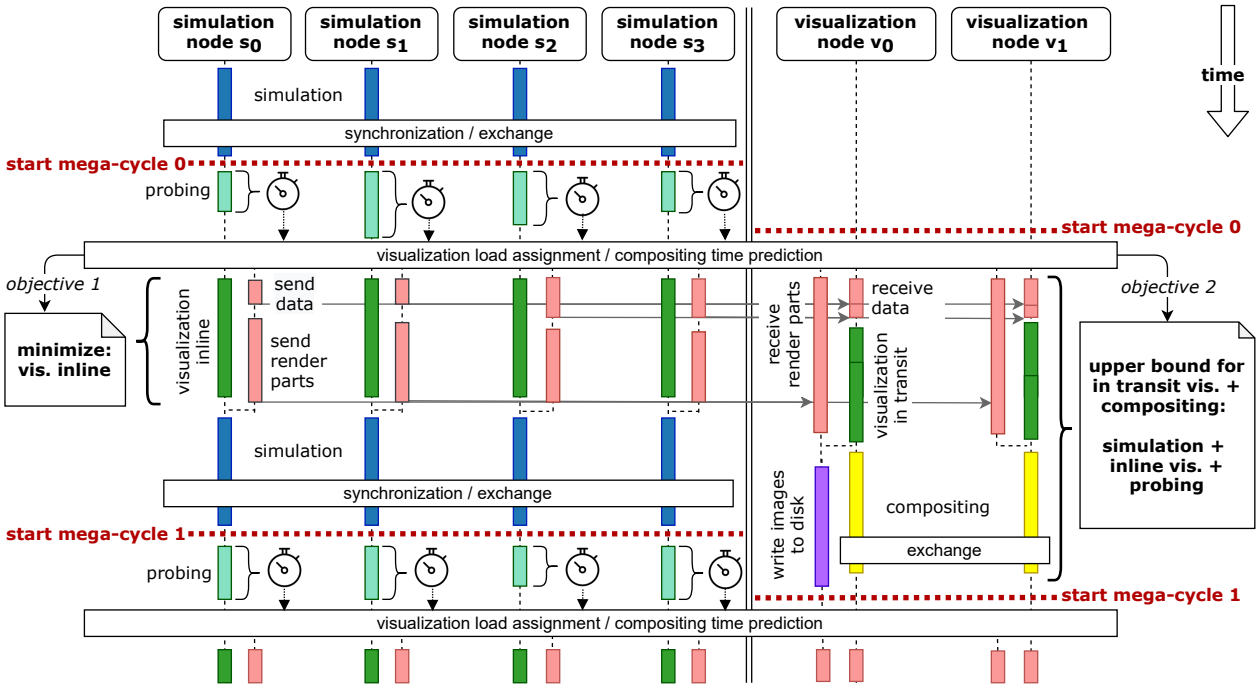


Fig. 2: Sequence diagram of our system. Probing times are used as a basis to distribute the rendering load. Our main objective is to minimize the inline visualization time and the upper bound for time spent rendering on the visualization nodes is constrained by the combined time of rendering inline (including probing) and the simulation time before the next visualization step. Compositing is done on visualization nodes only. All sub-images created inline or during probing are sent to the visualization nodes.

tasks I_s of the respective nodes s in each mega-cycle to select probing samples $I'_s \subset I_s$. We then render and measure timings for these on s as the basis to predict the costs of the remaining tasks I^*_s. We use the arithmetic mean of the probing render times as a runtime estimate (per render) of the respective data partition. Typically, a random sampling of positions of an arcball-style camera provides a good coverage of the overall performance distribution [34]. During probing, we also detect if rendering can be skipped by checking whether the scalar value range in the data partition of s always yields opacity values below a threshold of 0.001 for the provided transfer function.

Compositing Time. For direct send compositing, cost can be estimated as a function of nodes participating [35], [36]. In our case, we specifically consider whether nodes produce images that contribute to the final results (i.e., which were not skipped via opacity checking):

$$1.05 \cdot \left(\alpha + \beta \cdot |V^*| \cdot \frac{|S^*|}{|S|} + \gamma \cdot |V^*|(1 - \frac{|S^*|}{|S|}) \right).$$

Here, $|V^*|$ (with $V^* \subseteq V$) is the number of visualization nodes actively participating in compositing and α, β, γ are empirically determined constants on a target system. $|S|$ is the number of simulation nodes, while $|S^*|$ divided by the total number of simulation nodes represents the normalized amount of dataset partitions that actually need to be visualized (i.e., the ratio between skipped and non skipped partitions).

Since we determine α, β, γ empirically, a few measurements of compositing times are needed beforehand when running our system on hardware using a different interconnect. We determine the constants using the measurements and a non-linear least squares function fit, which resulted in a normalized root-mean-square error of 8.83% for our compositing time estimation. To account for the uncertainty in the prediction, we conservatively overestimate the time by 5% to avoid blocking of the simulation nodes.

4.3 Visualization Load Assignment

Node assignment $N : S \rightarrow V$**.** The estimated rendering time from probing for the remaining images $I^* = I_s \setminus I'_s$ provide the basis for assigning each simulation node $s \in S$ to one visualization node $v \in V$. For this, we iteratively assign the simulation node with the highest cost to the visualization node with the lowest accumulated cost until all nodes are distributed.

Rendering task assignment $A_s : I^*_s \rightarrow (s, N(s))$**.** Next, rendering tasks I^*_s that remain after probing on

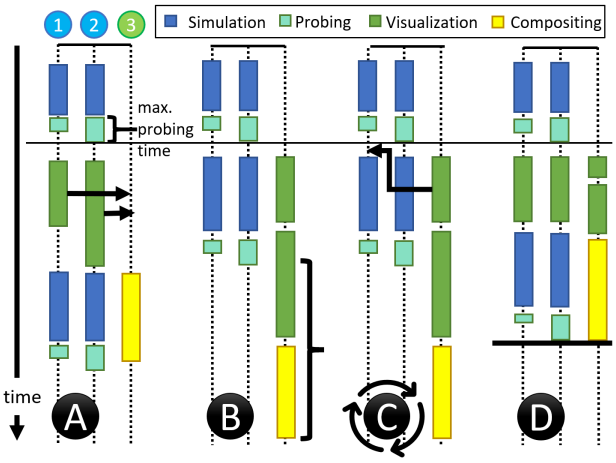


Fig. 3: Our load balancing scheme exemplified with two simulation nodes (1, 2) and one visualization node (3). Initially, all visualization load is assigned to the visualization resource (A). If the estimated runtime there exceeds the full cycle time on the simulation resources (B), we iteratively shift render load to the simulation node with the currently lowest total (anticipated) runtime (C) until runtimes are evenly balanced across all nodes (D).

a simulation node s are scheduled to be either tackled by s or its assigned visualization node $N(s)$. Our main objective is to minimize the inline visualization time in order to maximize the time that the simulation node can use for simulation. Initially, we expect all rendering to be done on the visualization nodes (Fig. 3A). However, when aiming to avoid idle times, there is an upper bound for time spent rendering and compositing on the visualization nodes (Fig. 3B). It is constrained by the combined time of three parts: (inline) rendering time on simulation nodes (influenced by A_s), the simulation steps before the next visualization run on the simulation nodes, and the maximum probing time of the next visualization cycle. The probing time is approximated from the current megacycle. As long as the combined time predictions for rendering and compositing on the visualization nodes exceed the time of those three parts, we gradually shift rendering load to inline on the simulation nodes. For this, we consider the currently fastest simulation node, randomly pick a corresponding image part that is currently not assigned to it, and transfer this task from the respective visualization node (updating the respective time estimates in the process, Fig. 3C). This balances the render load across simulation and visualization nodes (Fig. 3D).

