Performance Modeling of In Situ Rendering

Matthew Larsen*†, Cyrus Harrison†, James Kress*‡, David Pugmire‡, Jeremy S. Meredith‡, Hank Childs*§
*University of Oregon, †Lawrence Livermore National Laboratory
‡Oak Ridge National Laboratory, §Lawrence Berkeley National Laboratory
Email: †{larsen30, cyrush}@llnl.gov,*{jkress, hank}@cs.uoregon.edu,*{pugmire, jsmeredith}@ornl.gov

***Abstract*—With the push to exascale, *in situ* visualization and analysis will continue to play an important role in high performance computing. Tightly coupling *in situ* visualization with simulations constrains resources for both, and these constraints force a complex balance of trade-offs. A performance model that provides an *a priori* answer for the cost of using an *in situ* approach for a given task would assist in managing the trade-offs between simulation and visualization resources. In this work, we present new statistical performance models, based on algorithmic complexity, that accurately predict the run-time cost of a set of representative rendering algorithms, an essential *in situ* visualization task. To train and validate the models, we conduct a performance study of an MPI+X rendering infrastructure used *in situ* with three HPC simulation applications. We then explore feasibility issues using the model for selected *in situ* rendering questions.**

I. Introduction

Visualization on high performance computers has traditionally been accomplished with *post hoc* processing. With this model, a simulation code writes its state to disk at regular intervals (or irregular intervals), and a visualization program reads that data from disk. The dominant factor in overall execution time for most common visualization operations with *post hoc* processing is I/O performance [1]. Further, while I/O bandwidth is increasing on new supercomputers, the ability to generate data is increasing much more quickly. As a result, *post hoc* processing may soon become impractical on leading edge supercomputers.

In situ processing is widely viewed as the best suited paradigm for upcoming supercomputers [2], [3], [4]. With the *in situ* paradigm, visualizations are created while the simulation is running, making it unnecessary to store the simulation's state to disk and read it afterwards. The term *in situ* processing actually represents a family of approaches. With one approach, referred to in this paper as tightly-coupled, the simulation and visualization routines share resources. That is, when it is time to perform visualization tasks, the simulation pauses, hands control to a visualization routine which produces images, and then hands control back to the simulation. This approach is used by both ParaView Catalyst [5] and VisIt's LibSim [6]. With another common approach, referred to in this paper as loosely-coupled, the simulation and visualization routines each have their own dedicated resources, and data is sent between their resources. This approach is also referred to as in transit processing [7] or staging, and is the approach most frequently employed by ADIOS [8].

The requirements for *post hoc* processing and *in situ* processing are different. For *post hoc* processing, the driving requirement was interactivity for human observers. If the user was satisfied with the delivery rate of results, then the performance requirement was met. With *in situ* processing, and especially with tightly-coupled *in situ* processing, the performance requirement is different. Running visualization routines with the tightly-coupled approach means that compute resources are not being devoted to the simulation, and so it slows down the overall simulation time. This trade-off engenders an important feasibility question: is it possible to perform X_1 visualization tasks while devoting no more than X_2 time to these tasks?

Our study considers this feasibility question for parallel rendering. Rendering is often a relatively quick operation (on the order of a second), and so the question is not all that interesting if the desired rendering task is to, for example, render one frame every 500 cycles of the simulation. However, recent research [9], [10], [11] has inspired a new approach, where thousands to millions of renderings are extracted during the simulation, in order to construct an image database that can be explored by domain scientists after the simulation has completed. These images are often of the same geometry (such as an isosurface), but with different rendering parameters, including camera angle and coloring. When considering this number of renderings, the feasibility question we consider becomes very important.

We approached the issue of rendering feasibility through performance modeling. We started by implementing three rendering algorithms: rasterization, ray-tracing, and volume rendering. We implemented them in the VTK-m framework [12], which allows a single implementation to run efficiently over multiple architectures. We then ran 1,350 experiments to collect performance data, varying over rendering algorithm, architecture (CPU and GPU), concurrency, simulation code, and data size. Next we established performance models for each rendering technique using linear regression and evaluated these models using standard regression metrics and k-fold cross validation. We expanded our study to a leading-edge supercomputer (ORNL's Titan) and predicted performance at large scale. Once the performance models were validated, we used them to answer rendering feasibility questions. We believe the result is a novel study, both in our application of performance modeling to visualization and in the results we obtain informing important *in situ* feasibility questions.

SC16; Salt Lake City, Utah, USA; November 2016

II. Related Work

A. Rendering

There are three major types of parallel rendering [13]: sort-first, sort-middle, and sort-last. Sort-first rendering parallelizes over pixels within an image, and redistributes geometry so that every processor has the data necessary to generate its portion of the image. Sort-middle rendering follows a similar approach, but it applies some transforms to the geometry before redistributing. Sort-last rendering parallelizes over data, having each processor create a sub-image that contains a rendering of its own data, and then compositing the sub-images together, i.e., using depth information and parallel communication to create the final image.

Parallel rendering on supercomputers is typically done with the sort-last approach [14]. With this approach, there are two distinct phases: local rendering and compositing. Local rendering requires no parallel coordination, while compositing typically does not begin until all MPI tasks have finished their local rendering.

Each of our rendering algorithms were implemented in VTK-m [12]. VTK-m makes use of data parallel primitives, an abstraction that enables algorithms to be written one time, but used on multiple architectures (CPU and GPU).

We derived our ray-tracer from the implementation in EAVL [15]. An evaluation of the EAVL ray-tracer [16] found that it generally performed within a factor of two of Intel's Embree [17] and NVIDIA's OptiX [18].

Our volume renderer was developed for this study and was designed to work with structured grids. Previous studies of data parallel primitives-based volume renderers [19], [20] have found that the data parallel primitive approach is competitive with hardware-specific versions.

Our rasterizer was also developed for this project. Based on recent successes with rendering and portable performance, we felt that having a single implementation over multiple architectures was better than using architecture-specific rasterizers, such as OpenGL[21], Mesa [22], or Open SWR [23]. We note that our rasterizer is not the first CUDA-based software rasterizer on a GPU [24] (with CUDA being a back-end option for our data parallel primitives).

Although we implemented our own renderers, our study incorporated an existing parallel image compositor, IceT [25]. IceT has been shown to be scalable up to tens of thousands of MPI tasks, and contains implementations of many compositing algorithms, including direct send [26], binary swap [27], and Radix-k [28], [29].

Several previous studies have looked at rendering performance at massive scale [25], [27], [28], [30]. Our study differs from these previous works in that we are considering multiple rendering methods and that we have developed a performance model that allows us to answer feasibility questions.

B. Performance Modeling

Simulators take a fine-grain approach to performance modeling, by attempting to model the behavior of underlying features of the target system, from the microprocessor instruction level to the network messaging level. Examples include SST, ROSS, and GEMS [31], [32], [33], [34]. At the other end of the spectrum, analytical approaches are methods for generating symbolic equations describing an algorithm or program which can quickly generate performance predictions for known input parameters. Examples of analytical models and tools include BSP, LogP, and Aspen [35], [36], [37].

