Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis Nr.

Investigation and prediction of distributed volume rendering performance

Gleb Tkachev

Course of Study: Computer Science

Examiner: Prof. Dr. Thomas Ertl

Supervisor: Dr. Steffen Frey
Christoph Müller
Valentin Bruder

Commenced: July 11, 2016

Completed: January 10, 2017

CR-Classification: I.3.8

Abstract

In this work, I describe the process of developing a cluster scalability model that is capable of predicting performance of a parallel rendering application running on a cluster while only having data that can be obtained from one of its nodes. I begin by studying scaling behavior of a single cluster, employing linear regression and neural networks to construct a cluster-specific scalability model, which implicitly captures its hardware characteristics. I use this model as a foundation for further work, developing a hardware-agnostic cluster scalability model.

Instead of using explicit hardware characteristics as input, the hardware-agnostic model takes in a distribution of node computation time, which encapsulates local computational load of a rendering application, enabling the model to focus on predicting communication overhead of a cluster. This allows simulation of different hardware by varying the node computation time, gathering enough data to train a neural network that predicts the overall performance of the rendering application on a cluster with arbitrary node hardware.

Table of contents

1. Introduction ... 7

   1.1 Motivation ... 7

   1.2 Goal of the project ... 8

   1.3 Outline ... 8

2. Core concepts and related work ... 11

   2.1 Volume rendering ... 11

   2.2 Distributed volume rendering ... 14

   2.3 "2-3 swap" composition ... 19

   2.4 Machine learning ... 21

      2.4.1 Linear regression ... 23

      2.4.2 Artificial neural networks ... 24

   2.5 Related work ... 28

3. Cluster scalability model ... 31

   3.1 The rendering application ... 31

   3.2 Experiment setup and results ... 33

   3.3 Prediction model ... 36

      3.3.1 Linear regression ... 36

      3.3.2 Nonlinear extensions ... 39

      3.3.3 Neural network ... 42

4. Hardware-agnostic cluster scalability model ... 47

   4.1 Rendering simulation ... 48

   4.2 Prediction model ... 53

      4.2.1 Artificial generation of node time histograms ... 54

      4.2.2 Final model ... 56

5. Conclusions and future work ... 59

   5.1 Summary ... 59

   5.2 Current limitations and future work ... 60

References ... 62

Table of figures

Figure 1: Depiction of front-to-back compositing 14
Figure 2: Object-space partitioning vs image-space partitioning 16
Figure 3: Depiction of the direct-send composition scheme 17
Figure 4: Depiction of the binary-swap composition scheme 18
Figure 5: An example of a 2-3 swap compositing tree 20
Figure 6: Group merging operation for the example compositing tree 20
Figure 7: An example of an ANN architecture 24
Figure 8: An example of the rendering results 32
Figure 9: Overall volume renderer performance 35
Figure 10: Cluster frame time prediction using linear regression 37
Figure 11: Cluster frames-per-second prediction using linear regression 38
Figure 12: Algorithm for generating additional polynomial features 39
Figure 13: Cluster frame time prediction using linear regression, with additional polynomial features generated from the original data. 40
Figure 14: Extrapolation properties of the linear regression model 41
Figure 15: Cluster frame time prediction using a neural network 43
Figure 16: Extrapolation properties of a single-layer neural network model 44
Figure 17: Extrapolation properties of a neural network with ReLU activation 45
Figure 18: Comparison of real and simulated performance 49
Figure 19: Local computation time of each node during a single run 50
Figure 20: Comparison of real and simulated performance, with simulation using maximum local computation time 51
Figure 21: Comparison of real and simulated performance, with simulation using local computation time distributions 52
Figure 22: Measured distributions of local computation time 53
Figure 23: Artificially generated distributions of local computation time 54
Figure 24: Evaluation of the final model on the validation dataset 55
Figure 25: Learning curves of the final model. 56
Figure 26: Evaluation of the final model without the histogram 'bin features' 57

1. Introduction

1.1 Motivation

Volume rendering is a visualization technique that has found application in many fields, including biology, medicine, meteorology, astrophysics, etc. It enables the user to explore structures in three-dimensional data, a task which is common for natural sciences and studying physical phenomena.

As measurement and simulation technology improves, both in methods and in hardware, the size of the resulting datasets keeps increasing. Combined with the computationally intensive nature of volume rendering and its interactivity requirements, this presents a growing challenge for visualization applications. Even though their methods and hardware are also improving, it is not enough to keep up with the amount of generated data, so the parallelism of rendering tasks is exploited, solving the performance problem quantitatively, by adding more hardware and parallelizing.

Parallel volume rendering is not a trivial problem, which is demonstrated by both the amount of research in this field, and the number of different rendering methods developed over the years. One of the effects of this complexity is the increasing difficulty of making predictions about the overall performance of the application.

Although performance prediction may appear as a secondary task, it has a number of important applications. First, it supports decisions made during systems design: a performance model can be quickly used to estimate how hypothetical changes to hardware or software will affect the performance of an application, without implementing them. Second, it could be used for recognizing the need and driving the optimization process of a newly-created application. Implementation of parallel applications is a difficult, error-prone task, and an existing model may help find issues in application performance. Finally, a performance model is useful during equipment procurement, allowing potential performance to be estimated before purchasing expensive hardware, helping to achieve an optimal performance-to-price ratio.

Performance prediction is not a new problem; therefore, an extensive amount of research is already available in the field of high performance computing. However, volume rendering is an inherently interactive technique, with user's input having a large impact on the load balance and, ultimately, application performance. This makes HPC models, which assume stable and repetitive performance, inapplicable

to parallel volume rendering, warranting development of specialized performance models.

1.2 Goal of the project

In this project, I aim to create a performance model for parallel volume rendering applications. The model should predict performance of a typical rendering application under different input (dataset size, image size, etc.) and hardware (GPU, node number, etc.) parameters.

The model is developed with the purpose of supporting hardware procurement decisions, therefore it should be capable of predicting performance of a cluster, while only having access to data collected on as single node of the cluster. This means that I can execute neither the volume rendering application, nor any benchmarks/skeletons on the whole cluster, but can run benchmarks on one of its nodes. Due to this limitation and overall complexity of the problem, machine learning techniques are to be used as a foundation for the model, aiming to provide data-driven decisions without requiring deep domain-specific knowledge from the user.

Two multi-GPU clusters are available for collection of performance data, however one of them is reserved for validation of the final results, and cannot be used for training or tuning the model.

In summary, the goal is to implement a typical volume rendering application, collect performance data on one of the clusters, and to construct a machine learning model that can predict performance of another cluster, without running the rendering application on that cluster.

1.3 Outline

In **Chapter 2** I briefly discuss the fundamental concepts used in this thesis. I describe the volume rendering problem more formally, and talk about the ways to parallelize it. Then, I present an advanced parallel volume rendering algorithm that is used in my implementation and defines the communication costs of the application. Finally, I outline the core concepts of machine learning, and a few particular techniques used in this work.

In **Chapter 3** I set out to develop a performance model of a single cluster. Since

only a single set of hardware is used, the model has no input features describing the hardware, which is implicit in the model. I construct this model to better understand the scaling behavior of a cluster, and assess suitability of various machine learning approaches. I begin with linear regression and its extensions, which gives reasonable results, but requires error-prone manual feature selection. So, I go through several iterations of neural networks, arriving at a model which has both acceptable accuracy, and better scales to a larger number of features. Although, neural networks show high accuracy 'near training data', they are not well-suited for extrapolation, which is the final goal of this project. I address this problem in the chapter 4.

The goal of **Chapter 4** is to create a general cluster performance model. I begin by recognizing that the problem of poor extrapolation can be solved by obtaining more training data, from multiple clusters with different hardware. Since only a single cluster is available for running experiments, the model needs to be reformulated in a way that allows obtaining more data on a single set of hardware. Before, I was aiming to create a holistic model that would map application-specific parameters (e.g. image size, dataset size), hardware parameters (e.g. GPU specs) and cluster size to overall rendering time. Now, the hardware and application-specific parameters are abstracted away behind a *local computation time* parameter, which describes how much time each node takes to perform local rendering. The new formulation focuses on predicting the communication overhead of the cluster. By varying the local computation time values, it is possible to *simulate* different hypothetical hardware on a single cluster by stalling each node instead of performing actual local rendering. This way additional training data is gathered, helping to isolate general effects of cluster communication. Then, the final model is trained and validated using data obtained from a different cluster.

Chapter 5 contains a short summary of the results, their limitations and a discussion of potential future work.

2. Core concepts and related work

2.1 Volume rendering

Volume rendering is a set of techniques in scientific visualization, concerned with visualizing data defined on a 3D domain. This data may represent real-world phenomena recorded with 3D-scanning methods, e.g. using computer tomography to scan human body in medicine [1], or be a result of a simulation, such as supernova birth simulation in astrophysics [2]. Regardless of the origin, one property is always shared: in its raw form, the data is inaccessible to humans, both due to the sheer number of data points and its volumetric, possibly multivariate character, making mapping to a 2D representation non-trivial. Thus, the goal of volume rendering is to map the volumetric data to a human-accessible form, making its further analysis possible.

There are three main families of volume rendering methods: *slicing methods*, which visualize only 2D subsets of the volume (‘slices’). Although they do not provide overview of the data, they are still used in medicine due to their simplicity and lack of complex transformations, which might obscure important details. *Indirect methods* visualize structures derived from the volume data, e.g. constructing a polygonal iso-surface and rendering it on an image using polygonal rasterization techniques. Finally, *direct methods*, derived from the rendering equation [3], which governs light transport. Direct methods aim to transform and project on the image the volume data itself.

The direct methods have a common trait: since the volume data itself is ‘put’ on the image, there needs to be defined a mapping from the abstract physical properties, such as density, velocity, etc., to the color and transparency of the image. This mapping is called *a transfer function*. Usage of a transfer function does not only make the direct volume rendering possible, but also can provide further control to the user: by changing the function he can focus on different features and value ranges of the data.

A number of direct volume rendering techniques have been developed, e.g. shear-warp [4], volume splatting [5], texture-based methods [6]. In this work, I use a method called *volume raycasting* [7], which became more popular with the rise of programmable GPU shaders and general purpose GPU computing.