5 Overview of Experiments

Our experiments are organized into three phases.

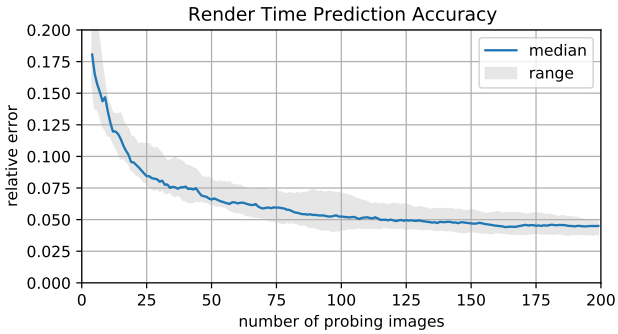


Fig. 4: Render time estimation error depending on the number of probing images (total: 400). The data is based on twelve random sequences.

- A **parametric study** evaluates different configurations with varying amounts of work (number of images in the image database) and resources (number of visualization nodes).
- **In-depth experiments** are conducted to compare our hybrid method with inline and in transit approaches for a selected configuration.
- A **weak scaling study** evaluates how changes in concurrency affect cost.

The remainder of this section describes software, hardware, and workload details.

Software: We used the *Ascent* [11] in situ framework, which can generate *Cinema* [8] image databases. Ascent implements parallel rendering by using *vtk-m* [37] for shared-memory parallelism and MPI for distributed-memory parallelism. Ascent natively does inline and in transit, and we extended it to do our hybrid method for this study. Further, direct-send compositing is implemented using the *DIY2* [38] library. Reproducibility details about the integration can be found in the supplemental material. Ascent, VTK-m, and Cinema are all open source, as are our extensions.

One important aspect to our software is how much probing to perform. Prior to our experiments, we conducted an analysis to identify a good trade-off between performance and accuracy. This analysis used an example case with 400 renderings. To calculate the accuracy, we generated twelve random sequences of camera positions (see Sec. 4.2), and generated estimations using the sequence with the first $n \in [4, 200]$ probings (Fig. 4). With this, we identified $15\,\%$ of the renderings to be a good compromise of scheduling flexibility versus accuracy (yielding an error of $6.3\% \pm 1.3\%$), and use this ratio in all experiments that generate 400 renderings.

Hardware: We ran all our experiments on TACC's Stampede2 supercomputer. We used SKX nodes that feature Intel Xeon Platinum 8160 CPUs with 48 cores on two sockets (24 cores/socket). The CPUs support two hardware threads per core, adding up to a total of

96 threads per node. Typically, we ran 6 MPI tasks per node with 16 OpenMP threads per task.

When comparing across inline, in transit, and hybrid, we fixed the number of simulation nodes and considered different numbers of visualization nodes. This is crucial for comparability as the number of simulation nodes impacts the domain decomposition of the simulation, which not only influences the simulation itself but also the rendering tasks.

Workloads: A workload consists of running a simulation code for some number of mega-cycles, as well as generating an image database for each mega-cycle. We determine the number of simulation steps in all mega-cycles by running the simulation in intervals of 120 seconds of wall clock time before invoking the in situ visualization. For image databases, we generate a Cinema database of 400 volume renderings per visualization cycle using an orbital camera with regular spacing of the angles and a single zoom level. The images have a standard resolution of 800×800 pixels, unless noted otherwise. Experiments with larger image resolutions can be found in the supplemental material. We use front-to-back volume raycasting accelerated by early ray termination and block-based empty space skipping. Finally, we employed supersampling during the weak scaling phase.

Two simulation codes were used throughout our study. The first two phases used *Cloverleaf3D* [39], a 3D Lagrangian-Eulerian hydrodynamics benchmark. The third phase used *Nyx* [40], a massively parallel code for cosmological hydrodynamics simulations as a real world example. We visualize the energy field for Cloverleaf3D (Fig. 6) and the density field for Nyx (Fig. 7). While Cloverleaf3D uses regular grids, Nyx uses block structured adaptive mesh refinement with *AMReX* [41]. The simulations also differ in how they evolve over time. The Cloverleaf3D simulation starts with two initial energy fields in opposite corners of its domain, and these two energy fields extend over the course of the simulation until they visually fill the whole domain. The volume rendering's transfer function treats low energy regions as fully transparent, resulting in imbalanced work from empty space skipping. As the simulation continues this effect fades, but rendering imbalances emerge due to early ray termination. With Nyx, the simulation starts with an initial random seed of dark matter particles distributed across the whole domain. Over the course of the simulation, the particles attract each other to form clusters, creating empty spaces as a side effect. As a result, data blocks become less active over time.

6 Results

Our results are organized into parametric study (Sec. 6.1), in-depth experiments (Sec. 6.2), and weak scaling experiments (Sec. 6.3).

6.1 Parametric Study

In our first phase, we compare different combinations of workload (i.e., varying image count) and resources (i.e., number of visualization nodes). This phase consisted of 36 experiments, as a cross product of four image database sizes (81, 144, 256, and 400 images) and nine in situ configurations. Eight of the in situ configurations came from varying the number of visualization nodes (1, 2, 4, and 8) for both hybrid and in transit. The ninth configuration was running inline. Each configuration ran Cloverleaf with eight simulation nodes. Four of the 36 experiments in this phase are investigated in more detail in Sec. 6.2. Further, results of a similar study using different image resolutions instead of varying image counts can be found in the supplemental material.

Fig. 5 compares the efficiency of our hybrid approach with inline and in transit in a 4×4 matrix. The lower left of this matrix has the least work per visualization node (81 images and 8 visualization nodes), while the upper right of this matrix corresponds to the most work per visualization node (400 images and one visualization node). Our hybrid technique has the most opportunity for cost savings when there is more work, since there is more **(i) variability** we can reduce. Further, when the amount of work per visualization node becomes too low, then these nodes will have to sit idle while the simulation advances, making **(iv) rightsizing** impossible. The fact that the probing step in hybrid conceptually pins some work to the simulation nodes contributes to the issue.