Performance models can use data from either known hardware parameters (e.g., peak floating point or messaging rates) or from empirical measurements such as performance counters and execution time. When the latter are combined with analytical performance models, the derived models are termed "semi-empirical" [38]. Our approach falls into this category, generating an *a priori* analytical performance model based on known attributes of a specific algorithm and fitting with observed data.

C. Performance Modeling and Visualization

There are relatively few rendering works that incorporate performance models. Cohen et al. [39] used an architecture-specific constant to evaluate a cost model, maximizing performance by switching between polygon and point rendering based on an evaluation. Similarly, Tack et al. [40] used a fine-grain performance model to estimates run-time cost per triangle, based on the steps within a software-based rasterization pipeline, on resource constrained mobile devices. Their system ran a set of tests to estimate constants in their model, then adjusted rendering for the given device. While these studies share the elements of performance modeling and rendering, their goals and rendering tasks differ from our own.

Bowman et al. [41], described a performance model for a visualization pipeline, and included rendering in their model. They modeled rendering by multiplying the number of triangles to render by a constant, which was determined from the average time to render different amounts of triangles using a GPU rasterization pipeline. Finally, Rizzi et al. [42] described a distributed GPU ray casting performance model for volume rendering by enumerating each component, including network bandwidth and GPU transfer times, and their goal was to help to provide guidance in the design of future machines. While both were successfully able to model their use cases, our study is different in that we are focused on in situ rendering, which allows us to go beyond their studies in significant ways: by considering multiple rendering techniques, by evaluating varied rendering configurations, by establishing statistical bases for our models, by considering multiple parallel architectures, and by exploring the feasibility regions implied by our models.

III. Methodology

In this section, we introduce the methodology we used to develop the performance models for the three rendering techniques (ray-tracing, rasterization, and volume rendering).

Our goal is to create linear models that relate key variables about data and rendering configurations to the run-time performance of each renderer. We created general linear models

based on the nature of the algorithms used in each renderer. To allow our general linear models to be used for estimates on specific hardware architectures, we ran a study using a range of representative configurations and fit model coefficients to this data using multiple linear regression. We then used several methods to evaluate our models.

A. Model Input Variables

The first modeling step was to select the key input variables of the rendering process. Several factors influence the run-time of rendering, but at a high level, the two essential variables are the number of objects (e.g., cells or triangles) and the number of pixels rendered. For each of the renderers we studied, we started with initial linear models that relate these two variables to the algorithmic complexity of their rendering pipelines.

These models captured the run-times of the different pipeline stages for each type of renderer. While rendering algorithms are generally characterized by the iteration target of their outer loop — object-order algorithms (e.g., rasterization) loop over objects, and image-order algorithms (e.g., ray casting) loop over pixels — this basic characterization overlooks the fact that rendering implementations may actually use pipelines with multiple steps that mix object-order and image-order algorithms. For example, ray-tracing is an image-order algorithm that uses an acceleration structure constructed with an object-order algorithm.

Directly using all of the individual pipeline steps as components in our models could create too many degrees of freedom. To avoid this, we grouped related pipeline steps into higher-level stages and related these stages to the input variables to create the components of our models. For example, our ray-tracing performance models considers the acceleration structure build stage as a single model component, although it actually consists of five smaller steps.

To capture architecture-specific details, we added experimentally-obtained coefficients for each of our linear model components.

We also selected variables that are quantitative proxies to important user settings (e.g. data set size, output image sizes, etc.), which influence rendering performance. These variables include the number of objects in view of the camera, the actual amount of pixels that need to be rendered for the view, and view specific counters for rasterization and volume rendering. The values for these proxies are not known exactly *a priori*, but can be estimated from a user's specific rendering settings. When we fit coefficients to capture architecture-specific details, we used measured data points for these variables to create more accurate models.

In summary, the input variables we used for our models are:

- General Input Variables
  - Objects (O): the number of cells or triangles to render
  - Active Pixels (AP): the number of pixels that are updated as a result of rendering
  - c_i: empirical constants to capture architecture-specific details
- View-Specific Variables for Rasterization
  - Visible Objects (VO): the number of cells or triangles to render visible to the camera
  - Pixels Per Triangle (PPT): the average number of pixels considered per triangle
- View-Specific Variables for Volume Rendering
  - Samples Per Ray (SPR): the average number of samples along a ray that are inside the data set
  - Cells Spanned (CS): the maximum number of cells that a ray can span

B. Study Parameters

To obtain data to fit model coefficients and evaluate our models, we conducted a study to gather run-time performance data for each renderer. We tested our models using a range of configurations that represent *in situ* use cases.

We selected configurations exploring parameters that, from a user's perspective, influence the number of objects and pixels relevant to rendering. These parameters include the type of simulation data, simulation data set size, desired image resolution, and the total number of MPI tasks.

We used multiple physics simulation applications to explore performance with structured and unstructured meshes. We also adjusted the size of the simulations (i.e. the total number of cells in the discretization) to vary the number of objects rendered. We varied the the total number of MPI tasks to change the number of active pixels on each task and to test distributed-memory image compositing run-times.

C. Model Fitting and Evaluation

We used multiple linear regression to fit coefficients that allow us to use our models to estimate rendering run-times for a specific architecture. We used the R programming environment [43] and many of its packages [44], [45], [46], [47], [48] to compute and evaluate the regressions and to create plots.

To evaluate our models, we examined the metrics of multiple R-squared, residual standard deviation, the values of our model coefficients, and the average relative error. Multiple R-squared effectively reports percentage of the variance of the run-time that is captured by a model. The residual standard deviation is an indicator of how well a model fits the data, where a value of zero means the model perfectly fits the data. For rendering algorithms, no input variables should have a negative linear relationship to run-time, and so the presence of regression coefficients less than zero typically indicates that a model is not valid or some feature of the hardware architecture is compensating.

To further evaluate our models, we used correlation analysis and k-fold cross validation. Correlation analysis gave us a basic view of how the input variables are related to the measured run-times. This helped us check our linear modeling assumptions and quickly screen for potential implementation issues. K-fold cross validation systematically fits model coefficients to subsets of the available data and tests how well

these fitted models represent the remaining data. We used k-fold cross validation to check that we were not overfitting our models to data and as another measure of the variance of our models.

IV. Study Overview

A. Software Implementation

We implemented our rendering algorithms using VTK-m [12], which allows a single implementation to execute on multiple architectures. Currently, VTK-m supports two back-ends: TBB and CUDA. VTK-m algorithms are composed of a series of data-parallel primitives such as map, scan (also known as prefix sum), and reduce. Each back-end implements supported data-parallel primitives that are optimized for the architecture, which enables portable performance of algorithms with different back-ends. The majority of the rendering algorithm implementations use a map operation to loop over pixels (e.g., volume rendering and ray tracing) or objects (e.g., rasterization).