When rendering an image of the volume using raycasting, one has a scene containing a bounding box of the volume and a virtual camera. Typically, a perspective

camera is used, because it works similarly to a human eye and produces results that are more intuitive. A camera can be described as a series of linear operators, that transform points of the scene from local coordinate system of each object to the image coordinate system, effectively placing every point somewhere on the image. The overall transformation T is often specified in terms of three matrices (Equation 1).

$$T = PVM \tag{1}$$

Here M is the *model matrix*, which transforms *local coordinates* of an object into *world coordinates* to bring all objects into a common coordinate system. The model matrix is defined for each object and depends on the object's position, orientation and scale. The *view matrix* V is constructed from the camera's position and orientation and transforms world coordinates into *view coordinates*, which are centered and aligned to the camera. This is done to simplify the definition of the final matrix – the *projection matrix* P, which projects the points from the view coordinates into *image coordinates*. The matrix P is what determines the type of projection performed by the camera (orthogonal, perspective, etc.). All three matrices have a 4x4 size, representing transformations in three-dimensional *projective space*. The points in this space use homogeneous coordinates that can be represented using a four-component vector. Addition of the extra fourth component allows perspective projection to be expressed using linear operators. To convert a three-component vector (x, y, z) representing an affine point into homogeneous coordinates, one simply appends the fourth component w equal to one: $(x : y : z : 1)$. To perform the opposite conversion, all components are divided by the fourth component, which is then removed (Equation 2).

$$p_{proj} = (x : y : z : w) \longmapsto p_{affine} = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) \tag{2}$$

This way of implementing the camera using transformation matrices is very common, which is why I describe it, but further details are required to apply it to volume raycasting. To perform raycasting, a ray emanating from the camera is constructed for every pixel of the image. The ray comprises all points that are projected onto the pixel, and it is later traversed to determine the pixel's color. To this end, its origin and direction in world coordinates are computed. The origin of the ray lies on the image plane of the camera and can be obtained by transforming the point in image coordinates with the *inverse view-projection matrix* $(PV)^{-1}$ effectively 'unprojecting' the point from image back to world coordinates (Equation 3).

$$o_{proj} = (PV)^{-1}(x : y : 0 : 1)^{T} \tag{3}$$

Note that depth of zero is used (i.e. point lies exactly on the image plane) and the fourth component equal to one is appended. To get the ray direction, another point lying on the ray is required. For that a point with the same pixel coordinates, but an arbitrary non-zero depth z is inverse transformed (Equation 4). This correspond to a point that is projected onto the same pixel, but lies some distance away from the camera. Next, the two obtained points are converted to affine coordinates using Equation 2. Finally, the ray direction is computed as a normalized difference between the two points (Equation 5).

$$o'_{proj} = (PV)^{-1}(x : y : z : 1)^T \tag{4}$$

$$d = \frac{o'_{affine} - o_{affine}}{\left\| o'_{affine} - o_{affine} \right\|} \tag{5}$$

Then, the ray is intersected with the volume box. If the ray has missed the box, then the corresponding pixel is assigned background color. Otherwise, the intersection results in two points: ray entry and ray exit. To obtain the pixel color the ray is traversed step-by-step, moving a fixed distance h along the ray every iteration (Equation 6). For volume data defined on a uniform grid, it is reasonable to select the step size h to be equal to voxel size, such that regardless of ray orientation no voxel can be ‘stepped over’.

$$p_{i+1} = p_i + d \cdot h \tag{6}$$

where p_i is the current position in the i-th step of the algorithm.

Before the first step, the position is initialized to the exit point, and the current color to background. As steps are taken towards the camera, the volume texture is sampled and resulting values are converted to color and opacity by the transfer function. Color and opacity obtained at each step are blended with the current color, effectively simulating light transport, as it passes through the volume voxel. The resulting change in light intensity depends on the following effects: 1) absorption of light by the volume 2) emission of light by a potentially luminous material 3) scattering of light inside the volume [1]. If the problem is simplified to consider only light absorption, then the blending can be performed using the ‘over’ operator (Equation 7) [8].

$$C_{i+1} = (1 - \alpha)C_i + \alpha C \tag{7}$$

where C_i and C_{i+1} is the accumulated color before and after i-th step respectively; C and α are color and opacity obtained by sampling the volume.

This method is known as *back-to-front* compositing. A common optimization is to

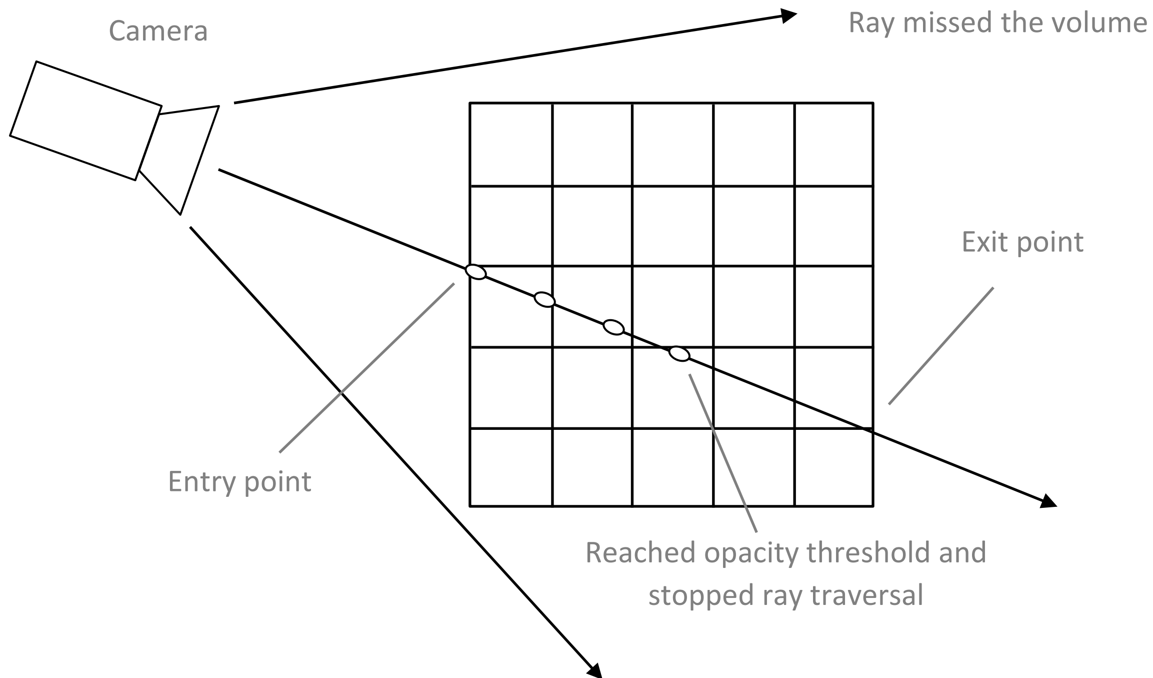

Figure 1: Depiction of direct volume rendering with front-to-back compositing. When a ray emanating from the camera hits the volume, it is stepped through, starting at the entry point. Each step the volume density is sampled, converted to color and opacity with the transfer function and blended with the current result. If a high value of opacity is reached, the traversal is stopped prematurely, since further steps would yield only a small contribution to the result.

use *front-to-back* compositing [1] (Figure 1), starting at the ray entry point and traversing the ray in the opposite direction, away from the camera. The idea behind this, is that parts of the volume that are far away from the camera are often occluded by the rest of the volume. Thus, they contribute very little to the final color of the pixel, and yet they are still traversed. In front-to-back compositing, the blending operations are reversed, which also requires the implementation to maintain not only the current pixel color, but also the opacity (Equation 8). This allows for *early ray termination*, stopping ray traversal when an opacity value close to 1 has been reached, meaning that further computation will yield very little contribution to the result and can be skipped.

$$C_{i+1} = C_i + (1 - \alpha)C\alpha \tag{8}$$

$$\alpha_{i+1} = a_i + (1 - \alpha)\alpha$$

2.2 Distributed volume rendering

As the size of the volume dataset increases, computational power of a single GPU

becomes insufficient to render it in real-time. This presents a problem, since, volume rendering is a technique that enables a human to explore volumetric data, and as the frame rate drops, interaction becomes cumbersome and frustrating.

An obvious solution is to parallelize the problem and use more hardware to perform the rendering. However, the details of how to parallelize the problem are not as obvious. A generic way of classifying parallel rendering applications has been developed by Molner et al. [9]. Rendering is essentially computing the effect of geometric primitives on pixels of the image, and as such can be treated as a problem of sorting the primitives to the image. A standard rendering pipeline consists of two stages: geometry processing (transformation, clipping to screen, etc.) and rasterization (shading, occlusion, etc.), which results in three possible places where the sorting can occur: during geometry processing stage ("sort-first"), during rasterization ("sort-last"), or in-between the two ("sort-middle"). *Sort-first* renderers distribute the primitives among the nodes according to where they fall on the screen. Each node is assigned a region of the image, and receives primitives that contribute to that region. In *sort-middle* approach, primitives are at first distributed randomly for geometry processing, but redistributed before rasterization occurs. So, similarly to sort-first, the nodes are assigned regions of the image, but only for rasterization stage. Finally, in *sort-last* applications, the primitives are distributed arbitrarily, and all the computation is performed independently until the visibility resolution stage, during which the nodes exchange the pixel data to form the final image.

This classification can be applied to volume rendering applications, and results in roughly three groups of methods, expressed in how computational load is partitioned among the renderers: *object-space partitioning* (sort-last), *image-space partitioning* (sort-first) and *hybrid*, which try to find a compromise between the former two (Figure 2). *Object-space* methods parallelize the problem by assigning each rendering node a volume partition. When a frame is being rendered, each node simply renders its partition, but afterwards a complex composition step has to be performed, resolving occlusions and merging the results of each node into a single image. *Image-space* methods parallelize the problem by assigning each node a partition of the final image. During rendering, nodes have to determine, which parts of the volume contribute to their assigned image partition, and render them, producing a tile of the final image. No complex composition step is required, as the final image is assembled by simply 'gluing' together results from each node.

In this project, I am investigating the performance of *object-space partitioning* methods, so I discuss them in further detail. Conceptually, the rendering process has two phases: local computation and composition.

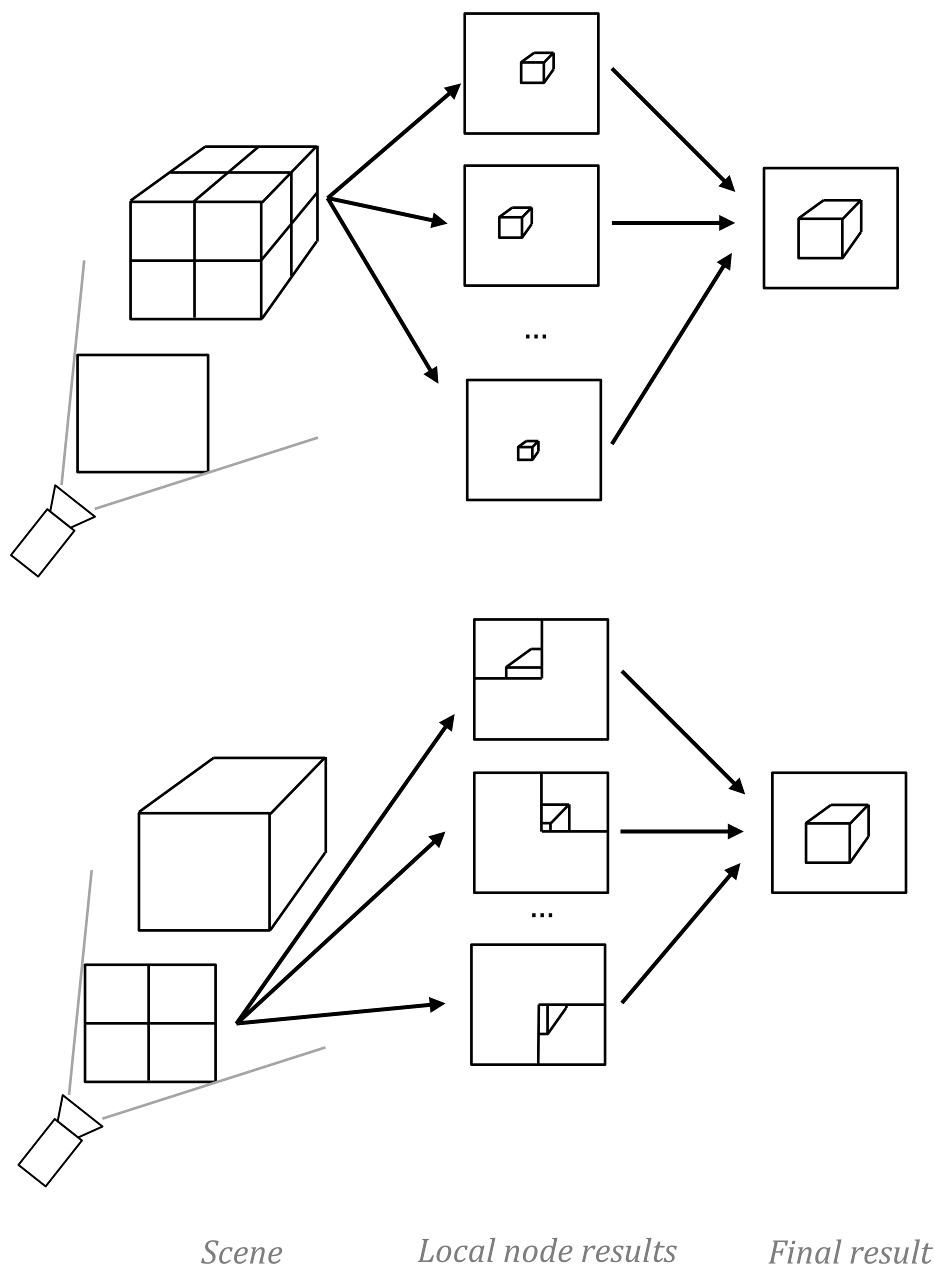

Figure 2: In object-space partitioning approach (top) the volume is subdivided into bricks and distributed among nodes. When each node has rendered its brick, all local results are composed into the final image. When using image-space partitioning (bottom), each node is assigned a region of the image and renders the whole volume into that region. Later, all image chunks are 'glued' together to form the final image.