Our results confirm the importance of sufficient work per visualization node for the success of our technique. Eight of the sixteen configurations ran fastest with our hybrid method. The best savings were achieved in the "upper right" configuration (most work per visualization node), with savings decreasing as the number of visualization nodes increased or the number of images decreased. The remaining eight configurations ran faster with either inline (seven times) or in transit (one time). The worst performance of our hybrid approach occurs in the "lower left," with performance increasing as the number of visualization nodes decreased or the number of images increased. Finally, for the configurations where our hybrid approach performs poorly, inline's strong performance may fade at higher scale, as inline tends to perform worse at larger scales as can be seen in Sec. 6.3.

Some of the sixteen configurations show our technique's flexibility in rightsizing. In particular, our hybrid technique took approximately the same amount of time to render 256 images whether it was assigned one visualization node or two visualization nodes. This is because our algorithm was able to adapt the assignments to do more work on the simulation nodes when there was one visualization node and more work on

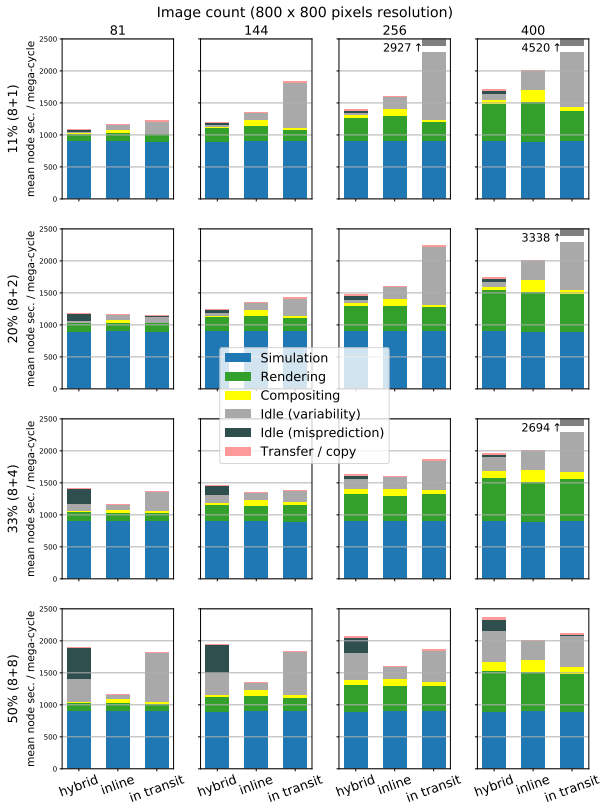


Fig. 5: The columns in the 4×4 matrix correspond to the work (i.e., number of images rendered), while the rows reflect the resources (i.e., number of visualization nodes). Each of the 16 stacked bar charts compares our hybrid in situ method with inline and in transit. The colors correspond to different activities, and the heights for each color indicates how much time was spent (on average) per mega-cycle. Broken bars indicate higher y-values. As the inline configuration does not use visualization nodes, a single inline run is repeated along each column. There are small variations in render times between the techniques for some configurations that we attribute to our use of 6 MPI ranks per physical node and hyper-threading, resulting in slightly different utilization of physical cores.

the visualization nodes where there were two. The cost is the same across the two experiments because both have the same savings on scalability and variability, overheads do not increase, and (critically) rightsizing is maintained for both. In all, this demonstrates that our method yields rightsizing in fairly wide ranges of configurations, while in transit is only able to achieve this in the rare case when all conditions align.

Finally, when the work per visualization node becomes extremely high, our method's performance can resemble a pure inline approach, and thus be subject to **(i) variability**. We did observe this in practice (see discussion in Sec. 6.2 on unavoidable idle from variability at mega-cycle 0), but the effect was small enough that our method was still the most efficient in comparison.

6.2 In-depth Experiments

Our in-depth experiment considered a single workload that was selected based on the results of the parametric study (Sec. 6.1). We ran Cloverleaf3D for 14 mega-cycles on eight simulation nodes, with a grid resolution of 384^3 (192^3 per node). For the image database, we generated 400 images in each mega-cycle. We compare four in situ approaches: inline (8), hybrid (8+2) with two dedicated visualizations ranks (i.e., 10 nodes total), as well as in transit (8+4) and in transit (8+8) with 4 and 8 dedicated visualization ranks, respectively. Fig. 6 shows volume renderings from these experiments, as well as Gantt charts for each in situ approach.

Our hybrid technique was the most efficient approach, requiring the least node seconds in total. It completed the simulation and visualization tasks in $2530\,\mathrm{s}$ using 10 nodes (25.3K node-seconds), while the inline configuration took $3576\,\mathrm{s}$ using 8 nodes (28.6K node-seconds—$13.1\,\%$ more). The in transit configuration took $3350\,\mathrm{s}$ with 12 nodes (40.2K node-seconds—$58.9\,\%$ more) and $1947\,\mathrm{s}$ with 16 nodes (31.2K node-seconds—$23.1\,\%$ more). The flexibility of hybrid enabled it to do better in terms of the four types of in situ inefficiency (see Sec. 3.1), despite introducing **(iii) overhead** for transfer (pink) in comparison to inline, and also exhibiting some **(i) variability** issues due to sub-optimal work assignments (black).

For inline, the effects from **(i) variability** can be seen in the high proportion of idle time (gray) in simulation ranks 1 through 6 while simulation ranks 0 and 7 are rendering (green). These effects are significant from mega-cycles 0 through 4, with only two corners of the volume actually contributing to the volume rendering. As the simulation evolves, this improves when all nodes engage in rendering work in mega-cycles 5 through 8, but inefficiencies re-emerge in mega-cycles 9 through 14 due to early ray termination (node 7 in particular). Our hybrid technique addresses variability by adapting the assignments to visualization ranks accordingly. Inline also demonstrates **(ii) scalability** issues during the compositing phase (yellow). Compositing costs were 160 node-seconds per mega-cycle with 8 simulation ranks participating for about $20\,\mathrm{s}$, while hybrid only required 100 node-seconds with just 2 nodes being involved for $50\,\mathrm{s}$.