For the *in situ* infrastructure, we used Strawman [49], a lightweight *in situ* framework that includes integrations with three physics simulation codes. Strawman uses a modular infrastructure that allowed us to implement a visualization pipeline using the VTK-m rendering infrastructure. For the sort-last rendering, IceT [25] was used to composite the images generated by each task.

B. Study Options

Our study was designed to test and validate the performance models under a wide variety of rendering conditions. In order to examine a representative space of rendering configurations, we varied the following factors:

- Architecture (2 options)
- Rendering Algorithm (3 options)
- Simulation Code (3 options)
- MPI Tasks (up to 7 options, depending on architecture)
- Image Resolution (many options)
- Data Size (many options)

We discuss each of the factors in the following subsections.

1) Architectures: Our study was performed on LLNL's Surface cluster. Each node contains two Intel Xeon E5-2670 and two NVIDIA K40m GPUs. We used the following configurations:

- CPU1: 1 MPI task per node
  - Simulation code: 16 OpenMP threads
  - Strawman: 16 TBB threads
- GPU1: 2 MPI tasks per node
  - Simulation code: 8 OpenMP threads per task
  - Strawman: 1 K40m per task

2) Rendering Algorithms: For the study, we implemented the following algorithms using VTK-m:

- Ray Tracing: a ray tracer that only casts primary rays and generates images similar to rasterization.
- Structured Volume Rendering: a ray caster for uniform and regular grids.
- Rasterization: an implementation based on sampling using barycentric coordinates.

3) Simulation Codes: Our study used the three physics simulation codes included with Strawman. Each of these codes are "proxy apps," meaning they are lightweight and meant to be accessible for computer science research, and figure 1 shows images from each of the simulations. The codes are:

- Lulesh: a Lagrangian shock hydrodynamics simulation on a 3D unstructured mesh
- Kripke: a deterministic neutron transport solver on a 3D structured mesh
- Cloverleaf3D: an Euler hydrodynamics simulation on a 3D structured mesh

4) MPI Tasks: We varied the number of MPI tasks to study the effects of active pixels per task as concurrency increased. The number of MPI tasks for each architecture were:

- CPU1: 1, 2, 4, 8, 16, and 32 tasks
- GPU1: 1, 2, 4, 8, 16, 32, and 64 tasks

5) Image Resolution: There are many possible choices for image resolution, and we wanted our models to predict values for the widest range of possible combinations that are appropriate for tightly-coupled *in situ*. For each test, we used stratified sampling, similar to Latin hypercube sampling, to choose image resolutions between 512^2 and 2880^2, which is equal to the total number of pixels in a standard 4K image (3840×2160). The data sets were cubic, so we chose to use square images to minimize the amount of unused pixels.

6) Data Set Size: As with image resolution, there are many possible choices of data set size, and using stratified sampling, we chose data set sizes ranging from 128^3 to 320^3 cells per node. We consider these sizes to be representative of the per node cell count of a large multi-physics simulation. The problem size per task was weakly scaled, so that the total number of cells over all processors increased proportionally to the number of MPI tasks. For the rasterizer and ray tracer, we used an external faces operation to generate triangles on each MPI task, and all tasks were assigned equal amounts of geometry. We used external faces to control the amount of geometry, but we believe it is representative of other visualization operations such as slicing and contouring, which takes $O(N^3)$ cells and creates $O(N^2)$ geometry.

C. Tests and Measurements

For the study, we ran the cross product of the study variables with 25 combinations on data set size and image resolution, and while not all combinations made sense (e.g., structured volume renderer with an unstructured data set), we ran 1350 total tests. Each test consisted of running the simulation code for 10 cycles, and we generated an image for each cycle. Once all of the data were gathered, we discarded the first data point because it proved to be unreliable. We gather data for each frame and for each rank. Since rendering is only as fast as the slowest MPI task, we only considered the slowest MPI task

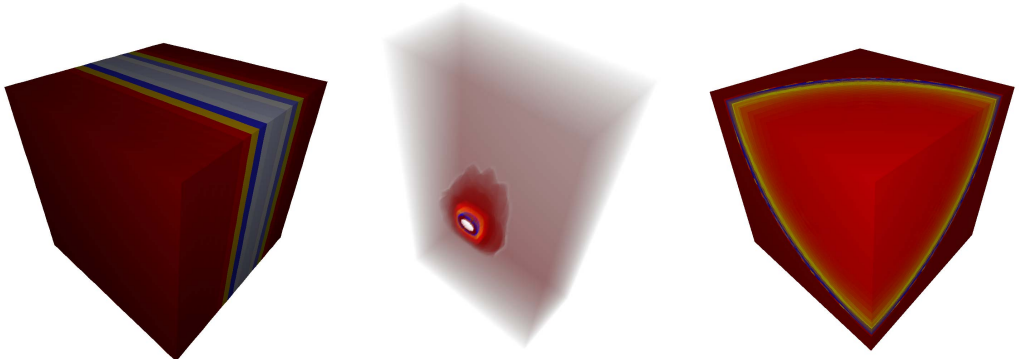

Fig. 1. Images from each of the simulation codes. From left to right, a rasterized image from Cloverleaf3D, a volume rendering from Kripke, and a ray-traced image from Lulesh.

from each of the 1350 tests. Further, for this slowest MPI task, we took its average time over the last nine cycles.

Finally, although we wanted to study rendering with GPUs, the simulation codes we studied only ran on CPUs. Our solution was to run the simulation code on the CPU, and then transfer its data to the GPU. Our measurements in this case only reflect the rendering times (i.e., after the data was transferred), which we feel would be representative of scenarios where simulations run directly on the GPU. While we did not consider it in our study, we feel that data transfer time could be added to our models without much effort.

V. Single Node Performance Model

Section V-A defines the models we used for each rendering type, while Section V-B describes our results evaluating our models' accuracy.

A. Model Definition

We describe our performance models for each of the three rendering techniques. Each of the three models use input variables that describe the rendering task to predict the corresponding execution time. The models described in this section are the final models we developed for each rendering technique, realized after exploring many alternative options. They represent the best fit to the data from our experiments, evaluated using methodology discussed in Section III.

The models describe only the time to do local rendering (i.e., a single MPI task rendering its own data); this model is extended in Section VI to include parallel rendering. When running in parallel (i.e., more than one MPI task), each node has its own value for each of the input variables. For this phase of our model, we are only concerned with the process of making a sub-image (which is embarrassingly parallel) and not with the compositing (which does require parallel communication). So our model can predict the individual execution times for each of the MPI tasks, given each of their individual input variables.

1) Ray-Tracing: Our performance model for ray-tracing execution time (T) is:

$$T_{RT} = (c_0 * O + c_1) + (c_2 * (AP * log_2(O)) + c_3 * AP + c_4) \quad (1)$$

The intuition behind this model is that ray-tracing consists of three parts: building the acceleration structure ($c_0 * O$), tracing the rays ($c_1 * (AP * log_2(O))$), and shading the intersection points ($c_2 * AP$). We timed each of these three phases separately, which enabled us to solve for constants c_i for each of the three separate linear regression models. Finally, we separated the term first term (i.e.,$c_0 * O + c_1$), from the rest of the terms. This allows us to consider cases where the BVH acceleration structure is built once, and then multiple renderings use that BVH afterwards.