First, the *local computation phase* is performed, during which each node renders its volume partition(s) into its frame buffer. If multiple partitions are to be rendered, they are rendered sequentially in visibility order, compositing the results on-the-fly into the same buffer (assuming convex partitions without gaps between

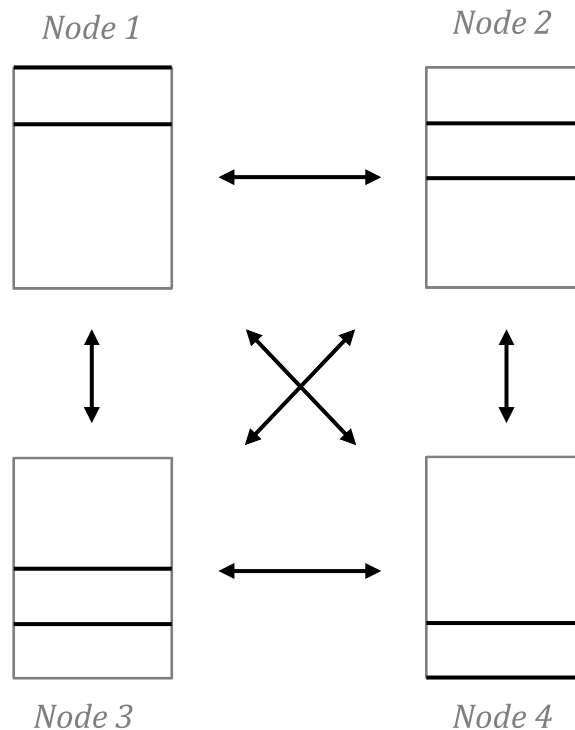

Figure 3: Depiction of the direct-send composition scheme. Every node is assigned a region of the final image. It receives data falling into that region from every other node to compose it into a chunk of the final image. Afterwards, all chunks are sent to a single node and 'glued' together to get the final result.

them). Note, that since the volume is partitioned before the rendering (independently from the current camera position), node's volume partitions may fall anywhere on the final image.

Second, the *composition phase* is performed, blending the local results of each node into the final image. After the local phase, each node has a full-sized image, but only with its own partition(s) rendered on it. Since it is common to use convex volume partitions, the nodes can be sorted according to visibility of their corresponding partitions. Then, a straight-forward way of performing composition is to gather all the local results on a single node, and blend them sequentially. However, this naïve approach is not scalable, since during the composition most of the nodes will stall, waiting for a single compositing node.

Direct-send is a more advanced parallel composition scheme in which each node participates in the composition process [10]. Each node is assigned a different region of the image, so it receives a corresponding chunk of local results from each node, and blends them all together to get a chunk of the final image. Then, these final chunks are gathered on a single node and 'glued' together to form the final image (Figure 3). Although all nodes perform compositing computations, this

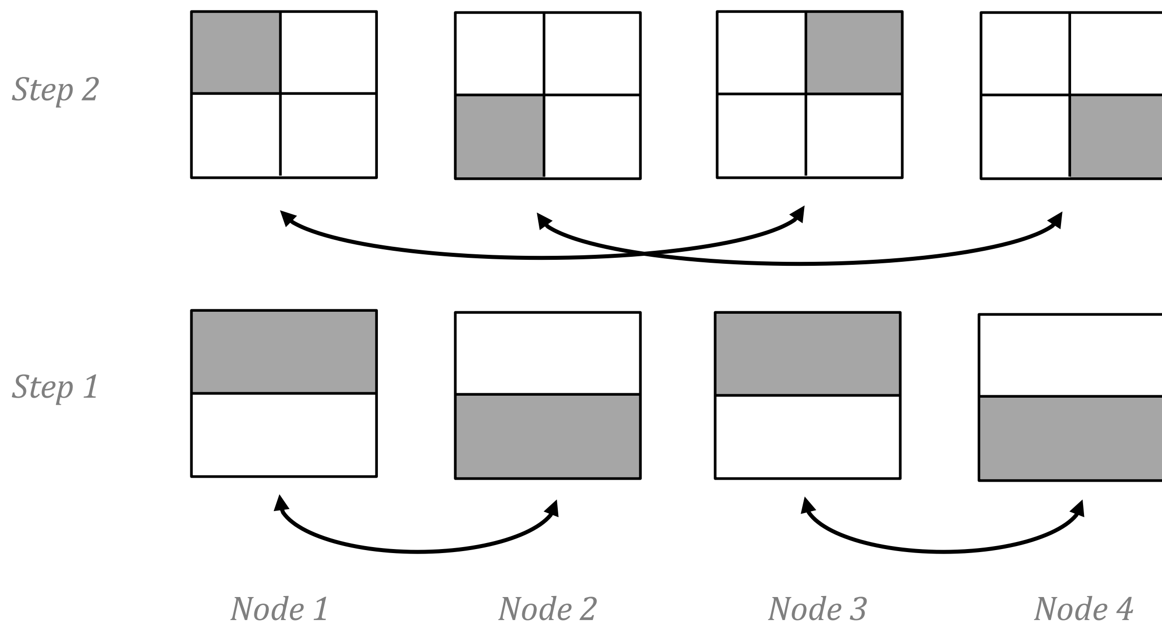

Figure 4: Depiction of the binary-swap composition scheme. Each step of the algorithm nodes exchange and compose data in pairs. Initially, every node has data covering the whole image, but only with one volume partition rendered. After all the steps are finished, every node ends up with a small chunk of the final image, which is sent to a single node to form the final result.

method still may scale poorly due to the high number of messages being exchanged simultaneously. Specifically, if there are N nodes, then the image is partitioned into N chunks; every node sends out $N - 1$ chunks of its local results, and receives $N - 1$ chunks from other nodes, so each node communicates with every other node, resulting in $N \times (N - 1)$ messages exchanged [11].

To address the problem of simultaneous exchange of many messages, more advanced composition schemes have been proposed. In *binary-swap* scheme [11], composition is performed in steps, during which nodes exchange data in pairs, gradually composing the final image (Figure 4). The algorithm begins with every node having a full image, but only with its own local rendering results. In every step, each node swaps half of its image with a neighbor, for example, sending its top half and receiving neighbor's bottom half. After the data has been exchanged, each of the two nodes composes the received image chunk with one of its own, and ends up with half of the image region it had before, but with 'twice as much' data rendered to it. This process is repeated until each node has a chunk of the final image with all possible contributions rendered to it. Similarly to direct-send, the chunks are sent to a single node to be 'glued' into the complete final image. Since composition takes $\log_2 N$ steps, the overall number of messages exchanged is

$N \log_2 N$, which is fewer than in direct-send scheme. Additionally, nodes exchange messages in pairs, resulting in fewer simultaneous messages and performing better in networks of more complex topology, where direct node-to-node links between every node are not available. Although showing better experimental performance [12], binary-swap scheme is limited to power-of-two cluster sizes. This problem is addressed in section 2.3, where a generalization of the binary-swap scheme is described.

Another problem that commonly occurs in distributed volume rendering applications is *load imbalance*. Since all nodes need to finish their local computation before composition phase can occur, the overall frame time is defined by the node that takes the longest to perform its part. Although the volume is partitioned equally among the nodes, a large disparity in local rendering time is possible due to difference in volume bricks' footprint. A *footprint* of a volume brick is its projection on the image plane of the camera. The larger the area of the footprint, the more pixels of the image are affected by the brick, thus more rays must be cast and more volume samples taken, increasing the render time. As camera moves around the volume, perspective projection causes volume bricks to appear larger or smaller, increasing or decreasing local rendering time of different nodes, and affecting the overall performance of the application. There are techniques designed to combat this flaw, dynamically assigning volume partitions to the nodes during runtime [13], but in this work, I focus on static object-space partitioning and its effects on the renderer performance.

2.3 "2-3 swap" composition

2-3 swap composition scheme is an extension of the binary-swap scheme [11], that can be used with an arbitrary number of rendering nodes. Both schemes result in a smaller number of messages being exchanged among nodes, compared to the direct-send algorithm. Albeit the overall amount of data sent is higher, by reducing the impact of the per-message overhead, and communicating in smaller groups of nodes, they show better experimental results [12].

In 2-3 swap composition, the composition is performed in steps, during which processors exchange and compose data in small groups. The scheduling of composition operations is based on a pre-constructed compositing tree (see Figure 5 for an example). Every level of the tree represents a step of the algorithm, while nodes represent groups in which processors exchange data. The leaf nodes are the initial state of the algorithm, and each corresponds to a group with just one processor.

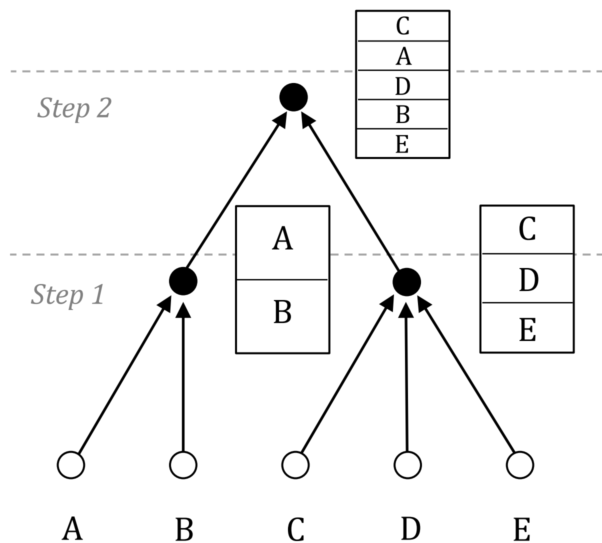

Figure 5: An example of 2-3 swap compositing tree for 5 processors (A-E). The algorithm comprises two steps: during the first step, the processors exchange data in two groups: (A, B) and (C, D, E). During the second step, all five processors are merged into a single group.

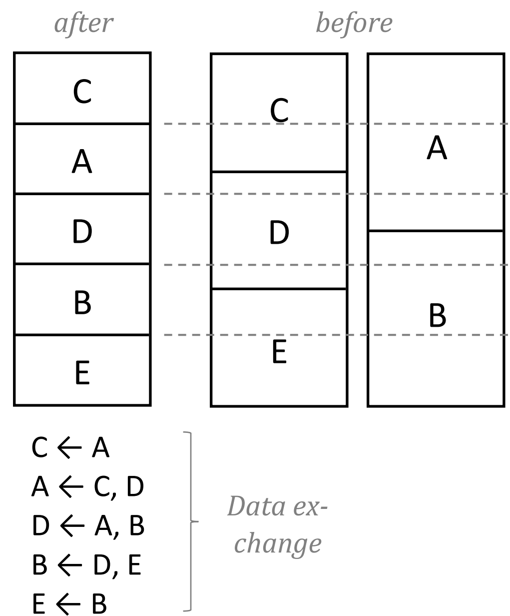

Figure 6: Group merging operation for the second step of the example compositing tree. Before the operation, processors (C, D, E) were responsible for thirds of the image, and processors (A, B) for halves. After the merge, each processor becomes responsible for a fifth of the final image, and receives data overlapping its region from other processors.