Both in transit configurations perform quite poorly in comparison. Hybrid reduces **(iii) overhead** inefficiencies with respect to in transit both by overlapping data transfer with visualization work and only triggering it when simulation nodes cannot process all visualization work themselves. However, for both in

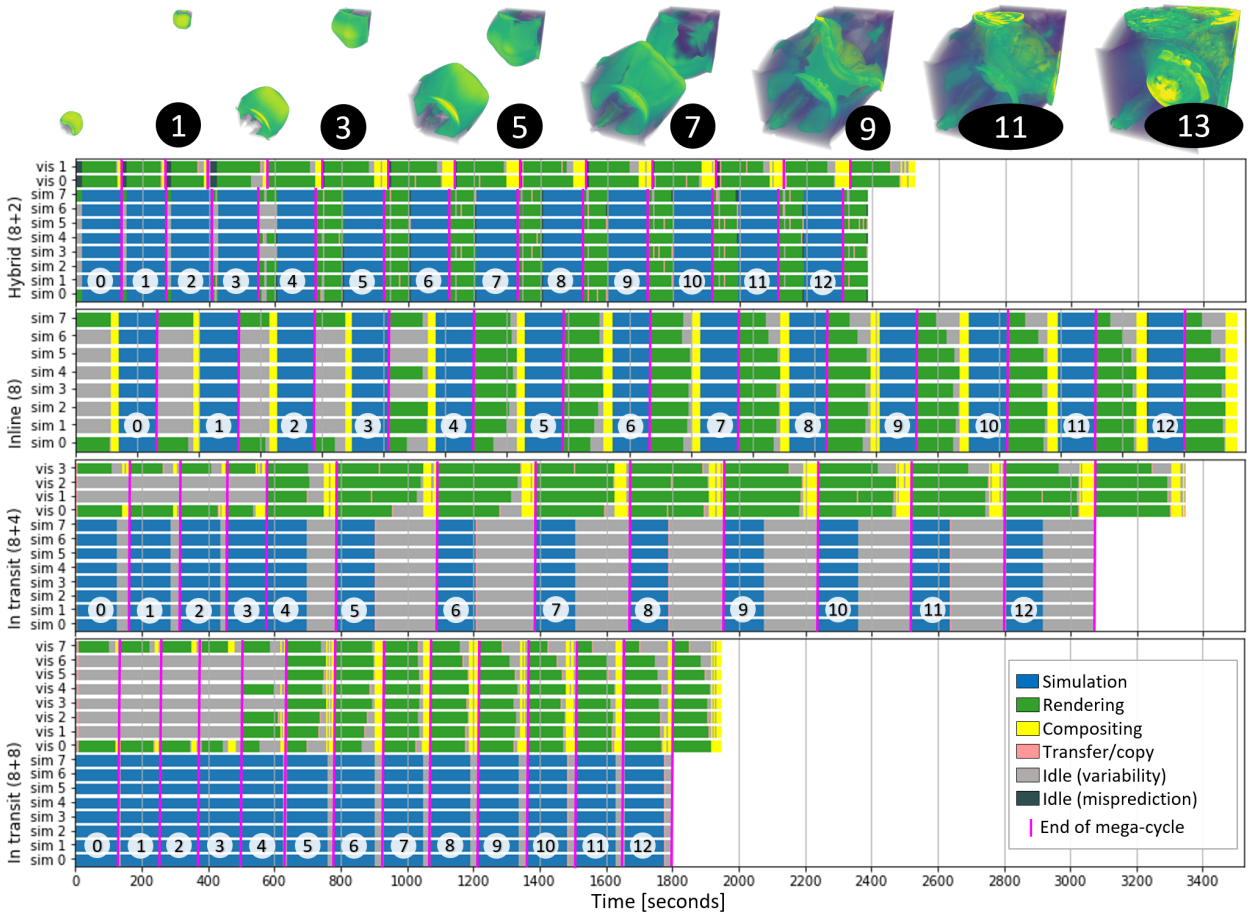


Fig. 6: In-depth experiment of 14 mega-cycles of Cloverleaf3D simulation and Cinema database generation. Images at the top show volume renderings from every other mega-cycle. The remaining rows show Gantt charts for inline, hybrid, and in transit configurations. The charts are colored to indicate the activity on a node, including two types of idle. Light gray depicts idle caused by variability while dark gray shows idle times caused by prediction errors in our hybrid approach. Although in transit (8+8) has an overall shorter completion time, hybrid is more efficient in terms of node seconds overall.

transit cases, the major issue is **(iv) rightsizing**—one configuration has too few visualization nodes and the other has too many. With too few visualization nodes (4 visualization nodes, 12 total), the rendering tasks cannot be completed in time, blocking the simulation nodes. This results in significant idle time (gray) on the simulation nodes, especially after mega-cycle 4. With too many visualization nodes (8 visualization nodes, 16 total), there are not enough rendering tasks to occupy the visualization nodes. This again results in significant idle time, although this time on the visualization nodes and before mega-cycle 4. Together, these two configurations demonstrate the difficulty in rightsizing in transit resources—whether too few or too many visualization resources, the result is idle time. In contrast, hybrid achieves rightsizing over a variety of visualization workloads by dynamically assigning render tasks. In early mega-cycles, the visualization ranks can almost exclusively handle the rendering tasks, allowing simulation ranks to focus on the simulation. When the cost of rendering tasks increase (around mega-cycle 5), work is shared between visualization and simulation ranks such that they complete their respective tasks right as the mega-cycle ends.

Although our hybrid approach was able to improve efficiency overall, this experiment also demonstrates some limitations in highly unbalanced scenarios. Specifically, the variability is extreme in mega-cycle 0: simulation nodes #0 and #7 have all the inline visualization work and the others have none. For this level of imbalance, the ratio between visualization nodes and simulations, and the amount of rendering work compared to the length of the mega-cycle, hybrid cannot schedule tasks that will fully prevent idling on simulation nodes #1–#6.

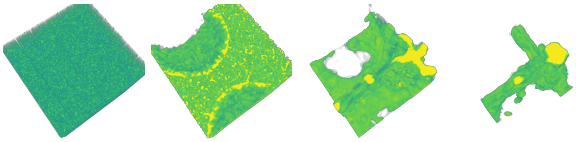


Fig. 7: Renderings of the Nyx simulation data.

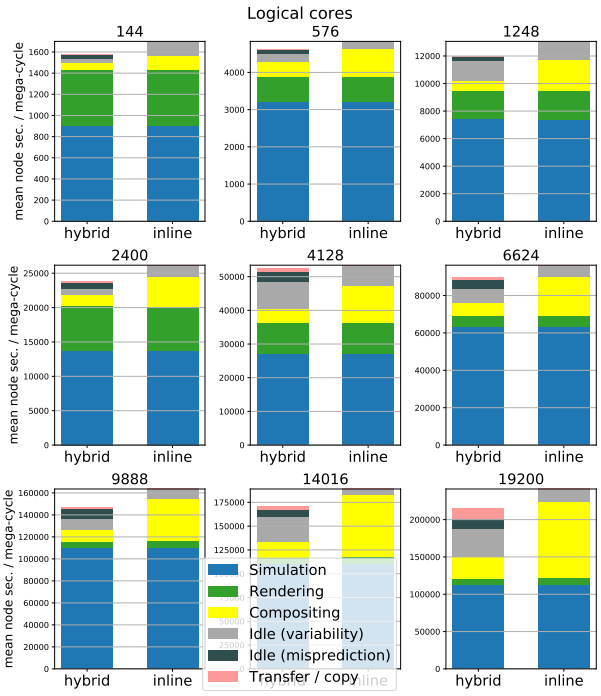


Fig. 8: Nyx weak-scaling results. For inline, compositing times grow with higher thread counts, while rendering gets relatively faster. For hybrid, overhead and render times grow, as well as idle times caused by prediciton errors. In transit runs are not considered since their results are substantially worse than hybrid or inline due to the rightsizing problem.