There are multiple ways to implement ray-tracing, and our model matches our particular implementation. Specifically, our acceleration structure is a variant of a Linear Bounding Volume Hierarchy (LBVH) [50], which has a build-time complexity of $O(n)$. Tracing rays is a function of the number of pixels (AP), and traversal through the acceleration structure, which is a binary tree, is a function of the number of objects (i.e., $log_2(O)$). It is worth noting that the shape of the binary tree and average traversal depth is related to the spatial distribution of the objects and camera position. Finally, the shading term is a function of the number of pixels.

2) Rasterization: Our performance model for rasterization execution time is:

$$T_{RAST} = c_0 * O + c_1 * (VO * PPT) + c_2 \quad (2)$$

The intuition behind this model is that rasterization consists of two parts: culling objects that are not visible ($c_0 * O$), and then rasterizing the visible objects ($c_1 * (VO * PPT)$).

3) Volume Rendering: Our performance model for structured volume rending is:

$$T_{VR} = c_0 * (AP * CS) + c_1 * (AP * SPR) + c_2 \quad (3)$$

The four main tasks in volume rending are locating cells that contain sample points, loading scalar values into memory, interpolating scalars, and compositing colors. Our implementation consists of a single component that samples by casting rays into the volume, but the four tasks fall into two distinct groupings: cell frequency ($AP * CS$) calculations and sample frequency ($AP * SPR$) calculations. The performance model groups the calculations into terms of the same frequency.

Cell frequency calculations involve locating scalar values for a cell and loading them into memory. The total amount of cell locations is the number of active pixels (AP) multiplied by the number of cells that span the ray's path (CS) (i.e.,

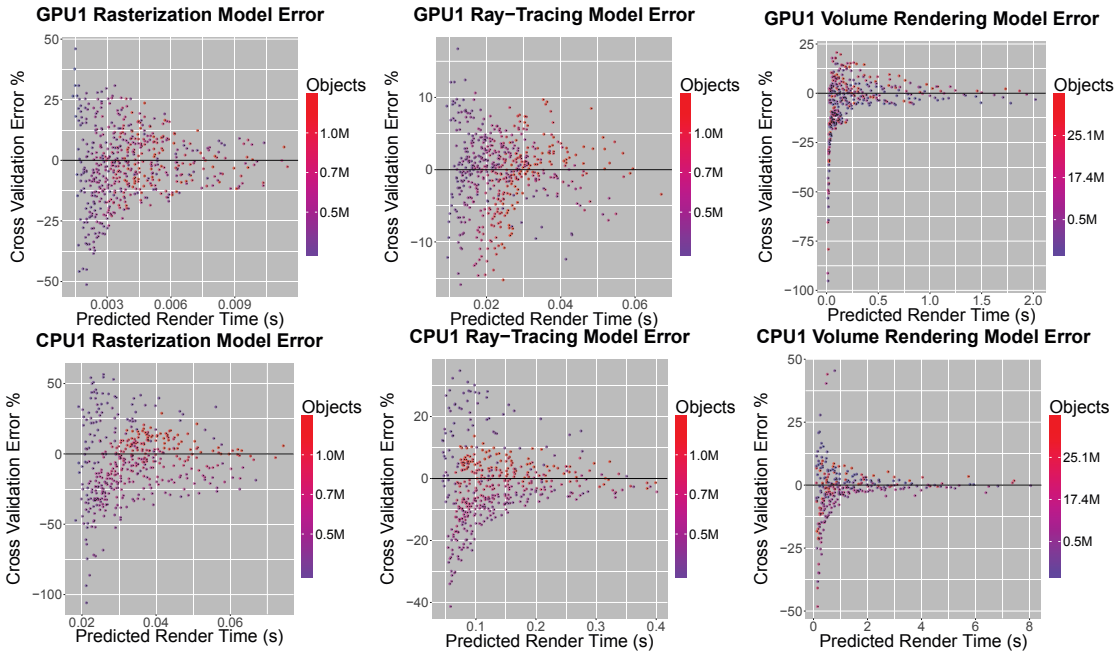

Fig. 2. The 3-fold cross-validation error plots for all six models. Renders are paired in columns, while architectures go along rows (GPU1 on top, CPU1 on bottom). For each fold, the models are trained with a partial set of the data, then the models predict the total render time with the remaining data points. Error percentage is calculated as $100 \times (time_{total} - time_{predicted})/time_{total}$. A value of zero represents a perfect prediction. Each test is colored by the number of objects it renders (i.e., triangles for rasterization and ray-tracing and cells for volume rendering).

TABLE I
R^2 VALUES FOR OUR PERFORMANCE MODELS. R^2 CAPTURES THE PERCENTAGE OF VARIANCE IN RUN-TIME CAPTURED BY THE MODEL AND IS AN INDICATOR OF MODEL FIT.

	R^2	
Renderer	CPU1	GPU1
Ray Tracing	0.9695	0.9728
Volume	0.9981	0.9978
Rasterization	0.6677	0.9425

$(AP * CS)$). Additionally, our implementation calculates interpolation constants that are re-used when sampling the same cell multiple times.

Sample frequency calculations consist of interpolation and compositing. This happens exactly $AP * SPR$ times, which is equal to the total number of samples for the render.

B. Model Evaluation

Table I shows the R^2 values for the six architecture-rendering pairings. In our case, five of the six models had values over 0.94, which indicates a good fit.

Our 3-fold cross-validation study involved creating a model using a subset of the data, and then evaluating the model with the remainder of the data to prevent overfitting. The results of this study are plotted in Figure 2, which shows the error for each test, and is summarized in Table II, which shows model accuracy over all tests. The model errors in Figure 2 shows that we are increasingly accurate as render time goes up. These errors are also seen in Table II, which shows the average error percentage for our models were at worst 19.1% incorrect on average, and at best 3.8% incorrect on average. In terms of model accuracy, Table II shows that all six models were predictive within 50% error at a very high rate — the worst model was still able to predict within 50% error for 96% of the time, and for three of the models, we were within 50% error all of the time.

The model that had the poorest fit was CPU1+rasterization. This reflects the high variability in observed run-time in its tests. Our reported values were an average of nine cycles; for every one of our CPU1+rasterization experiments, at least one of the nine tests was more than 10% above or below the test average.

That said, despite this low R^2 score, 70% of the predictions were within 25% of the measured values and 96% were within 50%.

The x-axis for Figure 2 is predicted render time, and it is clearly visible that our tests skew toward shorter run times. This does not represent bias in our tests, but rather the effects of increased concurrency, which then reduces the number of active pixels per MPI task. For all models, it is these lower render times that contribute the majority of the error occurs, where both timer resolution and memory hierarchy factors more heavily influence execution time.