The root node represents the state after the last composition operation, in which each processor is left with a chunk of the final image.

Note: to avoid confusion, in this section, I use the term "processors" to refer to the nodes of the rendering cluster, and "nodes" to refer to the nodes of the compositing tree.

During the course of the algorithm, one moves from the bottom of the tree to the top, merging together groups of processors, and exchanging data between them. Initially (at the leaf node level), each processor is responsible for the whole image region, but has only its own volume partition(s) rendered on it. As data is exchanged with other processors, the image region for which the processor is responsible shrinks, while its contents become more complete, i.e. contribution of other volume partitions is taken into account. Until finally, each processor is left with a complete chunk of the final image.

To better illustrate this process, consider a 2-3 swap compositing tree for 5 processors (Figure 5). In this case, the composition occurs in two steps, first merging data within two groups of nodes, and then merging the two groups together at the root node. Before the first step, every processor is responsible for the whole image region. During the first step, processors (A, B) exchange data to form a new group: processor A becomes responsible for the top half of the image, receiving top half of B's buffer, and vice versa. A similar exchange happens among processors (C, D, E), but each of them exchanges data with two neighbors. In the second step, the two groups are merged to form the final group of five processors. Each processor becomes responsible for a fifth of the image, and receives data from those processors, who have data overlapping its new image region (Figure 6). When the second step is completed, each processor ends up with a finished chunk of the final image. All that is left is to send all these chunks to a single processor, to form the final image.

An insight into scalability properties of the algorithm can be gained by examining the upper-bounds on the number of messages and amount of data exchanged, provided by Yu et al. [12]. It is shown, that at any step of the algorithm, a processor exchanges data with at most 4 other processors, thus at most $4 \times \lfloor \log_2 N \rfloor$ messages are sent during the whole composition process. For the amount of data exchanged, the upper-bound is $\frac{4}{3} \times P$, with P being the number of pixels in the image, but the algorithm shows better average performance in practice.

2.4 Machine learning

Machine learning is a field of computer science, which is concerned with developing methods that make predictions based on data, as opposed to being explicitly programmed to make static decisions. Machine learning is closely related to the fields of mathematical optimization and computational statistics, which serve as its theoretical and methodological foundation.

The input for a typical machine learning problem is a set of data points, called the *training set*. The data points could be either scalars or N-dimensional vectors, in which case the input data is said to have N input *features*. Tasks where the input is accompanied by its corresponding target outputs are instances of *supervised learning*, where a machine learning model is given examples of correct answers and is expected to learn the general dependency between inputs and outputs. This is opposed to *unsupervised learning*, where the model is given no target outputs, and is supposed to find generic patterns in the data.

Based on the desired output, machine learning tasks can be further classified into the following categories [14]:

- Classification
- Regression
- Clustering
- Density estimation
- Dimensionality reduction

In *classification* problems, the machine learning model, given an input data point, has to assign it to one of predefined classes. *Clustering* problems are defined similarly, but the classes are not specified before-hand. *Regression* tasks are supervised and require the model to learn the dependency between the input and one or more continuous output variables.

In a standard machine learning workflow, one begins by selecting a model, that is believed to be capable of capturing the relationship between the input and the output. Then, the model is *trained* by fitting its parameters to the input data, i.e. finding such model parameters, that minimize the value of the *loss function* on the training set. The loss function, given a set of data points and their target outputs, expresses how good the model is at predicting the output.

After training is complete, the quality of the model's prediction should be evaluated. As it is desirable for the model to generalize beyond the training data and give accurate predictions for all possible inputs, it is important to perform evaluation on the data that wasn't included in the training. A dataset used for evaluation is called the *test set*; a common strategy known as *holdout* is to simply split the input data into training and test sets before training the model (often in 2:1 proportion). To evaluate the model, one or more accuracy scores are computed on the test set, by using the loss function used to train the model, or other scores that capture different notions of "accuracy". In this work, I use a Mean Squared Error loss function and the R^2 score (coefficient of determination), that shows how much of the variation of the output is captured by the model [15]:

$$R^2 = 1 - \frac{S_{res}}{S_{total}} = 1 - \frac{\sum_i (y_i - o_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{9}$$

where o_i is the output of the model, y_i is the target output and $\bar{y}$ is its mean.

Results of the validation can often be used to diagnose some broad classes of problems with the resulting model. Poor accuracy on both the training and the test sets

suggest that the model might be *underfitted*, meaning that it is too simple to capture the relationship between the input and the output, for example when using a linear model to fit non-linear data. If only the test set accuracy is low, the model might be *overfitted*, capturing not only the general trends of the data, but also any noise and variation specific to the training set. In general, one wants the accuracy to be low both on the training and the validation datasets, which is indicative of a model that is not only capable of fitting the training data, but also generalizes well to previously unseen input.

In this work, I rely on linear regression and neural networks for performance prediction, thus I discuss them in closer detail.

2.4.1 Linear regression

Linear regression model assumes a linear relationship between input and output variables, and thus uses a linear function to predict the output value. Conceptually, this corresponds to fitting the "best" line to the training set of points, and then using this line to predict the output. What exactly is meant by the "best" line is determined by the *estimator,* i.e. the algorithm used for determining parameters of the model.

More formally, given a set of N data points with m features, and a single dependent variable y:

$$\{y_i, x_{i1}, x_{i2} \dots x_{im}\}_{i=1}^{N} \tag{10}$$

The following linear relationship is assumed to hold for all N data points:

$$y_i = \boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{T}} \boldsymbol{w} + \varepsilon_i \tag{11}$$

where $\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{T}} = (1, x_{i1}, x_{i2} \dots x_{im})^T$ is a vector of input variables, $\boldsymbol{w}$ is a vector of model parameters, and ε_i is a scalar residual term, which models deviation of the input data from the linear relationship. The same relationship can be expressed in matrix form:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\varepsilon} \tag{12}$$

Using Ordinary Least Squares estimator, the goal is to find $\boldsymbol{w}$ that minimizes the sum of squared residual terms:

$$\min_{\boldsymbol{w}} \boldsymbol{\varepsilon}^{\boldsymbol{T}} \boldsymbol{\varepsilon} \tag{13}$$

which corresponds to computing [15]:

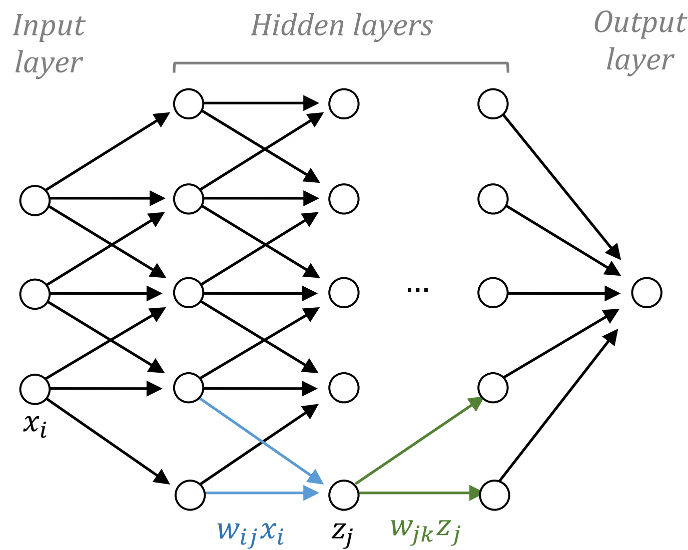

Figure 7: An example of ANN architecture, with an input layer, an output layer and a set of hidden layers shown. The input data enters the network as the signal coming from the input layer and propagates through the network left-to-right, until the final output value is computed.

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{14}$$

Yielding a line through the training set, which minimizes average vertical distance to the training data points.

2.4.2 Artificial neural networks

Artificial neural networks are a machine learning technique that was inspired by the structure of a biological brain. An ANN is a network of interconnected *neurons*, in which a signal flows from the input neurons through the whole network to the output neurons, being transformed on the way. A neuron can be viewed as a unit which outputs a nonlinear weighted sum of the incoming signals (z_j): it sums the incoming signals (z_i) according to weights of the connections (w_{ij}), and then applies the *activation function* (g) to the result (Equation 15) [16]. The activation function is a nonlinear function which enables ANNs to model complex nonlinear input data. Without it, the whole network could be reduced to a single linear operator.

$$z_j = g(a_j) = g\left(\sum_{i=0}^{d} w_{ij} z_i\right) \tag{15}$$

The architecture of a network, i.e. the number of neurons and the way they are

connected, varies from network to network. A common architecture, which I use in this work, is series of fully interconnected layers of neurons (Figure 7). The first fictitious layer is an input layer, and has a neuron for every input feature. The last layer is the output layer, and has a single neuron (when doing regression with a single dependent variable). The rest of the layers are called the hidden layers. Their number and the number of neurons they contain can be varied to control the complexity of the network.

During the training, input data is ‘fed’ to the network, gradually adjusting connection weights to improve network’s prediction accuracy. First, all the weights are initialized with random values. Care must be taken, as the details of initialization technique used may significantly affect the effectiveness of the training process. Next, the network is evaluated on the input data, making a left-to-right pass through the network, called *forward-propagation*. Next, the *back-propagation* algorithm [17] is used to compute all the gradients of the network, i.e. how much the overall loss is changed when changing each weight of the network. The algorithm is based on the chain rule of differentials and computing the derivative of the loss function w.r.t. to weights of the network. Considering a network with one output neuron (which is a case for univariate regression), the squared error loss function for a single data point can be written as:

$$E = \frac{1}{2}(y_t - y_o)^2 \tag{16}$$

with y_t being the target output of the network (provided with the input data) and y_o being the actual output, computed with forward-propagation. To compute the derivate of the loss w.r.t. to a weight, considering Equation 15 and applying the chain rule:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \tag{17}$$

The last term, for the derivative of neuron’s activation a_j w.r.t. to the weight, can be simplified to i-th neuron’s output z_i, since only one term in the sum depends on the weight w_{ij}:

$$\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{i=0}^{d} w_{ij} z_i\right) = z_i \tag{18}$$

The second term (the derivative of neuron’s output w.r.t. to its activation, i.e. the weighted sum of the input signals) is simply the derivative of the activation function:

$$\frac{\partial z_j}{\partial a_j} = \frac{\partial g(a_j)}{\partial a_j} \tag{19}$$

In case the neuron with output z_j resides in the last layer of the network (immediately before the output neuron), the first term representing the derivative of the loss w.r.t. to the neuron's output is:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_o} = y_o - y_t \tag{20}$$

However, if the neuron is not located in the last layer, the derivative has to be computed as a total derivative w.r.t. z_j, considering influence of every neuron in the *next* layer (having neurons with indices $L = \{m, n \dots\}$):

$$\frac{\partial E}{\partial z_j} = \sum_{l \in L} \left(\frac{\partial E}{\partial a_l} \frac{\partial a_l}{\partial z_j} \right) = \sum_{l \in L} \left(\frac{\partial E}{\partial z_l} \frac{\partial z_l}{\partial a_l} w_{jl} \right) \tag{21}$$

Derivatives w.r.t. to any weight in any layer of the network can be computed by recursively applying Equation 21. Each step, starting with the last layer, derivatives for the weights of the next layer are computed, moving backwards through the neural network (hence the name back-propagation). In the end, the derivatives can be used to update the weights, moving in the direction of decreasing loss, effectively performing *gradient descent*:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} \tag{22}$$

with manually chosen parameter α called the *learning rate*, which controls the speed of the descent.