6.3 Weak Scaling

Finally, we consider a weak scaling study with the Nyx simulation code, comparing our hybrid approach against pure inline. Based on our parametric study (Sec. 6.1), we aimed for a visualization to simulation resource ratio of $\frac{\text{visualization}}{\text{simulation}} = 0.2$ and used a data partition size of 32^3 per simulation rank. The randomly seeded dark matter particle count was adapted accordingly. We ran Nyx with nine different node configurations on up to 19 200 logical cores. Configuration details are listed in Table 1, sample renderings are shown in Fig. 7 (more can be found in the supplemental material). The node configurations result from the constraints that the MPI task count needs to be divisible by 6 to fully occupy the nodes, and the simulation task count needs to correspond to the number of (uniform) data partitions. We increased supersampling in the volume raycaster for higher node counts to balance out the render load decrease at higher concurrencies that is caused by our constant image resolution. The run was interrupted after 1 hour of execution (hybrid) respective 1 hour 20 minutes (inline). As discussed above, we allowed the inline case more processing time to get a similar number of full cycles as in the hybrid case.

TABLE 1: Node Configurations for Nyx on Stampede2.

Nodes	Logical cores	Sim. ranks	Vis. ranks	Factor	Grid size	Super-sampling
2	144	8	1	0.125	64^3	1×1
6	576	27	9	0.333	96^3	1×1
13	1248	64	14	0.219	128^3	2×2
25	2400	125	25	0.200	160^3	3×3
43	4128	216	42	0.194	192^3	4×4
69	6624	343	71	0.207	224^3	4×4
103	9888	512	106	0.207	256^3	4×4
146	14016	729	147	0.202	288^3	4×4
200	19200	1000	200	0.200	320^3	4×4

The performance summaries plotted in Fig. 8 show that the four types of inefficiency change as concurrency increases. The most obvious effect is with scalability. The inline approach devotes more and more time to compositing (yellow color) due to poor scalability, while our hybrid approach is able to reduce compositing time significantly. That said, as can be seen in Fig. 8, the compositing time increases with concurrency for our hybrid technique as well, as the number of visualization nodes increases proportionally and begin to exhibit their own scalability inefficiency. Another important effect is with overhead, which can be seen in the increasing transfer/copy times (pink) for very high core counts. With respect to rightsizing, our algorithm was able to make "rightsized" assignments for visualization and simulation work (i.e., all tasks should finish a mega-cycle at the same time), but these assignments did sometimes lead to idle resources because of mispredictions (black color). That said, the amount of misprediction does not appear to change significantly as concurrency increases. Finally, variability effects (gray color) increase slightly for very high concurrencies for our hybrid method and stay at a similar level for inline visualization. This effect has mainly two reasons. First, the variability in data decrease at higher concurrencies since the simulation progresses slower with respect to wall-clock time and we use the same amount of time per mega-cycle (more details and example renderings can be found in the supplemental material). Second, overall rendering times are getting smaller, since each block contributes fewer fragments.

In terms of actual savings, our hybrid approach had lower cost for all concurrencies, although these savings varied (144 cores: 6.9%, 576: 2.8%, 1248, 9.0%, 2400: 8.5%, 4128: 2.0%, 6624: 6.9%, 9888: 10.2%, 14016:

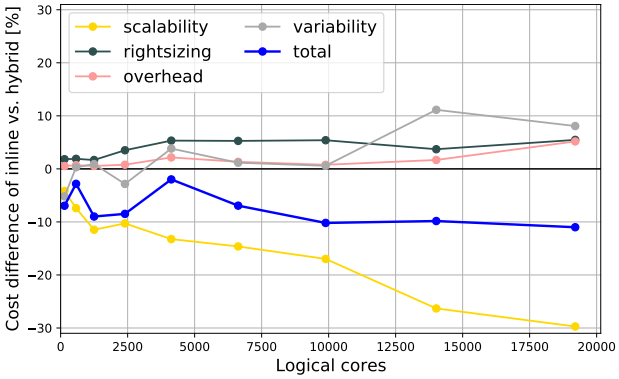


Fig. 9: Differences in cost between hybrid and inline as a function of concurrency. The differences are calculated relative to the inline run. For example, the total cost for the 2400 core inline experiment was 26.5K seconds, of which 4310 s were spent compositing. The corresponding hybrid run took 1700 s for compositing, representing a savings of 2610 s, which is 10% of the total inline run. In turn, the yellow (scalability) curve has a point at (2400, -10%). The overhead curve (pink) considers both transfer costs and reduced performance in rendering due to the overlap with transfers.

9.8%, 19200: 11.0%). While the savings are fairly consistent, the factors behind them are changing. These changes can be seen in Fig. 9. As concurrency increases, scalability savings are growing fast enough to offset additional overhead costs and reduced savings in variability.

7 Conclusion and Future Work

The premise of this work was that a hybrid in situ visualization approach had the potential to address fundamental inefficiencies in a way that the traditional inline and in transit techniques could not. That said, it was merely our hypothesis that the savings we could achieve would be significant—the magnitude of the resulting savings were unknown and required running experiments for evaluation. We find the results to be overall very promising. Compared to inline, we were able to achieve a cost savings for the majority of the configurations we considered. Of the remaining configurations, it was often infeasible for a hybrid method to achieve cost savings, as there was not enough work to justify the extra visualization nodes. These configurations can be detected ahead of time and eliminated (by requesting fewer visualization nodes), which would make our hybrid method superior at a much higher rate. Overall, we felt one of the strongest elements of our design was the ability to adapt to different ratios of work (images to render) and resources (extra visualization nodes), meaning that it should be able to operate robustly in production settings.

We also were enthused by our comparisons with in transit processing. While Kress et al. [21] was the first to show that in transit could save on scalability inefficiency enough to offset overhead, their approach was significantly more susceptible to rightsizing effects. In other words, they achieved maximum savings when their resource allocation had rightsizing harmony, but any deviation from that harmony immediately began reducing those savings. Our approach, however, is able to achieve rightsizing over a wide variety of configurations, and therefore maintain maximum savings much more often.

Although the actual savings in cost may appear to be modest (on the order of 7.5%), we feel this research has a chance to be very impactful—small speedups for ubiquitous operations on expensive devices add up to a large impact overall. For example, if all the jobs on a $200M supercomputer utilized our in situ visualization approach, then 7.5% savings would be the equivalent of $15M of extra compute power over the life of the machine. Of course, not all jobs will use our approach, and the benefit reduces proportionally. That said, simulations are increasingly adopting Cinema to generate image databases, thus increasing the need for our approach and increasing its potential benefit. However, we acknowledge that the design is somewhat complex, and this complexity will probably need to be hidden behind production software, such as we have done with Ascent.