One clear example of the influence of the memory hierarchy is the volume rendering model for GPU1. As render times approach zero, our model consistently over estimates the cost in a small part of the distribution. The majority of the data points have more active pixels when sampling the volume saturates GPU memory throughput, and the model coefficients reflect the majority of the data points. With a small number of active pixels, GPU threads do not have to wait long for

TABLE II

MODEL ACCURACY SUMMARY FROM THE RESULT OF THE 3-FOLD CROSS VALIDATION ANALYSIS FOR OUR SIX MODELS (THREE RENDERING TECHNIQUES TIMES TWO ARCHITECTURES). FOR EACH FOLD, TWO THIRDS OF THE DATA IS USED TO TRAIN THE MODEL AND THE REMAINING ONE THIRD IS USED TO TEST THE PREDICTION. EACH PERCENTAGE REPRESENTS THE NUMBER OF PREDICTED VALUES, FROM ALL FOLDS, THAT ARE WITHIN THE CORRESPONDING ERROR PERCENTAGE.

	CPU1					GPU1				
Renderer	50%	25%	10%	5%	Average %	50%	25%	10%	5%	Average %
Ray Tracing	100.0	93.3	65.6	35.6	9.6	100.0	100.0	94.8	71.7	3.8
Volume	100.0	95.4	81.3	63.0	6.3	97.8	92.8	64.7	46.0	10.0
Rasterization	96.0	70.2	35.8	18.4	19.1	99.8	90.2	53.2	26.8	11.5

memory requests to be satisfied. The effects of the memory hierarchy on both the CPU and GPU are complex and occur at too fine-grain a level to be useful to our models.

C. Memory Usage

In addition to timing rendering, we captured memory usage information during our experiments. Our data set dimensions and image sizes for individual tests were essentially random, so we could not directly compare memory usage statistics. On the whole, rasterization used the least amount of memory, and ray tracing used the most since it uses an acceleration structure. On average, rendering increased the simulation memory overhead by less than 1%, and peaked at 5% when image size was large and data set size was small.

VI. MULTI-NODE PERFORMANCE MODEL

In a distributed memory setting, there are two factors that contribute to the overall rendering time: single node rendering time and image compositing time. The first factor is the maximum run-time out of the set of all MPI tasks, and we can predict the bottleneck task using the performance models from section V. The second factor is the performance of compositing in which the images produced by each task are merged into a final image based on depth (e.g., rasterization and ray-tracing) or visibility ordering (volumes). In equation form, our total rendering model is:

$$T_{total} = max_{tasks}(T_{LR}) + T_{COMP} \quad (4)$$

where T_{LR} is the time for local rendering, i.e. T_{RT}, T_{VR}, or T_{RAST} based on the rendering technique used. Further, although it is not the primary focus of this paper, we need a basic model for image compositing (T_{COMP}) to answer questions about distributed rendering performance. Establishing T_{COMP} is the focus of the remainder of this section.

A. Observations

Figure 3 shows a histogram of compositing performance in our study as a function of MPI Rank and Pixels. The dominant trend in the histogram is that more pixels lead to slower times. The secondary trend is counterintuitive, which is that more tasks make compositing faster. This is because increasing MPI tasks causes the simulation domain to be divided up among more tasks, reducing the number of active pixels. This trend is likely to reverse as MPI tasks increase further, since the active pixels will decrease less quickly, but the amount of communication will increase. Unfortunately, our experimental study was limited in scope with respect to MPI tasks, and we were not able to capture these effects from our corpus of 1,350 experiments. A more thorough performance analysis at massive scales can be found in [25] and [29].

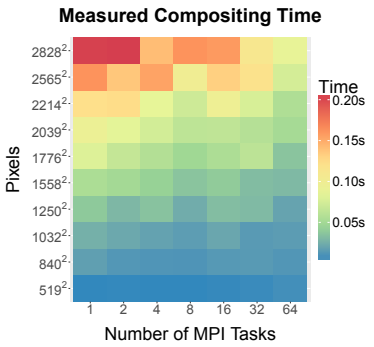

Fig. 3. Histogram of image compositing time with respect to number of MPI Tasks and Pixels.

B. Model Definition

Our performance model for compositing is:

$$T_{COMP} = c_0 * ave_{tasks}(AP) + c_1 * Pixels + c_2 \quad (5)$$

where Pixels is the (full) image resolution.

Image compositing proceeds in a hierarchy of rounds exchanging pixel information, and the main factors are the number of pixels in the image and the number of MPI tasks. Our model uses image resolution for the number of pixels, and average number of active pixels captures the amount of work performed, since the active pixels is related to the number of cores and pixels that can contribute to the final image. As noted in Section VI-A, large numbers of MPI tasks will slow down compositing. We did not include that term in our model, since our regression analysis latched onto the decrease in execution time as MPI tasks went up (because of the decrease in active pixels) — adding that input variable using our corpus led to a model where increasing MPI tasks led to faster and faster compositing times. Again, while we believe we have the correct corpus for our single-node performance model, we understand our corpus is too limited for performance modeling of compositing.

C. Model Evaluation

The performance of our model is summarized in Figure 4, which shows the results of the cross-validation analysis, and table III, which shows the percentage of predictions that fall within percentiles. The performance model underestimates the cost of compositing images of low resolution, but the model is

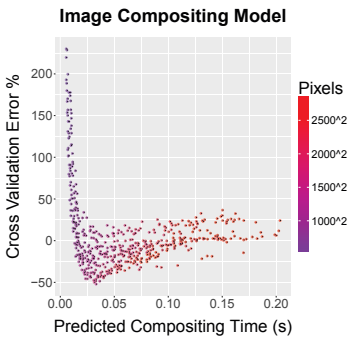

Fig. 4. 3-fold cross-validation error plots for compositing. For each fold, the models are trained with a partial set of the data, then the models predict the total render time with the remaining data points. Error percentage is calculated as $100 \times (time_{total} - time_{predicted})/time_{total}$. A value of zero represents a perfect prediction. Each test is colored by the number of pixels composited.

TABLE III
MODEL ACCURACY SUMMARY FROM THE RESULT OF THE 3-FOLD CROSS VALIDATION ANALYSIS FOR OUR COMPOSITING MODEL. FOR EACH FOLD, TWO THIRDS OF THE DATA IS USED TO TRAIN THE MODEL AND THE REMAINING ONE THIRD IS USED TO TEST THE PREDICTION. EACH PERCENTAGE REPRESENTS THE NUMBER OF PREDICTED VALUES, FROM ALL FOLDS, THAT ARE WITHIN THE CORRESPONDING ERROR PERCENTAGE.

	50%	25%	10%	5%	Average %
Compositing	88.4	68.3	28.3	15.0	29.3

fairly accurate for higher resolutions to provide a reasonable estimate (within the range of concurrencies we consider).

VII. EVALUATION ON LEADING-EDGE SUPERCOMPUTER

To further validate our models, we tested on a leading edge supercomputer, ORNL's Titan machine. Each node contains a 16-core AMD processor with 32 GB of memory, and we used the following configuration:

- GPU2: 1 MPI task per node
  - Simulation Code: 16 OpenMP threads
  - Strawman: 1 K20 GPU per task

The architecture-specific coefficients (c_i) changed on this machine, so we ran a small number of experiments to calibrate.