In the simplest case, the gradient is computed for the whole training dataset, averaging all individual gradients resulting from each data point, after which a single update of the weights is performed. In order to improve memory requirements of this procedure, and to decrease chances of being 'stuck' in local minima of the loss function, often *stochastic gradient descent* is used. In this algorithm, one does not compute the gradient of the loss function on the whole dataset simultaneously, but splits the dataset into batches of 16 data points, and makes steps along gradient evaluated on these small subsets, iterating through the whole input dataset. It is stochastic, because the path taken in the parametric space depends on the order and contents of the batches that are chosen randomly. The idea is, that this approach introduces 'jittering' which can help to get out of local minima, but given a large enough batch size, the algorithm still moves towards the global optimum.

Furthermore, stochastic gradient descent can be parallelized to accelerate the training process [18].

Another improvement to the algorithm concerns adjustment of the learning rate. In this work, I use *Root Mean Square Propagation (RMSProp)* approach, in which the learning rate is adjusted separately for each weight, using a running average of its corresponding gradient. Dynamic learning rate adjustments allows a larger learning rate to be used in the beginning of training, gradually lowering it towards the end.

A problem common to most machine learning methods is the problem of *overfitting*. As the complexity of a neural network is increased by adding more neurons and layers, it becomes capable of fitting the training data so well, that it learns even the random noise present in the data, and becomes worse at generalizing to unobserved data. A typical solution to this problem is to introduce a regularization term to the loss function, penalizing large network weights:

$$E_r = E_0 + \frac{\lambda}{2} \sum_{i,j} w_{ij}^2 \tag{23}$$

where E_0 is the non-regularized loss and λ is the regularization parameter, which controls the amount of regularization. The idea is that this change to the loss function makes the gradient descent 'compromise' between a close fit and small network weights. This results in more weights having near-zero values, reducing the complexity of the network and alleviating overfitting.

The last element of neural network's implementation that remains to be covered is the *activation function*. The activation function is what allows the network to model nonlinear dependencies in the input data. When an output of a neuron z_j is computed, the activation function g is applied to the linear combination of the incoming signals (neuron's activation a_j, see Equation 15). A typical choice of activation function is to use the $tanh$ function [19]. However, many other functions have been proposed, among them is the *rectified linear unit (ReLU)* [20]:

$$g_{ReLU}(x) = \max(0, x) \tag{24}$$

however, since *max* function is not differentiable, a smooth approximation is often used:

$$g_{ReLU}(x) = \ln(1 + e^x) \tag{25}$$

ReLU activation helps with the problem of *vanishing gradients* that occurs in neural networks with many layers: a large number of layers implies many derivative

terms being multiplied together when computing the loss gradient using the chain rule (see Equation 17). If the derivatives are small (which is often the case for neurons with *tanh* activation), then their product is smaller still, resulting in only a tiny change of network's weights in every step of the gradient decent. This results in very slow training or even hinders the training completely, as gradients approach numerical precision of the floating-point numbers. ReLU on the other hand, always has a relatively large derivative, which both makes larger neural networks feasible and results in faster training time.

2.5 Related work

Performance prediction for parallel computing has many useful applications, such as runtime prediction for load balancing, informing system design and hardware procurement, or supporting performance tuning. So, it is not surprising that there is an ample amount of research investigating various performance prediction techniques.

Yang et al. [21] present an *observation-based* technique for predicting performance of an application on a different platform. They argue, that due to iterative structure of most scientific computation programs, a short partial execution of an application is sufficient to predict its overall performance. They develop a simple API to mark and time iterations within a running application, and use it to perform a partial execution of the same application on different hardware, collecting data, which is later transformed into an approximation of the overall performance.

Bailey and Snavely [22] developed a method for predicting application performance without running the application on the target system. They combine static source code analysis and code instrumentation to obtain an application signature that characterizes its computation load. Together with results of low-level benchmarks from the target machine, this allows for *simulation-based* performance prediction, obtaining an estimate of application performance on the target system.

A related approach is investigated by Sodhi et al. [23]. They predict performance of a parallel application using *application skeletons*. An application skeleton is a small program that represents a fraction of the overall computational load of the actual application. It is automatically generated through analysis of an execution trace and can be quickly run on a target system to estimate the performance of the whole application.

Several researchers have developed methods for extrapolating performance of a

parallel application on a cluster, from measurements made on smaller number of its nodes. Barnes et al. [24] propose a *regression-based* method capable of extrapolating performance to a larger number of computational nodes. Zhai et al. [25] created a framework, which first collects nodes' computation time sequentially on a single machine, and then uses it in a trace-driven simulation for estimating performance of a full cluster of nodes.

Although a lot of different methods have been designed for predicting performance of HPC applications, they are difficult to apply to visualization. Most of the methods make an explicit assumption that scientific applications are iterative, highly repetitive and have stable performance after the initialization phase. However, visualization applications are interactive, and can have a large performance variation depending on the user's choice of visualization parameters, such as camera orientation. Moreover, standard performance metrics, such as computation time, do not fully represent the performance of an interactive application, with latency being another important factor to consider. Finally, due to development in the field of general-purpose GPU computing, many visualization applications are now utilizing the computational capabilities of GPUs, which not only introduce another layer of parallelism, but also rely on SIMT architecture [26] and differ drastically from CPUs in performance.

On the topic of GPU performance models, Schaa and Kaeli [27] presented an analytical model for predicting execution time of a multi-GPU systems, with varying number of GPUs and the input data set size. Baghsorkhi et al. [28] developed a compiler-based approach for analyzing GPU kernel code and modelling its performance, which can be used for guiding performance optimization. Zhang and Owens [29] utilize micro-benchmarks to accurately measure various aspects of GPU performance and construct a model that guides GPU application optimization process.

In recent years, there has been some research on performance of parallel visualization applications, and volume rendering in particular. Rizzio et al. [30] constructed an analytical model for performance of a GPU cluster, separately considering different frame rendering phases. Larsen et al. [31] developed a performance model for rasterization, ray tracing and volume rendering algorithms to inform decision regarding feasibility of in-situ visualization. They have first constructed an analytical model of performance of every application being executed on a single machine, and used statistical methods to determine the constants. Later, they have extended the model to parallel execution by introducing a similar model for image composition performance.

Overall, despite abundant work in studying parallel application performance, only a limited number of researchers have attempted modelling and prediction of parallel visualization applications. Furthermore, a few, if any, investigated the effects of using hybrid parallelism, i.e. multi-GPU clusters, on the overall performance of an application.

3. Cluster scalability model

In this chapter, I report the process of constructing a scalability model that extrapolates performance of a rendering application to a larger number of nodes. Although the model is limited by the fact that it requires running the renderer on the actual cluster to obtain the training data, it represents scaling behavior of a typical distributed volume rendering application and serves as a foundation for a more generic model presented in chapter 4.

I begin by describing the volume rendering application and its capabilities. Then, I proceed to experiment setup and results of scalability testing. Finally, I use the data obtained and train several iterations of statistical models to predict performance of the renderer.

3.1 The rendering application

For the purpose of studying performance and scaling behavior of distributed volume rendering applications, and for collecting data required for training and eventual evaluation of prediction models, I have implemented a typical distributed volume rendering application.

My implementation utilizes *Nvidia CUDA* platform [32] for volume raycasting on GPUs, which allows usage of the same language and environment when writing both CPU and GPU code. CUDA comes with an extensive set of debugging and profiling tools, simplifying performance optimization critical for computation-intensive applications. It also provides the means for collecting various GPU metrics that could be used for performance prediction and explicit assignment of tasks to GPUs in multi-GPU systems.

For implementation of inter-node communication within the cluster, I use MS-MPI – Microsoft's implementation of MPI standard [33]. MPI provides a simple interface and covers most typical cluster communication patterns, making for a faster and more reliable application development. It specifies a set of asynchronous operation that can be used to decrease communication latency. Finally, it leaves the implementation details out of the standard, allowing vendors to make platform-specific performance optimizations, which are, again, critical for parallel volume rendering applications.

The rendering application can be classified as a *sort-last* renderer [9], and performs object-space partitioning to distribute the volume partitions among the

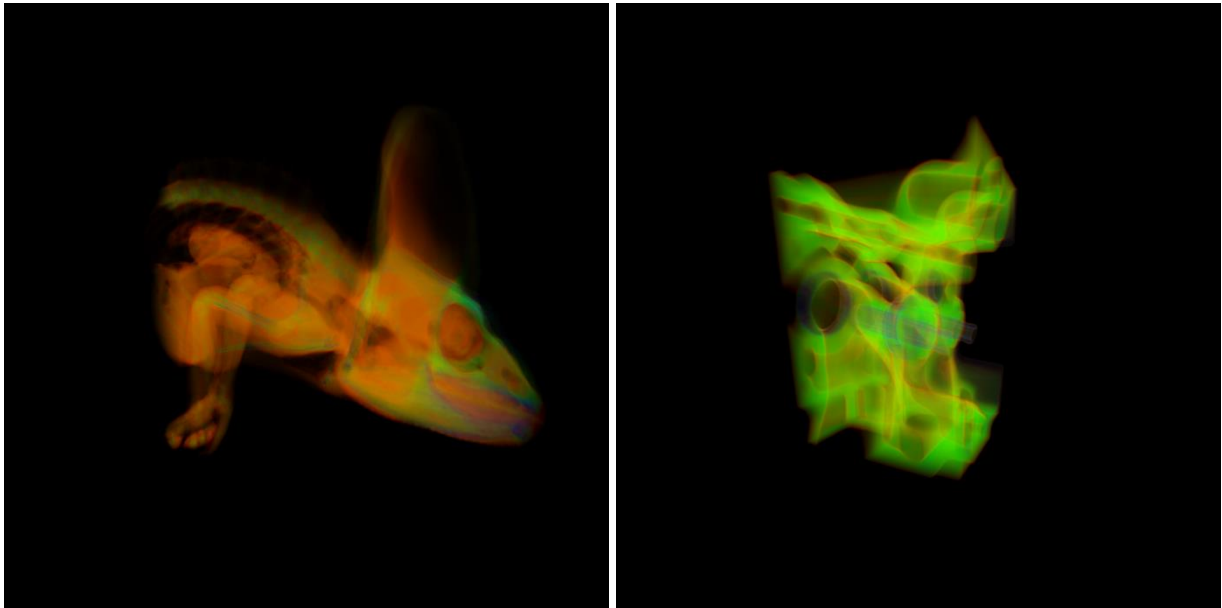

Figure 8: An example of the rendering results: chameleon dataset (left), engine dataset (right)

nodes of the cluster. Partitioning is performed recursively using a Kd-tree: starting with the whole volume and all cluster nodes, each step the volume is split along X, Y or Z axis into two partitions, with each partition being assigned half of the cluster nodes. If some of the partitions are still assigned more than one cluster node, the process is repeated recursively for those partitions. The splitting axis is changed on every recursion level, to keep partitions closer to a cubic shape. If the current number of cluster nodes is divisible by two, then the splitting is done exactly in the middle. Otherwise, the split point is shifted to keep the number of voxels in the two partitions proportional to the number of their corresponding cluster nodes. For example, if at some recursion level a partition corresponding to five cluster nodes is being split, the partition with three cluster nodes will be one and a half times larger than the partition with two cluster nodes. This way when the partitioning is done, all the partitions will have approximately the same number of voxels. Partition-to-node assignments are static and do not change during the rendering process. After each node has finished local rendering, the results are exchanged among the nodes to perform composition, with details depending on the composition scheme. Four composition schemes are implemented: naïve, direct-send [10], binary-swap [11] and 2-3 swap [12]. In naïve scheme, complete local rendering results from each node are sent directly to the master node for composition, i.e. composition tasks are not distributed among the nodes.

The renderer supports direct volume rendering by performing raycasting on

GPUs. Each node of the cluster can utilize multiple GPUs for rendering its volume partition, in this case, the partition is further subdivided among the GPUs. During frame rendering, after each GPU has finished rendering its part, an inter-GPU composition step is performed, either using a binary tree to schedule composition among the GPUs in parallel, or sequentially on the CPU. The renderer can also work in *metrics-mode*, performing multiple runs of GPU kernels to collect GPU performance metrics, such as the number of memory read/write transaction, texture cache hit rate, processor occupancy etc.