Overall, our approach exhibits two main parameters: (i) the ratio of visualization nodes and (ii) the ratio of work items used for probing. As a rough guideline, (i) the ratio of visualization nodes should be chosen such that it is lower than the expected ratio of visualization costs regarding the simulation costs, whereas for (ii) the number of probing items needs to be sufficient to adequately reflect the total cost of work items. Too many visualization nodes means they cannot be efficiently occupied with work throughout, and too many probing items limits the flexibility regarding work distribution. In our current approach this needs to be set manually, but could also be determined automatically with a small prior test run (subject to future work). As discussed in the related context of rightsizing above, we consider a main strength of our approach to be its flexibly in adapting to different settings, yielding robustness in the sense that execution is efficient for a range of parameter settings.

Our approach should generalize to other visualization or analysis tasks that can be split into fine granular sub-tasks for dynamic distribution. The image database generation considered in this work demonstrates benefits for a combination of independent tasks (the rendering of images) as well as reduction (the compositing of partial results), which is a common

scheme for various distributed visualization methods. In general, we anticipate that our approach would particularly be beneficial for computation-heavy tasks with heterogeneous costs and a high degree of parallelism, potentially also requiring communication between nodes. Visualization techniques using intermediate representations (e.g., explicit isosurface rendering) could be approached in two different ways: either the intermediate representation could be generated on the simulation node and then this representation would be distributed instead of the data, or the data partition could be further subdivided and a work item would comprise all operations—comprising intermediate representation generation and rendering—on a respective sub-partition of the data. We also expect our hybrid approach to profit at scale for visualization techniques that need additional communication (e.g., passing around rays for path tracing). We aim to investigate further visualization scenarios with our approach in future work.

We also see potential to further refine our approach for image database generation. Here, we aim to focus on improving the accuracy of cost predictions via importance sampling for probing and advanced compositing and simulation time predictions.

Acknowledgment

This work was partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) within Project A02 of the SFB/Transregio 161 (project number 251654672) and by the Intel Graphics and Visualization Institutes of XeLLENCE program (CG #35512501). This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas for providing HPC resources.

References

[1] R. Binyahib, T. Peterka, M. Larsen, K.-L. Ma, and H. Childs, "A scalable hybrid scheme for ray-casting of unstructured volume data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 7, pp. 2349–2361, 2018.

[2] S. Dutta, C. Chen, G. Heinlein, H. Shen, and J. Chen, "In situ distribution guided analysis and visualization of transonic jet engine simulations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 01, pp. 811–820, 2017.

[3] 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 Transactions on Visualization and Computer Graphics*, vol. 22, no. 01, pp. 857–866, 2016.

[4] P. O'Leary, J. Ahrens, S. Jourdain, S. Wittenburg, D. H. Rogers, and M. Petersen, "Cinema image-based in situ analysis and visualization of mpas-ocean simulations," *Parallel Computing*, vol. 55, pp. 43–48, 2016.

[5] A. C. Bauer *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," *Computer Graphics Forum*, vol. 35, no. 3, pp. 577–597, 2016.

[6] H. Childs, J. Bennett, C. Garth, and B. Hentschel, "In Situ Visualization for Computational Science," *IEEE Computer Graphics and Applications*, vol. 39, no. 6, pp. 76–85, 2019.

[7] T. Peterka *et al.*, "Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources," *The International Journal of High Performance Computing Applications*, vol. 34, no. 4, pp. 409–427, 2020.

[8] 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 *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 424–434.

[9] H. Childs *et al.*, "A Terminology for In Situ Visualization and Analysis Systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[10] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proc. Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2015, pp. 25–29.

[11] 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 on Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2017, p. 42–46.

[12] B. Whitlock, J. Meredith, and J. Favre, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proc. Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, vol. 10, 2011, pp. 101–109.

[13] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *Proc. IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 67–75.

[14] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, "The sensei generic in situ interface," in *Proc. Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2016, pp. 40–44.

[15] W. F. Godoy *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[16] J. C. Bennett *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–9.

[17] F. Zheng *et al.*, "PreDatA–preparatory data analytics on peta-scale machines," in *Proc. IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[18] J. Dayal *et al.*, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014, pp. 246–255.

[19] M. Dorier *et al.*, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, pp. 1–43, 2016.

[20] J. Kress *et al.*, "Comparing the efficiency of in situ visualization paradigms at scale," in *High Performance Computing*. Springer International Publishing, 2019, pp. 99–117.

[21] ——, "Opportunities for cost savings with in-transit visualization," in *High Performance Computing*. Springer International Publishing, 2020, pp. 146–165.

[22] F. Zheng *et al.*, "In-situ I/O processing: a case for location flexibility," in *Proc. Workshop on Parallel Data Storage*, 2011, pp. 37–42.

[23] T. M. A. Do *et al.*, "A novel metric to evaluate in situ workflows," in *Proc. International Conference on Computational Science (ICCS)*, 2020, pp. 538–553.

[24] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, and M. Day, "In situ and in-transit analysis of cosmological simulations," *Computational astrophysics and cosmology*, vol. 3, no. 1, pp. 1–18, 2016.

[25] Z. Wang, P. Subedi, M. Dorier, P. E. Davis, and M. Parashar, "Staging based task execution for data-driven, in-situ scientific workflows," in *Proc. IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 209–220.

[26] K. Mehta *et al.*, "A codesign framework for online data analysis and reduction," in *Proc. IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2019, pp. 11–20.

[27] P. Grosset and J. Ahrens, *Lightweight Interface for In Situ Analysis and Visualization of Particle Data*, 2021, p. 12–17.

[28] M. Dorier, O. Yildiz, T. Peterka, and R. Ross, "The challenges of elastic in situ analysis and visualization," in *Proc. hop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2019, p. 23–28.

[29] F. Zheng *et al.*, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[30] A. Goswami *et al.*, "Landrush: Rethinking in-situ analysis for GPGPU workflows," in *Proc. IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 32–41.

[31] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin, "Melissa: Large scale in transit sensitivity analysis avoiding intermediate files," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[32] E. Dirand, L. Colombet, and B. Raffin, "Tins: A task-based dynamic helper core strategy for in situ analytics," in *Supercomputing Frontiers*, 2018, pp. 159–178.

[33] J. Barbosa, P. Navratil, L. Paulo Santos, and D. Fussell, "Loom: Interweaving tightly coupled visualization and numeric simulation framework," in *Proc. Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, 2021, p. 1–5.

[34] V. Bruder, C. Müller, S. Frey, and T. Ertl, "On evaluating runtime performance of interactive visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 9, pp. 2848–2862, 2019.

[35] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An image compositing solution at scale," in *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–10.

[36] M. Larsen, K. Moreland, C. R. Johnson, and H. Childs, "Optimizing multi-image sort-last parallel rendering," in *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2016, pp. 37–46.