We used Cloverleaf3D as the simulation code, and ran between 20 and 31 experiments for each renderer, using the same methods as described in Section IV. Next, we performed a large-scale run, using 1024 nodes of GPU2 for each renderer with an image resolution of 2048^2 and over 16 billion total elements, and tested the models predictions against the measured run-time.

Table IV shows the results of the runs including actual render time, predicted render time, percentage error, and number of samples used to train the model for each renderer. For ray-tracing and rasterization, our render time predictions were within 6% and 18% of the actual runtime, which was in line with our expectations. The prediction for volume rendering was 78% away from the actual run-time, and this matched the error we saw in the upper-right of figure 2 as the renderer time approached zero.

As discussed in Section VI-A, our compositing model is not appropriate at the scale of 1024 MPI tasks, so we do not present it here.

TABLE IV
WE TRAINED OUR MODELS USING A MINIMAL NUMBER OF DATA POINTS USING CLOVERLEAF3D AND BETWEEN 1 AND 32 MPI TASKS ON TITAN. WE USED THE SAME SAMPLING STRATEGY AS THE FIRST PHASE OF OUR STUDY. WE THEN INCREASED THE NUMBER OF MPI TASKS TO 1024 WITH 16 BILLION TOTAL CELLS AT AN IMAGE RESOLUTION OF 2048^2, AND TESTED THE RUN-TIME AGAINST THE MODEL PREDICTIONS.

Rendering Technique	Actual Time	Predicted Time	Difference	Sample Points
Ray-Tracing	0.0165s	0.0176s	-6.0%	31
Volume Render	0.00638s	0.0136s	-78.6%	30
Rasterization	0.00275s	0.00224s	18.5%	20

VIII. MAPPING RENDERING CONFIGURATIONS TO PERFORMANCE MODEL VARIABLE INPUTS

Visualization experts and domain scientists likely think of rendering configurations in terms of our study options from Section IV-B (i.e., architecture, rendering technique, number of MPI tasks, image resolution, and data set), and it is doubtful that they would immediately think of their rendering problem in terms of our variable inputs (i.e., O, AP, VO, PPT, SPR, CS, and c_i). To bridge this gap, a mapping from rendering configurations to variable inputs is needed. In general, an exact mapping can be difficult to generate, especially because some variable inputs can be data-specific. For example, the number of triangles in an isosurface is not known until you apply an isosurfacing algorithm. That said, it is typically easy to provide upper bounds for the terms, which in turn can be fed into our model, even for arbitrary rendering configurations. Further, since all coefficients c_i are positive, as the upper bound becomes larger and larger, then our prediction will get larger and larger. This is a desirable property, since it means that overestimates lead to conservative results.

For our study, we use the following mappings:

- Objects (O): varies based on rendering type
  - For rasterization and ray-tracing and a data set of size N^3, this is $12 \times N^2$, since we are taking external faces (each of the six exterior faces of the domain has N^2 quadrilaterals, which is two triangles).
  - For volume rendering, this is N^3.
- Active Pixels (AP): $55\% \times \frac{1}{\#MPITasks^{1/3}} \times Pixels$
  Our camera positions filled about 60% of pixels by default. Each time the number of blocks went up by N^3, the number of active pixels for a rank dropped to be approximately $\frac{1}{N^{th}}$ as much. So, with 8 blocks, each MPI task would have about 30% of all available pixels as active.
- Visible Objects (VO): min(AP, O)
  If there are fewer triangles than active pixels, then all triangles are visible. Otherwise, the number of visible objects is approximately the same as the number of active pixels.
- Pixels Per Triangle (PPT): $AP \times 4$
  For our external faces, active pixels on average have two overlapping triangles (front face and back face). Further, an additional two triangles (the triangles that "complete

TABLE V

THIS TABLE CONTAINS FIVE GROUPINGS OF INFORMATION THAT COLLECTIVELY PROVIDE EVIDENCE THAT OUR MAPPING FROM RENDERING CONFIGURATION TO VARIABLE INPUTS IS VALID. THE FIRST GROUPING DESCRIBES SIX RANDOMLY CHOSEN TEST CONFIGURATIONS FROM THE STUDY, ONE FOR EACH OF OUR PERFORMANCE MODELS. THE MIDDLE THREE GROUPINGS SHOW THE VALUES FOR VARIABLE INPUTS IN OUR PERFORMANCE MODELS FOR THE THREE RENDERERS. THE FINAL GROUPING SHOWS THE EXECUTION TIME FROM A PREDICTION USING OUR MAPPING (LABELED MAPPING), FROM A PREDICTION USING OUR OBSERVED VARIABLE INPUTS FROM AN EXPERIMENT (LABELED EXPERIMENT), AND THE ACTUAL EXECUTION TIME FROM THE TESTS (LABELED ACTUAL).

Test #	Arch.	Rendering Technique	Mesh Size	Image Size	MPI Tasks
0	CPU	Volume Rendering	206^3	2375^2	4
1	CPU	Ray-Tracing	224^3	1598^2	32
2	CPU	Rasterization	185^3	1712^2	8
3	GPU	Volume Rendering	226^3	2322^2	2
4	GPU	Ray-Tracing	155^3	1688^2	64
5	GPU	Rasterization	177^3	2838^2	16

Test (Description)	Type	Active Pixels	Samples Per Ray	Cells Spanned
0 (CPU / Vol.Ren.)	Predicted	1.96M	235	206
0 (CPU / Vol.Ren.)	Observed	1.84M	217	206
3 (GPU / Vol.Ren.)	Predicted	2.35M	296	262
3 (GPU / Vol.Ren.)	Observed	2.24M	281	262

Test (Description)	Type	Objects	Active Pixels
1 (CPU / Ray-Tracing)	Predicted	442K	339K
1 (CPU / Ray-Tracing)	Observed	602K	338K
4 (GPU / Ray-Tracing)	Predicted	228K	232K
4 (GPU / Ray-Tracing)	Observed	288K	232K

Test (Description)	Type	Objects	Visible Objects	Pixels Per Triangle
2 (CPU / Rasterization)	Predicted	406K	406K	7.94
2 (CPU / Rasterization)	Observed	410K	408K	7.1
5 (GPU / Rasterization)	Predicted	371K	371K	18.9
5 (GPU / Rasterization)	Observed	376K	375K	18.0

Test #	Mapping	Experiment	Actual
0	2.47s	2.075s	2.02s
1	0.14s	0.10s	0.08s
2	0.009s	0.006s	0.006s
3	0.97s	0.88s	0.89s
4	0.07s	0.05s	0.05s
5	0.14s	0.13s	0.13s

the quadrilateral" with the original two triangles) will still consider these pixels, but will fail their inside-out tests.