As described later in section 4.1, my application supports simulation of local node computation, which is later used to simulate rendering performance of a cluster with different GPUs. Simulation can be performed either with *node time vectors*, specifying fixed amount of time each node should stall simulating local computation, or with *node time histograms,* where each node stalls for a random amount of time sampled from a given distribution.

The rendering application is accompanied by the *operator program*, which performs remote camera control and live streaming of the rendering results from the main application. The operator program is used for validation of rendering results, and could also be used for studying human-related effects of volume rendering performance such as responsiveness.

3.2 Experiment setup and results

For running scalability experiments and obtaining training data I have used a cluster of 33 nodes, each equipped with:

- Two NVIDIA GeForce GTX 480 GPUs
- Two Intel Xeon E5620 2.40GHz CPUs
- Supermicro X8DAH motherboard
- 24Gb 1066 Mhz RAM
- Mellanox ConnectX IPoIB 16 Gb/s network adapter

For final validation of results, measurements have been performed on another cluster of 20 nodes with following hardware:

- NVIDIA Quadro M6000 GPU
- Two Intel Xeon E5-2640 v3 2.60Ghz CPUs
- ASUS Z10PG-D16 motherboard
- 128Gb 2400 Mhz RAM

- Mellanox ConnectX-3 IPoIB 54 Gb/s network adapter

Both cluster have full bidirectional interconnectivity (each node has a connection to every other node).

A full suite of performance experiments includes executing the rendering application on the cluster for every combination of the following input parameters:

- Composition scheme C ∈ {'naïve', 'direct-send', 'binary-swap', '2-3 swap'}
- Image resolution R ∈ {1024^2, 2048^2, 3072^2, 4096^2, 5120^2, 6144^2}
- Volume dataset size S ∈ {256^3, 512^3, 1024^3}
- Node number N ∈ {1, 2, …, 33}

Square image resolutions with fixed aspect ratio of 1:1 have been used to remove any effects that image aspect ratio could have on the renderer performance (changing the aspect ratio affects the proportion of rays that hit the volume box, changing the amount of computation performed). During all the runs the same 'chameleon' dataset has been used, scaled to different sizes. A single run consists of rendering 72 frames, while orbiting the camera twice around the volume in horizontal XZ-plane. During execution, on each node the renderer collects various CPU and GPU timings (and optionally GPU-metrics), and after all the frames have been rendered, writes the raw data to disk. When all the runs are completed, a post-processing step is executed, collecting and aggregating the raw data for further analysis.

Next I present a concise account of the renderer's performance. I begin by examining relative performance of four implemented composition schemes (Figure 9, top-left). As expected, 2-3 swap scheme demonstrates the best performance, while also supporting non-power-of-two number of nodes. For this reason, I assume it to be a typical choice for parallel volume rendering applications, and use for all experiments in the rest of this work. Wherever composition scheme is not specified, usage of 2-3 swap is implied.

On Figure 9 top-right, I depict the dependency between median frame time and image size. The error bars represent minimal and maximal frame time that occurred in that renderer run. The disparity between minimal and maximal frame time can be better understood by looking at how the frame time changes during a run (Figure 9, bottom-right). One can clearly see, how camera rotation causes repetitive variation in performance. When a node's partition is closer to the camera, the partition covers more pixels in the image, resulting in more rays being cast through the volume. This causes some of the nodes to take a considerably longer time to render their partitions, which leads to load imbalance and overall worse

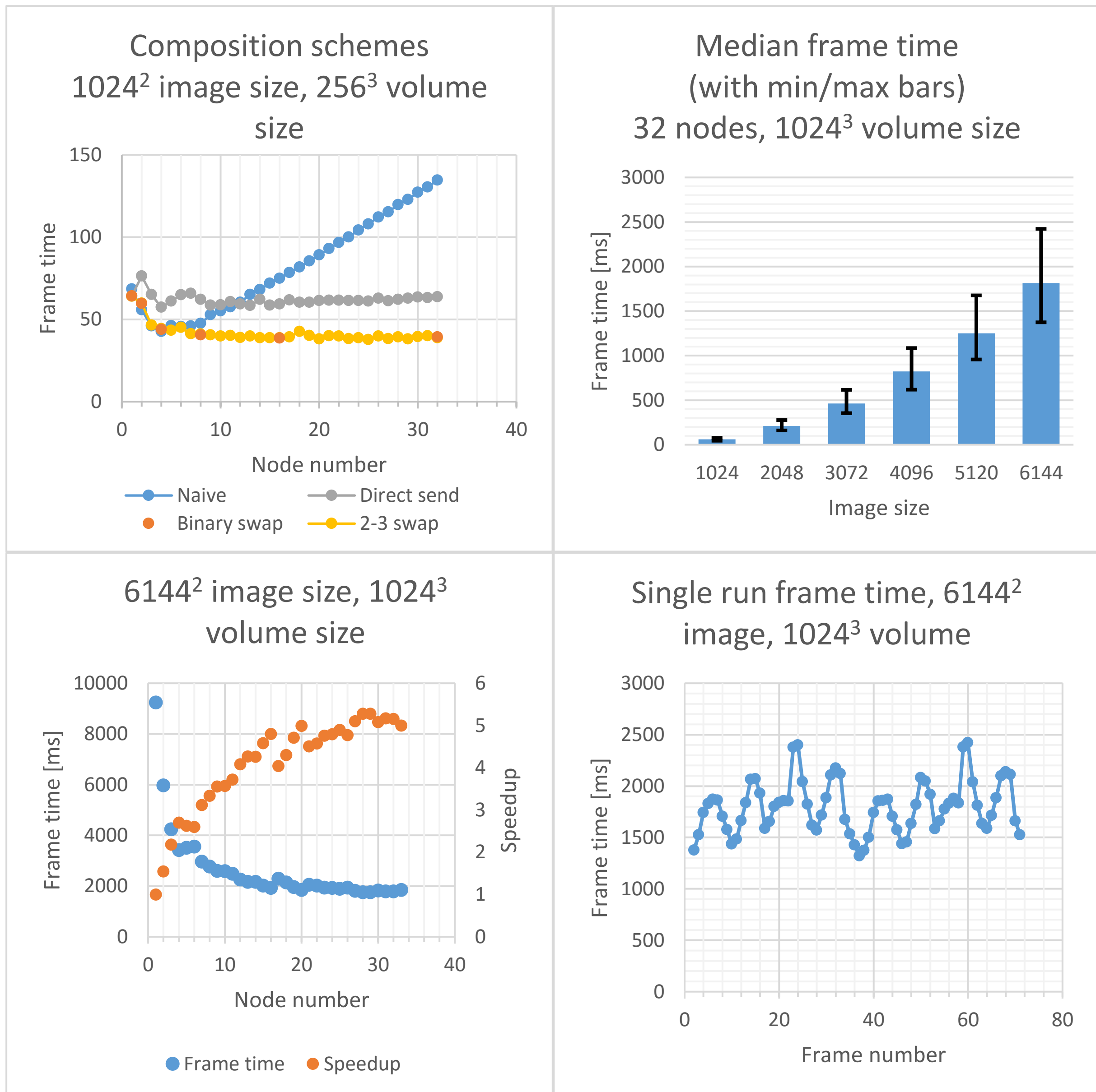

Figure 9: Overall volume renderer performance. Top-left: comparison of composition scheme performance (average frame time). Top-right: median frame time w.r.t. to image size. Error bars represent min and max frame time that occurred during the run. Bottom-left: average frame time on a large image and dataset size, 2-3 swap composition. Bottom-right: measured frame time during a single run of the renderer, with repetitive variation occurring due to camera orbiting the volume dataset.

performance. I show scaling behavior of the renderer on Figure 9 bottom-left, by plotting average frame time for large image and dataset sizes, measured with various number of nodes.

Having a basic understanding of the rendering application's scaling behavior, and

some of the factors that define it, I now begin the development of a cluster-specific performance model.

3.3 Prediction model

After obtaining the experimental data from running the volume renderer on the cluster, I can proceed to constructing a model that is capable of predicting renderer performance, both matching the training data and extrapolating to a larger number of nodes. More formally, I aim to construct a model that maps image resolution (R), volume dataset size (S), cluster node number (N) and the GPU of the nodes (G) to cluster frame time (T_c) (Equation 26). Note, that since the training data is obtained on a single cluster, the node GPU parameter is *implicit* in the model, i.e. the model represents scaling behavior of a concrete cluster.

$$(R, S, N) \rightarrow T_c \qquad (26)$$

3.3.1 Linear regression

To better understand scaling behavior of the volume rendering application, and to determine suitable complexity of a statistical model used for performance prediction, I begin by applying simple linear regression to the experimental data. I am using the implementation of linear regression provided by the scikit-learn library [34].

The training dataset includes 594 data points, obtained by running the volume rendering application with every combination of the following input parameters:

- Image resolution R ∈ $\{1024^2, 2048^2, 3072^2, 4096^2, 5120^2, 6144^2\}$
- Volume dataset size S ∈ $\{256^3, 512^3, 1024^3\}$
- Node number N ∈ {1..33}

Each data point passed as input to the model consists of four features:

- Image width
- Volume width
- Node number
- Frame time

Since I am using square image resolutions and cubic volume sizes, there is no need to pass every image/volume dimension to the model, as they all have similar values. The model is a standard linear regression, fit using Ordinary Least Squares

Figure 10: Cluster frame time prediction using a standard linear regression model.

estimator [15].

A subset of results is presented on Figure 10, having a residual MSE loss of $2.512 \cdot 10^5$ and an R^2 score of 0.66. A slight improvement could be made by introducing three new features: pixel number, voxel number and a product of the two. The logic behind it is that most volume rendering applications are limited by the memory read operations, the number of which scales linearly with the product of the number of pixels and voxels. This results in a model with a residual MSE loss

Figure 11: Cluster frames-per-second prediction using linear regression, with additional pixel/voxel number input features.

of $1.986 \cdot 10^5$ and an R^2 score of 0.73.

Noticing that the frame time graphs (Figure 10) have hyperbola-like shape suggests that I could further improve accuracy of the linear model by using frames-per-second value (i.e. the inverse of frame time) as dependent variable for the regression, since a linear hypothesis should yield better results approximating a linear function. The model resulting from this improvement is presented on Figure 11, and has an R^2 score of 0.86. (Note, that since the dependent variable has been

```
GenereatePolynomialFeatures(data, featureSet, degrees)

     foreach featureA in featureSet
        foreach featureB in featureSet
           if featureA == featureB
                 continue
           foreach degreeA in degrees
              foreach degreeB in degrees
                 newFeature = multiply(power(data[featureA], degreeA),
                                       power(data[featureB], degreeB))
                 data.insert(newFeature)


end
```

Figure 12: Algorithm for generating additional polynomial features from the original input data.

changed the residual loss cannot be meaningfully compared.)

Although I have improved the quantitative results, the graphs indicate that the linear hypothesis is far too simple to approximate the non-linear input data. The problem is further aggravated by high dimensionality and large range of the data, making a fit using a single hyper-plane particularly challenging.

3.3.2 Nonlinear extensions

To cope with non-linearity of the measured performance data, I extended the linear regression model with a data pre-processing step, during which I generate additional polynomial features based on the original data. First, I manually select a subset of features to participate in non-linearization and a set of possible degrees. Then, I take pairs of features out of the preselected subset, raise each of them to one of the degrees and multiply them together to get a new polynomial feature. This process is repeated for every possible combination of features and degrees. A more formal description is presented on Figure 12.