[37] K. Moreland *et al.*, "Vtk-m: Accelerating the visualization toolkit for massively threaded architectures," *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, 2016.

[38] D. Morozov and T. Peterka, "Block-parallel data analysis with DIY2," in *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2016, pp. 29–36.

[39] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, 2013.

[40] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel AMR code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[41] W. Zhang *et al.*, "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019.

Valentin Bruder defended his PhD thesis in computer science at the University of Stuttgart, Germany in 2021. He was a researcher at the University of Stuttgart Visualization Research Center (VISUS) while this research was conducted, and is now with Daimler Truck AG. His research interests include scientific visualization methods and their performance assessment, modeling, and optimization.

Matthew Larsen received the PhD degree in computer science from the University of Oregon, in 2016. He was a computer scientist with Lawrence Livermore National Laboratory while this research was conducted, and is now with Luminary Cloud, Inc. His research interests include rendering for visualization, performance modeling for visualization, and many-core architectures.

Thomas Ertl received the MS degree in computer science from the University of Colorado at Boulder and the PhD degree in theoretical astrophysics from the University of Tuebingen. He is a full professor of computer science with the University of Stuttgart, Germany in the Visualization and Interactive Systems Institute (VIS) and a co-director of the Visualization Research Center (VISUS).

Hank Childs received the PhD degree in computer science from the University of California at Davis in 2006. He is a Professor of Computer and Information Science at the University of Oregon. His research focuses on scientific visualization, high performance computing, and the intersection of the two.

Steffen Frey received his PhD degree in computer science from the University of Stuttgart, Germany in 2014. He is an assistant professor at the Bernoulli Institute at the University of Groningen, Netherlands. His research interests are in visualization methods for increasingly large quantities of scientific data.

Supplemental Material

This supplemental material to the paper *"A Hybrid In Situ Approach for Cost Efficient Image Database Generation"* provides additional results, renderings, and implementation details. Sec. A presents further results complementing the parametric study found in Sec. 6.1 of the paper with a study on the impact of changing the image resolution used for rendering. Sec. B provides additional renderings of the Nyx simulation data, showing and discussing the changes in variability at higher scales that are observed in Sec. 6.3 of the paper. Finally, Sec. C provides further implementation details on the integration into the simulation codes as well as the integration of our hybrid approach into the Ascent library for reproducibility. Also, background information on the used libraries and APIs is provided to complement Sec. 4 of the paper.

Appendix A
Image Scaling Results

In this section, results of an additional parametric study similar to the one presented in Sec. 6.1 of the paper are shown. The experiment consisted of 27 similar experiments, but in contrast to the parametric study in Sec. 6.1, the image resolution was changed instead of the image count (which remained at 400 images). Three different resolutions were used (800^2px, 1100^2px, and 1400^2px), where the latter two roughly correspond to doubling, respectively tripling, the pixel count of the 800^2px default configuration.

This experiment series demonstrates the impact of higher render loads when using a larger image resolutions that might be desirable in some visualization scenarios, e.g., where fine detail are important. The results are shown in Fig. A.1, where the columns represent the different resolutions while the rows correspond to the number of visualization nodes (i.e., the visualization resource ratio). As in our set of experiments examining a varying number of images, the inline case was run only once per configuration since it does not use dedicated visualization resources. When compared, both experiment series show similar results, while the distance of our hybrid technique to inline even increases for higher resolutions in the cases where there is more work per visualization node (upper half).

This series underlines our previous conclusion that our method yields rightsizing in fairly wide ranges of configurations, while in transit is only able to achieve this in the rare case when several conditions align.

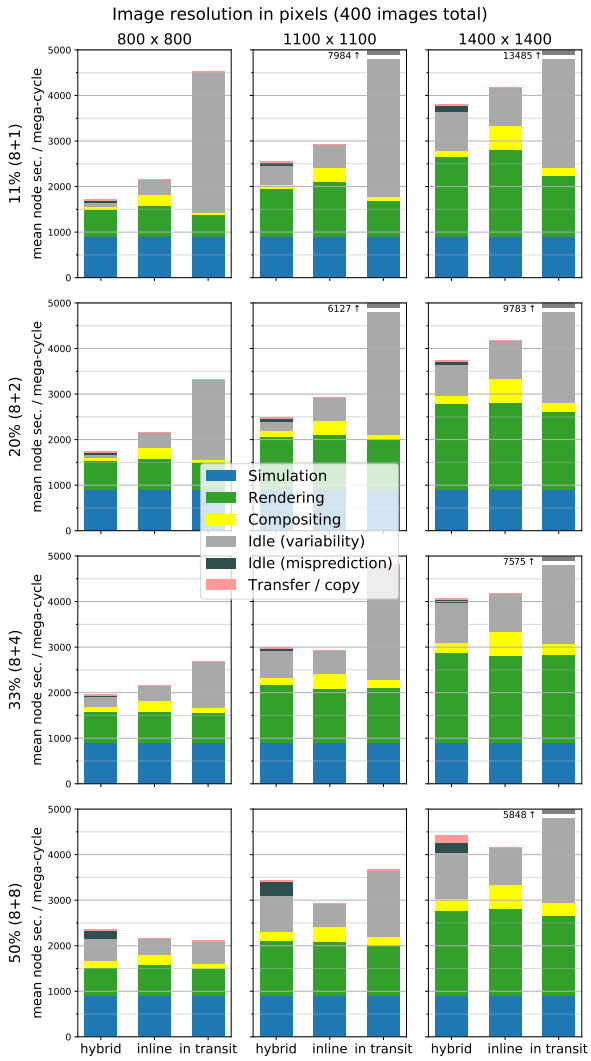


Fig. A.1: The columns in the 3×4 matrix correspond to the work (i.e., image resolution), while the rows reflect the resources (i.e., number of visualization nodes). All configurations run Cloverleaf3D with 8 simulation nodes and generate a Cinema databases. Each of the 12 stacked bar charts compares our hybrid method with inline and in transit. The colors correspond to different activities, and the heights for each color indicate how much time was spent (on average) per mega-cycle. Broken bars indicate higher y-values. The inline configuration does not use visualization nodes, a single inline run is repeated along each column. There are small variations in render times between the techniques for some configurations that we attribute to our use of 6 MPI ranks per physical node and hyperthreading, resulting in slightly different utilization of physical cores.