- Samples Per Ray (SPR): $373 \times \frac{1}{\#MPITasks^{1/3}}$
  Like Active Pixels, we found the baseline value for a single core, and then considered behavior as the number of MPI tasks increased.
- Cells Spanned (CS): N (from an N^3 mesh)
  The maximum value is the number of cells that span the diagonal of the data set and the minimum is zero. We found N to be a good estimate.

Again, the mappings above were appropriate for our study (and we validated them by checking against experimental results), but would have to be re-thought for other rendering configurations. The purpose for showing them was (1) to provide background for our next section exploring rendering feasibility questions and (2) to demonstrate an example mapping for those considering other rendering configurations.

TABLE VI

EXPERIMENTALLY-DETERMINED COEFFICIENTS FOR OUR MODELS.

Technique	Arch	c_0	c_1	c_2	c_3	c_4
Ray-tracing	CPU1	5.39e-8	1.13e-2	1.84e-9	3.46e-8	1.28e-2
Ray-tracing	GPU1	1.32e-8	3.81e-3	3.63e-10	2.13e-9	3.54e-3
Rasterization	CPU1	1.29e-8	1.97e-9	1.75e-2	–	–
Rasterization	GPU1	2.10e-9	3.73e-10	1.36e-3	–	–
Volume	CPU1	3.68e-10	4.46e-9	9.35e-2	–	–
Volume	GPU1	1.43e-10	1.07e-9	9.35e-3	–	–
Compositing	–	1.89-e8	4.73e-5	-3.25e-2	–	–

A. Validation of Mapping

To validate our mapping procedure, we considered six rendering configurations. For each configuration, we applied our mapping to obtain variable inputs, and then used our performance model. For those same six rendering configurations, we also retrieved the observed variable inputs from the experiments and used those variable inputs to predict execution time with the performance model. Finally, we also compared with the observed execution time for each rendering configuration. The results of this validation are in Table V. It shows that our mapping does a reasonable job of predicting variable inputs (for our randomly selected test cases), and also that our conservative estimates do lead to slower prediction times in some cases.

IX. IN SITU VIABILITY

In this section, we use our performance models to ask rendering feasibility questions. We utilize the experimentally-determined coefficients from Sections V and VI in our model, and list those coefficients in Table VI. We also make use of the mapping from Section VIII so that we can pose our questions in terms of rendering configurations (as opposed to our performance model's input variables).

We focus on two specific rendering feasibility questions here, to establish the benefit of our performance model and also to provide a demonstration of its use. However, we note that our model can be used to answer many more types of questions.

A. Maximizing Images Rendered in a Fixed Time Allotment

As mentioned in the introduction, a new image-based paradigm is emerging for *in situ* processing [9], [10], [11]. The researchers behind these works are advocating for extracting thousands to millions of images, the rendering of which could significantly impact the overall execution time of a simulation code in a tightly-coupled setting. While we could explore many issues in this context, we consider the following: what is the tradeoff between image size and the total number of images produced? While we cannot know whether a domain scientist prefer to produce many small images or few large ones, we can enable that scientist to make an informed decision.

We approached our example problem as follows: We assumed a fixed configuration for data and number of MPI tasks, since these factors would typically be fixed when running tightly-coupled *in situ*. For our example, we assumed 32 MPI tasks, each with a data set that is 200^3. We also assumed a budget of sixty seconds from the simulation code to perform

Fig. 5. Predictions from our performance model on how many images each rendering technique can generate in 60 seconds, as a function of image size. This analysis could allow a domain scientist to make informed tradeoffs between quality (high image resolutions) and quantity (number of viewpoints).

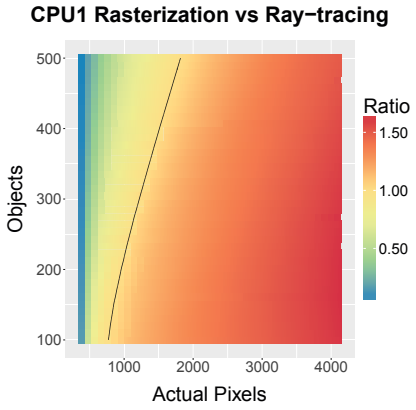

Fig. 6. Heatmap comparing ray-tracing and rasterization performance. A value of 1.5 means that rasterization is faster, and can produced three images for every two ray-tracings. A value of 0.5 means two ray-tracings can be rendered for every rasterization. While rasterization is faster in a larger proportion of the configurations, ray-tracing has much more significant advantages in the configurations where it is faster.

our visualizations. We then considered all six performance models (two architectures times three rendering techniques) and ran our performance model with images sizes varying from 1024^2 to 4096^2, in steps of 128 in both dimensions. For each image size, we got a predicted rendering time, which we could then use to calculate the number of frames per minute. For example, if a single rendering takes 0.2s, then we could do 300 renderings in one minute. The results are listed in Figure 5.

B. Which is Better?: Ray-Tracing and Rasterization

While rasterization has traditionally been the dominant paradigm in scientific visualization for rendering surface data, researchers have recently been considering if ray-tracing may be superior [51]. The intuition for ray-tracing is that it can perform well with large data sets, since its work is proportional to the number of pixels. On the downside, ray-tracing works best when accelerated by efficient spatial search structures (like the LBVH), and these structures take time to build. However, in repeated rendering use cases (like the ones described in Section IX-A), the build time is amortized, since it only needs to be done before the first render.

With our performance model, we can consider the ray-tracing versus rasterization question. We supposed an example setting where there were 32 MPI tasks and we wanted to perform 100 renderings. We then varied the image size from 384^2 to 4096^2 in steps of 128 in both dimensions and considered data sizes from 100^3 to 500^3 in steps of 25^3 in all three dimensions. For each of the results 510 configurations, we ran our performance model for both rendering types and considered the ratio between their predicted execution times. Figure 6 plots the results. As suspected, ray-tracing does have a significant advantage when there is a lot of geometry and few pixels. In the most extreme case — 384^2 images and 500^3 data — ray-tracing could produce 16 images for every one rasterization. However, rasterization gained regular advantages when images went above 1024^2, and was a consistent winner by 1920^2. That said, its biggest advantages, when image size was large and geometry was small, was nowhere near as lopsided as ray-tracing's: the top advantage for rasterization over all configurations we considered was to render three images for every two ray-tracings.

X. Conclusion

We established performance models for three rendering methods in order to explore feasibility questions with *in situ* rendering. These feasibility questions will be increasingly important as *in situ* processing becomes more and more common, leading simulation codes and visualization programs to compete for resources. Our models were validated through statistical analyses and shown to have good fit overall. This in turn allowed us to explore a sampling of interesting feasibility questions, which in turn demonstrates the utility of our models. In terms of future work, we would like to explore additional rendering considerations, including the impact of camera angle, varying geometry per MPI task, and compositing at higher scales.

Acknowledgment

Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Hank Childs is grateful for support from the DOE Early Career Award, Contract No. DE-SC0010652, Program Manager Lucy Nowell. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Finally, we thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. Weber, and E. W. Bethel, "Extreme Scaling of Production Visualization Software on Diverse Architectures," *IEEE Computer Graphics and Applications (CG&A)*, vol. 30, no. 3, pp. 22–31, May/Jun. 2010.