The resulting model is using the following input features:

- Image width
- Volume width
- Node number
- Pixel number
- Voxel number
- Product of pixel and voxel number

Which then all undergo non-linearization step with a degree set of $(-1, 1, 2)$, resulting in a total of 276 input features (for this particular model, usage of all the input features and the three degrees showed best experimental results). This data

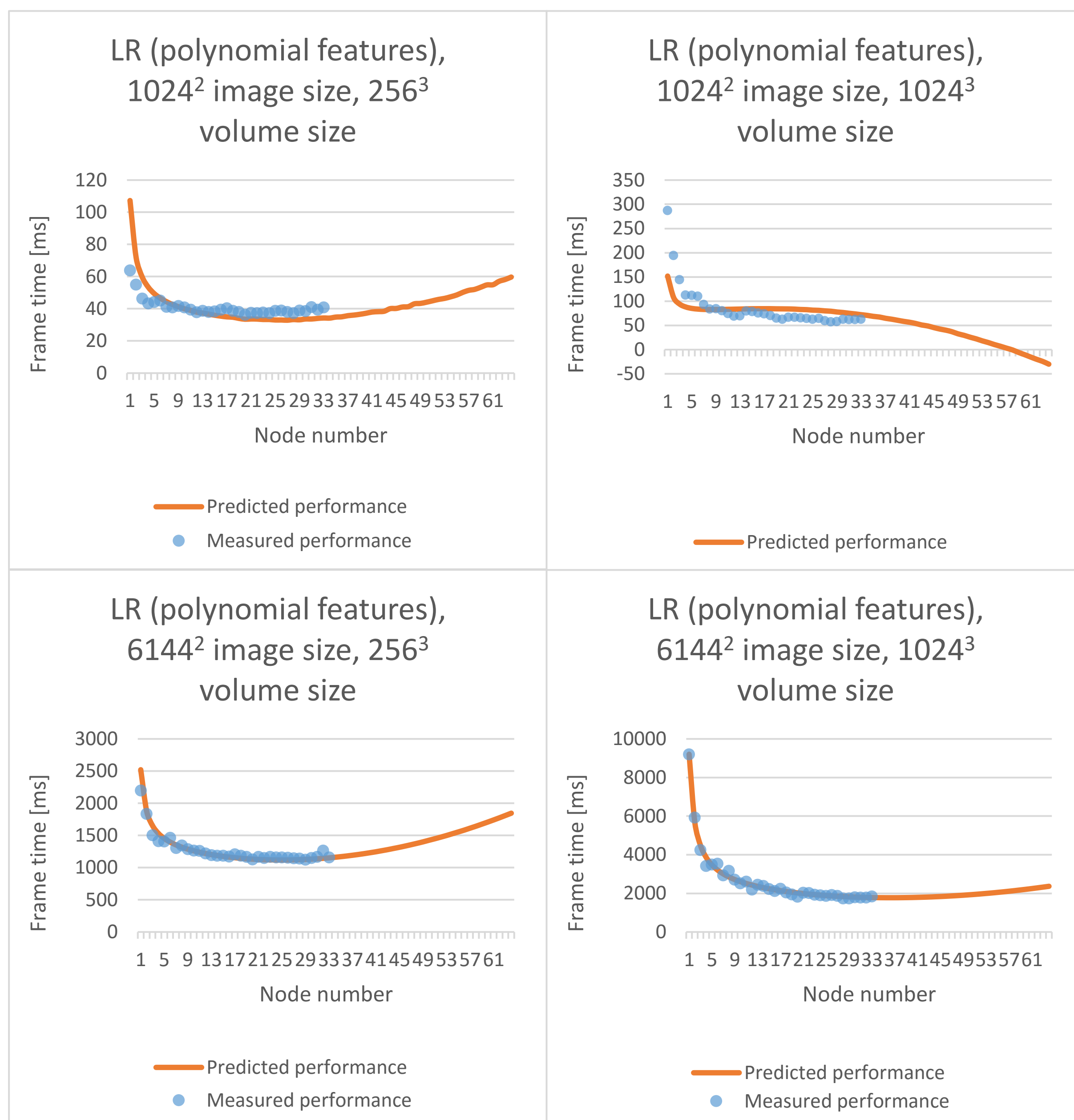

Figure 13: Cluster frame time prediction using linear regression, with additional polynomial features generated from the original data.

is then used to fit a standard linear regression model, producing a residual MSE loss of $4.978 \cdot 10^3$ and an R^2 score of 0.99. The graphs are presented on Figure 13.

Now that I have obtained a model capable of representing scaling behavior of a cluster, I need to further validate the results by testing its extrapolation properties. For this I train the model using half of the data: data points that have node number smaller or equal to sixteen serve as the training set, and rest of the data corresponding to larger cluster sizes is used for validation. The results are presented on

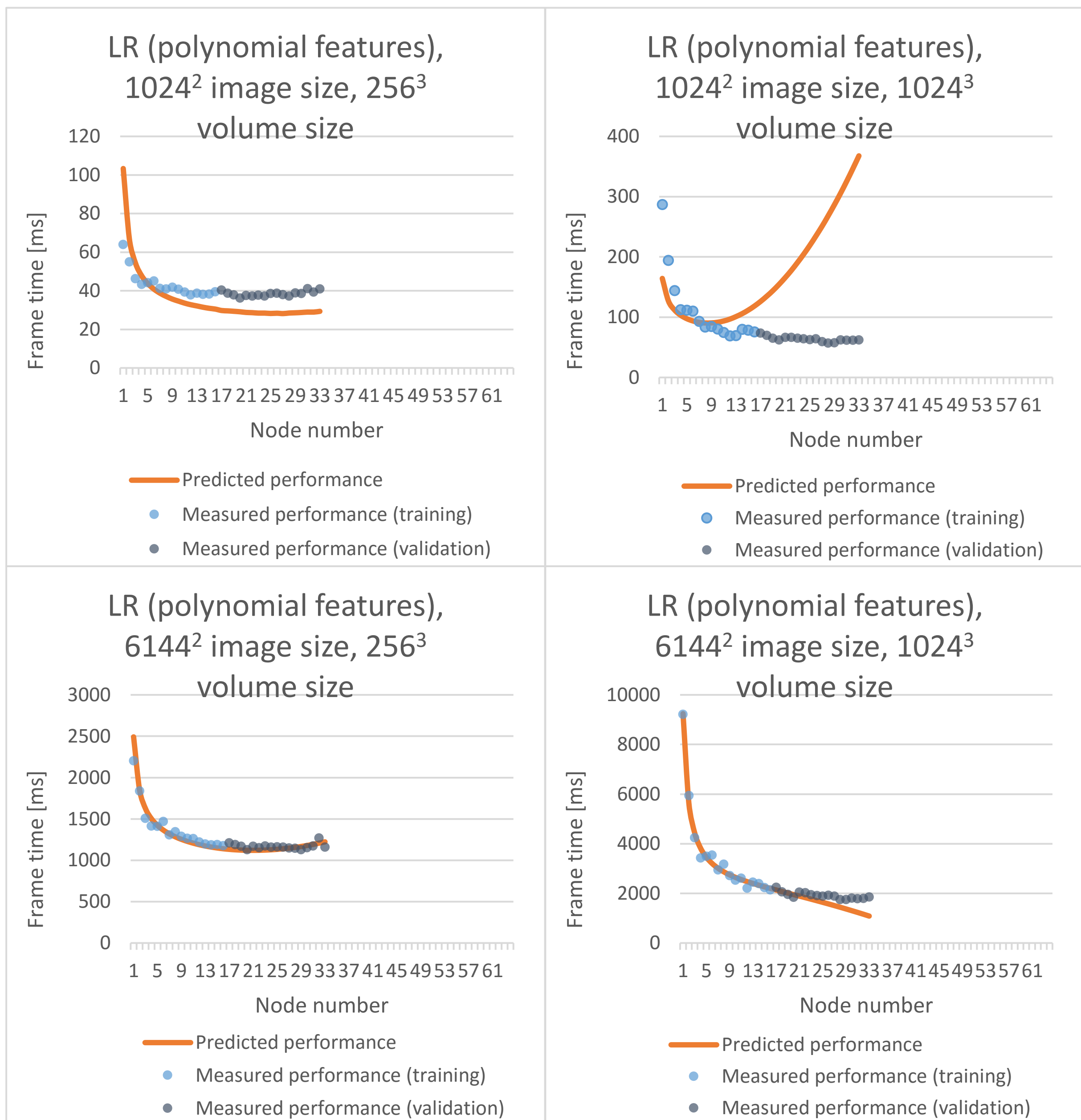

Figure 14: Testing extrapolation properties of the linear regression model with additional polynomial features: the first half of the data is used for training the model, the second half is used for validation.

Figure 14, having an MSE loss of $3.071 \cdot 10^4$ and an R^2 score of 0.89 on the validation dataset. As evident from the graphs, in most cases the model generalizes to a larger number of nodes reasonably well, but might behave erratically in some scenarios. Since no data for larger cluster sizes has been observed by the model during training, its behavior on the validation dataset can be arbitrary. The situation is somewhat alleviated by the simplicity of the model (the degree of the polynomial features cannot exceed four), thus the results are acceptable in most cases, but still

the prediction is not reliable.

Although the model developed in this section is capable of representing performance of a cluster and has some extrapolating potential, there is a limitation that prevents it from generalizing it to arbitrary clusters. As new features describing cluster hardware are introduced, not only the amount of polynomial data that needs to be generated increases exponentially, but more importantly, the results become more dependent on the choice of hyper-parameters, such as the number and degrees of the generated features, requiring manual experimentation and feature selection. Thus, I continue with the search for a cluster scalability model that both matches the results of linear regression models and does not require the manual error-prone process of feature selection.

3.3.3 Neural network

To address the shortcomings outlined in the previous section, I move to using neural networks as a foundation of the model. Neural networks are capable of approximating functions of higher complexity without the need to manually select the input features and generate their polynomial combinations. I am using implementation of neural networks provided by the *keras* library [35], which in turn relies on the *Theano* framework [36].

To begin with, I construct a basic fully-connected neural network, consisting of a single hidden layer that has 16 neurons using *tanh* activation function. Since, in contrast to linear regression, I have less control over the complexity of the model, I add L2 regularization as means of managing potential overfitting. I use the same dataset comprising 594 data points, but with the following input features:

- Image width
- Image height
- Node number
- Volume width
- Volume height
- Volume depth

The results after 50000 epochs of training with RMS propagation are presented on Figure 15 with a residual MSE loss of $8.649 \cdot 10^3$ and an R^2 score of 0.99. As can be seen, this neural network is capable of representing the scaling behavior of the cluster without overfitting the training data, thus matching the results achieved with linear regression in section 3.3.2. However, in case of the neural network I do not need to manually choose the feature subset or generate polynomial features, making further generalization of the model easier.