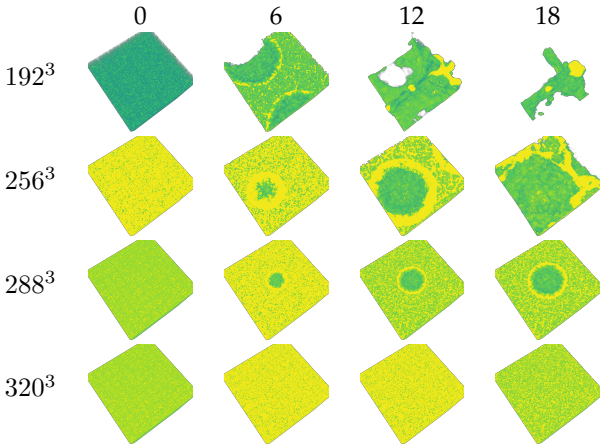


Fig. B.1: Renderings of the Nyx simulation data for selected mega-cycles (columns) and scaling levels (rows). Data heterogeneity between partitions decreases at larger sizes due to slower simulation progress. The same wall-clock time intervals are used to invoke the visualization.

APPENDIX B
RENDERINGS OF NYX SIMULATION DATA

Fig. B.1 shows renderings of the Nyx simulation data for mega-cycles 0, 6, 12, and 18 (columns), for different sizes of our weak-scaling study (Sec. 6.3 of the paper). The data set resolutions (rows) correspond to the runs using 4128, 9888, 14 016, and 19 200 logical cores. All runs call the visualization after a fixed wall-clock time interval of 120 seconds.

As can be seen, the simulation progresses more slowly at larger concurrencies, leading to almost no visible changes in the data for the 320^3 run. This effect causes less variation in the data between partitions and leads to similar, comparably fast render times due to early ray termination and no empty-space skipping. For our hybrid approach, the homogeneity in render times is less favorable, since it leads to less variability that can be "exploited" by our technique. This effect is also reflected in the variability curve in Fig. 9 in the paper.

APPENDIX C
IMPLEMENTATION DETAILS

As a basis for our hybrid in situ approach, we use the *Ascent* [11] framework that supports a wide range of data filters and several parallel scientific rendering algorithms using the *vtk-m* [37] toolkit. Furthermore, Ascent supports distributed memory and comes integrated with a couple of simulation codes. So far, we only use CPUs for both simulation and visualization in our experiments. A potential future extension to GPUs would be possible since they are already supported by vtk-m.

To make our hybrid approach feasible and keep the overhead to a minimum, we parallelized computations and use asynchronous communication wherever possible. MPI is used to distribute tasks among nodes, supporting running multiple MPI tasks per node. Generally, synchronizations are avoided if possible and if they are necessary, they are only applied to the smallest sub-group of ranks. All data sending and receiving operations are asynchronous both on simulation and visualization nodes. An MPI buffer is used for sending the render parts from simulation to visualization nodes.

We use OpenMP for shared memory parallelization of the simulation as well as the rendering tasks on the nodes. Ascent supports the generation of *Cinema* [8] image databases that we use in our approach. In our experiments, we generate volume renderings of the 3D scalar fields generated by the simulations. We perform volume rendering, this is done using an arcball-style camera, i.e., we choose camera positions on a sphere around the data set using pre-defined zoom levels and facing at the center of the dataset. For acceleration, we employ block-based empty space skipping and early ray termination.

For our tests with volume rendering on regular grids, we use the *Cloverleaf3D* [39] simulation, a 3D Lagrangian-Eulerian hydrodynamics benchmark. Ascent comes with a Cloverleaf3D version featuring an integration of the framework. As a second real world example we use *Nyx* [40], a massively parallel code for cosmological hydrodynamics simulations. Nyx uses *AMReX* [41], a software framework for block structured adaptive mesh refinement (AMR).

In the following, we first describe the integration of our hybrid approach into the simulation codes and then into the Ascent framework. Refer to Fig. 2 for an overview of the runtime process in form of a sequence diagram. All frameworks as well as our modifications are open source: Ascent [1], vtk-h [2], Nyx [3], AMReX [4].

C.1 Integration into the Simulation Codes

The integration of our system into the simulation codes is minimally invasive. Besides the integration of the Ascent in situ framework (i.e., mainly coupling of the simulation data), we split the MPI communicator into a simulation and a visualization group based on a user-defined split factor. The MPI ranks that belong to the simulation group proceed regularly with the simulation and start visualization once the insitu condition

1. https://github.com/vbruder/ascent-1
2. https://github.com/vbruder/vtk-h
3. https://github.com/vbruder/Nyx
4. https://github.com/vbruder/amrex-1

triggers, i.e., a user-defined time interval has passed or a number of simulation steps have been processed. We use time intervals to determine the simulation steps we want visualize in our experiments and then use the simulation step number to trigger visualization for a better comparability across runs..

The visualization is started by calling Ascent with the simulation data of the respective rank, the simulation time that we use as an estimate of the time for the next simulation block, and a set of user-defined actions. The latter include the visualization setup: filter pipelines, the scenes to be rendered (e.g., volume rendering including transfer function, Cinema configuration, etc.) and the probing setup for our hybrid approach such as the split factor between simulation and visualization ranks and the amount of renders used for runtime probing.

The visualization ranks wait for incoming data and perform visualization immediately after receiving data from the simulation ranks. In contrast to the simulation ranks, Ascent is called without a data set but the same configuration. Synchronization between the simulation and the visualization ranks only happens inside Ascent during the exchange of probing times.

C.2 Integration into Ascent

We extend the Ascent in situ framework by adding flexible in transit capabilities including a modified compositing that can work solely on visualization resources. Further, we extend Ascent to handle our hybrid approach. This primarily includes a probing step in which we distribute the rendering load across the MPI ranks. At the beginning of processing the visualization in Ascent, we first split the MPI communicator that is passed on by the simulation and includes all ranks into a simulation and a visualization communicator. For this, we use the same split factor as in the simulation code to exactly reproduce the split. A pseudo-random sequence is generated to select the images to be used for probing. Naturally, the same sequence is generated on all ranks.

All simulation ranks then proceed with the probing run, i.e. by rendering the subset of randomly selected images. After rendering, the images' pixel and depth values are encoded using simple run-length encoding. The resulting render times are gathered on all MPI ranks to determine the load distribution and the assignment of the simulation ranks to the visualization ranks. Then, the simulation ranks asynchronously send their simulation data to their assigned visualization ranks, where the respective subset of images is rendered in transit on arrival of the data. Meanwhile, inline rendering and encoding of the remaining images is performed on the simulation ranks. This happens in batches to facilitate asynchronous sending of the render parts to the respective visualization ranks where decoding and compositing happens once all parts are rendered or received. After sending their last render part, the simulation ranks return to compute the next simulation steps.

The visualization ranks receive the image parts of their assigned simulation ranks and use them with the ones rendered themselves for compositing. We use direct send compositing that is integrated in Ascent using the *DIY2* [38] library. Ascent uses direct send since it has to support the worst case scenario, i.e., unstructured meshes that fit together like puzzle pieces. Once the final image is composited, we compress it as a PNG and write it to disk on one of the visualization ranks. We use a producer-consumer approach to write out the compressed PNG images in parallel. After the last image is written to disc, the visualization ranks return to the simulation process and directly proceed with the next visualization cycle.