[2] K.-L. Ma, "In situ visualization at extreme scale: Challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, pp. 14–19, 2009.

[3] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, Valerio Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. I. Joy, Q. Koziol, J. Lofstead, J. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. J. Wu, "Scientific Discovery at the Exascale: Report for the DOE ASCR Workshop on Exascale Data Management, Analysis, and Visualization," July 2011.

[4] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E. W. Bethel, "Research Challenges for Visualization Software," *IEEE Computer*, vol. 46, no. 5, pp. 34–42, May 2013.

[5] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jan, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 2011, pp. 89–96.

[6] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*. Eurographics Association, 2011, pp. 101–109.

[7] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld *et al.*, "Examples of in transit visualization," in *Proceedings of the 2nd international workshop on Petascale data analytics: challenges and opportunities*. ACM, 2011, pp. 1–6.

[8] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[9] A. Kageyama and T. Yamada, "An approach to exascale visualization: Interactive viewing of in-situ visualization," *Computer Physics Communications*, vol. 185, no. 1, pp. 79–85, 2014.

[10] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 424–434.

[11] J. Ahrens, J. Patchett, A. Bauer, S. Jourdain, D. H. Rogers, M. Petersen, B. Boeckel, P. OLeary, P. Fasel, and F. Samsel, "In situ mpas-ocean image-based visualization," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*, 2014.

[12] K. Moreland, C. Sewell, W. Usher, L. ta Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications (CG&A)*, May/Jun. 2016, (to appear).

[13] T. W. Crockett, "An introduction to parallel rendering," *Parallel Computing*, vol. 23, no. 7, pp. 819–843, 1997.

[14] C. Hansen, E. W. Bethel, T. Ize, and C. Brownlee, "Rendering," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, Oct. 2012, pp. 49–70.

[15] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros, "EAVL: the extreme-scale analysis and visualization library," in *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012, pp. 21–30.

[16] M. Larsen, J. Meredith, P. Navrátil, and H. Childs, "Ray-Tracing Within a Data Parallel Framework," in *Proceedings of the IEEE Pacific Visualization Symposium*, Hangzhou, China, Apr. 2015, pp. 279–286.

[17] S. Woop, L. Feng, I. Wald, and C. Benthin, "Embree ray tracing kernels for cpus and the xeon phi architecture," in *ACM SIGGRAPH 2013 Talks*. ACM, 2013, p. 44.

[18] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison *et al.*, "Optix: a general purpose ray tracing engine," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 66, 2010.

[19] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs, "Volume Rendering Via Data-Parallel Primitives," in *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, Cagliari, Italy, May 2015, pp. 53–62.

[20] H. A. Schroots and K.-L. Ma, "Volume rendering with data parallel visualization frameworks for emerging high performance computing architectures," in *SIGGRAPH Asia 2015 Visualization in High Performance Computing*. ACM, 2015, p. 3.

[21] "The industry's foundation for high performance graphics," http://www.opengl.org/, accessed: 2016-4-01.

[22] "The mesa 3d graphics library," http://www.mesa3d.org/, accessed: 2016-4-01.

[23] "A high performance, highly scalable software rasterizer for opengl," http://http://openswr.org/, accessed: 2016-4-01.

[24] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 2011, pp. 79–88.

[25] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An image compositing solution at scale," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 25.

[26] U. Neumann, "Parallel volume-rendering algorithm performance on mesh-connected multicomputers," in *Parallel Rendering Symposium, 1993*. IEEE, 1993, pp. 97–104.

[27] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 59–68, 1994.

[28] T. Peterka, H. Yu, R. B. Ross, K.-L. Ma *et al.*, "Parallel volume rendering on the ibm blue gene/p." in *EGPGV*. Citeseer, 2008, pp. 73–80.

[29] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur, "A configurable algorithm for parallel image-compositing applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 4.

[30] M. Howison, E. W. Bethel, and H. Childs, "Hybrid parallelism for volume rendering on large-, multi-, and many-core systems," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 17–29, 2012.

[31] A. F. Rodrigues, K. S. Hemmert *et al.*, "The structural simulation toolkit," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.

[32] C. L. Janssen, H. Adalsteinsson *et al.*, "Using simulation to design extremescale applications and architectures: Programming model exploration," *SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 4–8, 2011.

[33] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low-memory, modular time warp system," *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[34] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[35] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[36] D. Culler, R. Karp *et al.*, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93, 1993, pp. 1–12.

[37] K. Spafford and J. Vetter, "Aspen: a domain specific language for performance modeling," in *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2012.

[38] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:12. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063356

[39] J. D. Cohen, D. G. Aliaga, and W. Zhang, "Hybrid simplification: combining multi-resolution polygon and point rendering," in *Proceedings of the conference on Visualization'01*. IEEE Computer Society, 2001, pp. 37–44.

[40] N. Tack, F. Morán, G. Lafruit, and R. Lauwereins, "3d graphics rendering time modeling and control for mobile terminals," in *Proceedings of*

the ninth international conference on 3D Web technology. ACM, 2004, pp. 109–117.

[41] I. Bowman, J. Shalf, K.-L. Ma, and W. Bethel, "Performance modeling for 3d visualization in a heterogeneous computing environment," *Lawrence Berkeley National Laboratory*, 2004.

[42] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath, "Performance Modeling of vl3 Volume Rendering on GPU-Based Clusters," in *Eurographics Symposium on Parallel Graphics and Visualization*, M. Amor and M. Hadwiger, Eds. The Eurographics Association, 2014.

[43] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014. [Online]. Available: http://www.R-project.org/

[44] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. [Online]. Available: http://ggplot2.org

[45] H. Akima and A. Gebhardt, *akima: Interpolation of Irregularly and Regularly Spaced Data*, 2015, r package version 0.5-12. [Online]. Available: http://CRAN.R-project.org/package=akima

[46] T. Wei, *corrplot: Visualization of a correlation matrix*, 2013, r package version 0.73. [Online]. Available: http://CRAN.R-project.org/package=corrplot

[47] J. H. Maindonald and W. J. Braun, *DAAG: Data Analysis and Graphics Data and Functions*, 2015, r package version 1.22. [Online]. Available: http://CRAN.R-project.org/package=DAAG

[48] D. Sarkar, *Lattice: Multivariate Data Visualization with R*. New York: Springer, 2008, iSBN 978-0-387-75968-5. [Online]. Available: http://lmdvr.r-forge.r-project.org

[49] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, "Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), held in conjunction with SC15*, Austin, TX, Nov. 2015, pp. 30–35.

[50] T. Karras, "Maximizing parallelism in the construction of bvhs, octrees, and k-d trees," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 2012, pp. 33–37.

[51] C. Brownlee, J. Patchett, L.-T. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C. D. Hansen, "A Study of Ray Tracing Large-scale Scientific Data in Two Widely Used Parallel Visualization Applications," in *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.