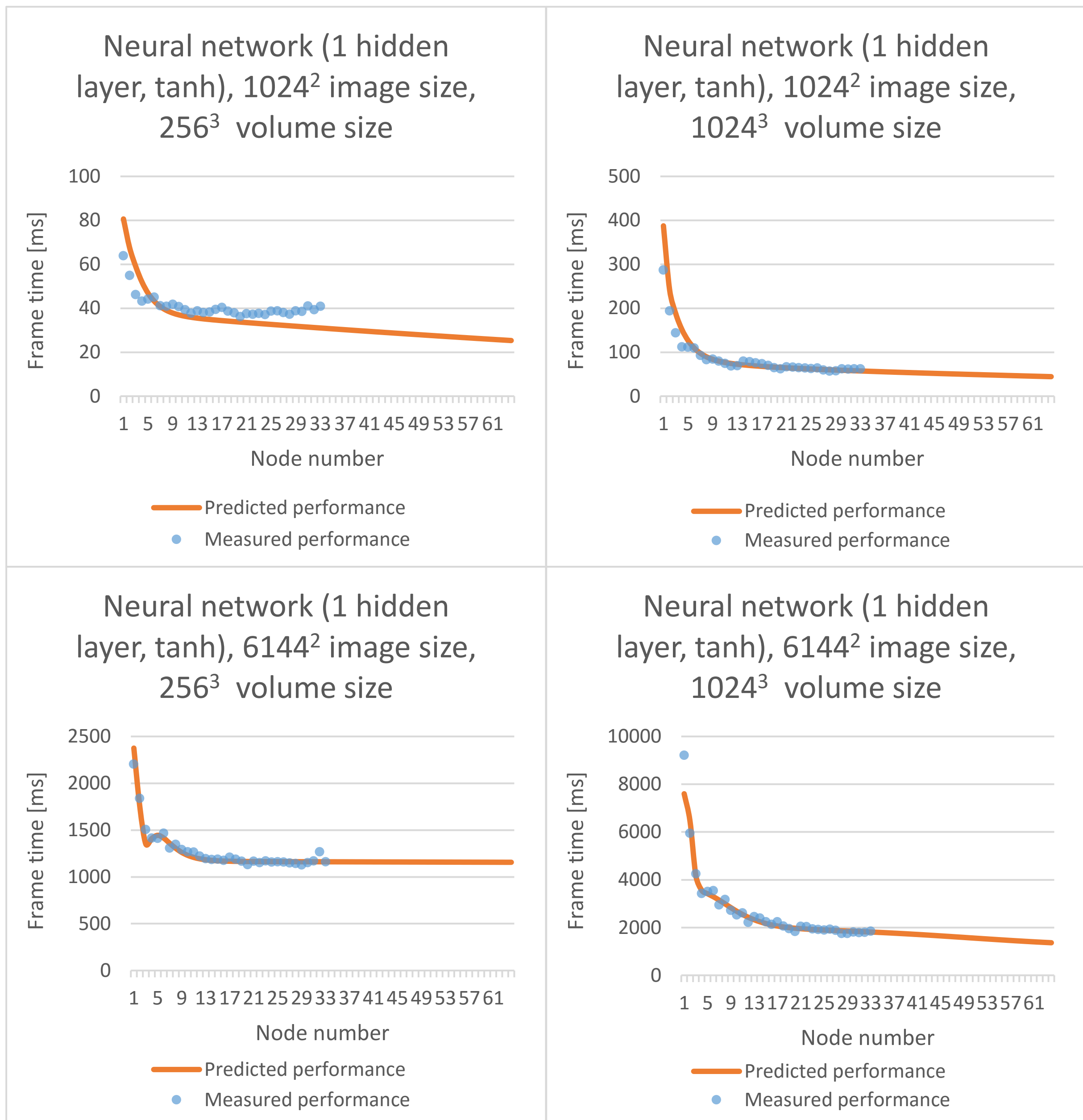

Figure 15: Cluster frame time prediction using a neural network. The network comprises one hidden layer of 16 neurons with 'tanh' activation and L2 regularization.

In line with the overall goal of the project, having constructed a neural network model that matches the accuracy of the earlier linear regression model, I proceed to testing its extrapolation properties. I repeat the experiment, using half of the data: data points that have node number smaller or equal to sixteen serve as the training set, and rest of the data corresponding to larger cluster sizes is used for validation. The result can be seen on Figure 16 with an MSE loss of $4.502 \cdot 10^4$ and

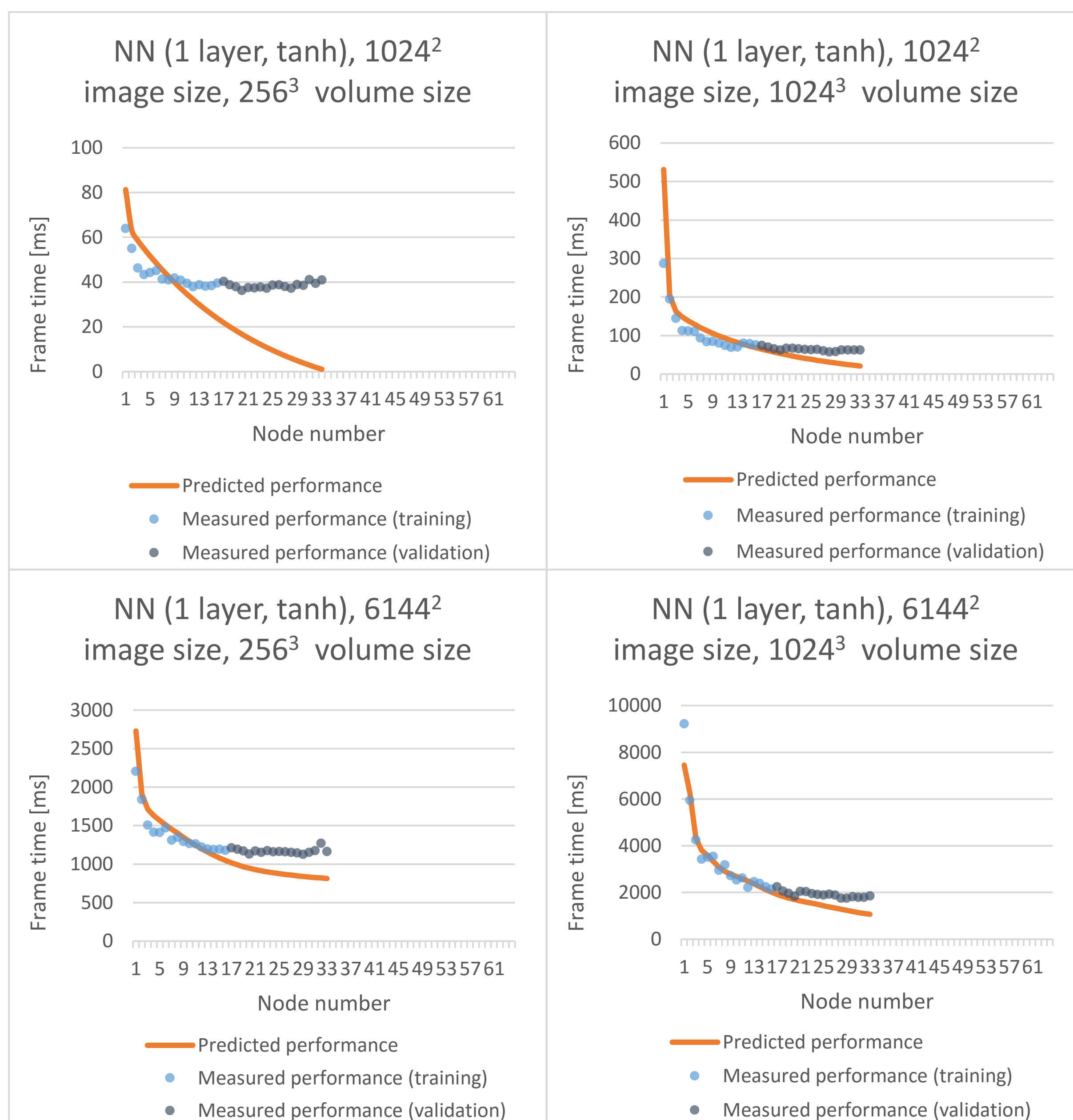

Figure 16: Testing extrapolation properties of the single-layer neural network model: the first half of the data is used for training the model, the second half is used for validation.

an R^2 score of 0.84 achieved on the validation dataset. As with the linear regression model, despite the overall accuracy of the prediction, the model has large error in some of the cases. More importantly, it doesn't capture the general trend of the scalability curves, which tend to reach a plateau or even degradation of performance after a certain number of nodes has been reached. Ability to model this behavior is important for prediction of an optimal cluster size for a particular volume rendering application.

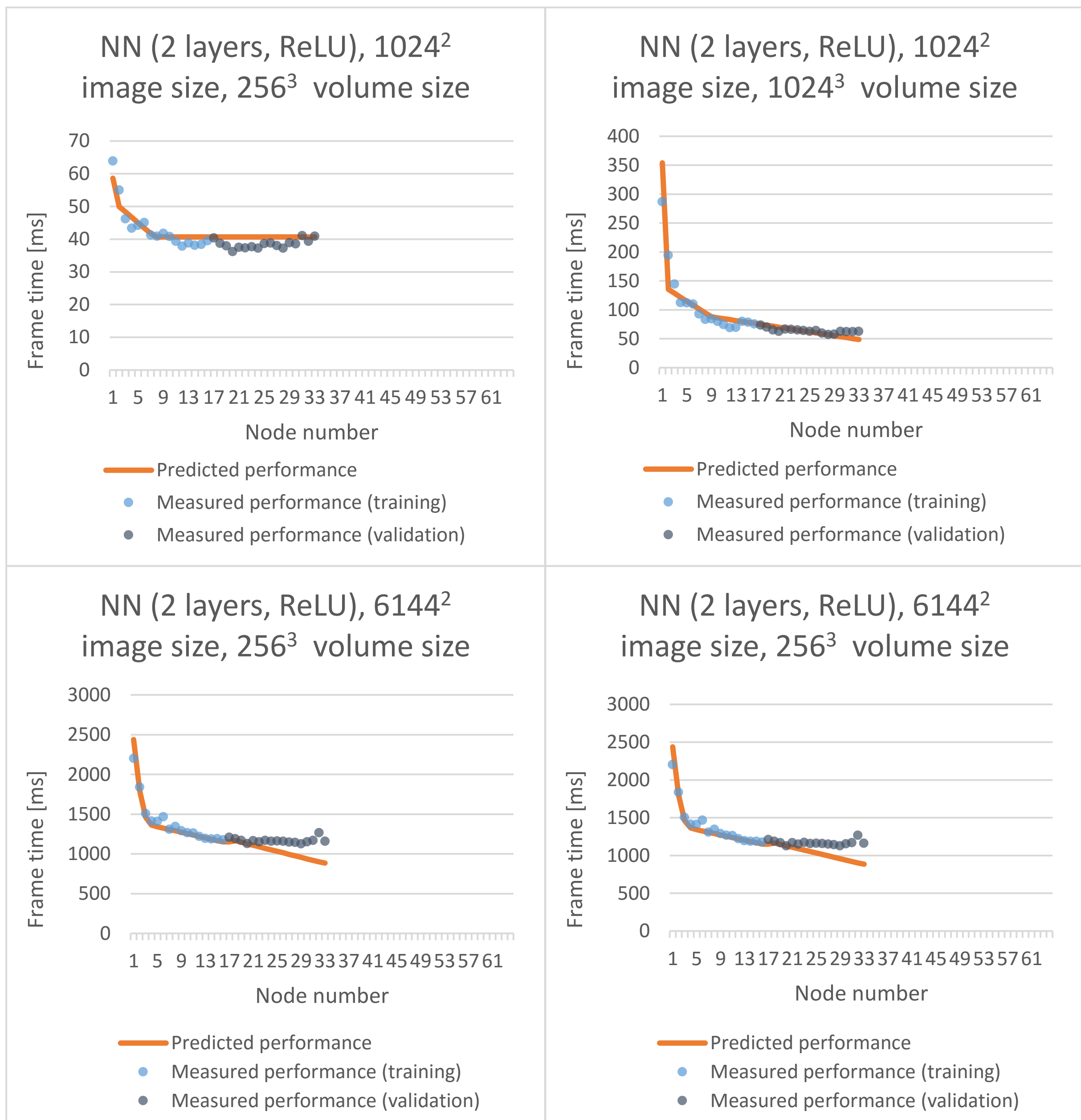

Figure 17: Testing extrapolation properties of the neural network model: the first half of the data is used for training the model, the second half is used for validation. The network uses 2 hidden layers of neurons with ReLU activation.

An improvement that can be made to the neural network is to use an unbounded activation function, such as ReLU (Rectified Linear Unit) coupled with strong regularization to allow the output function to better ‘extend’ into regions outside of the training data. Experiments have also shown that I need to add another hidden layer to the network, to avoid piece-wise linear character of the resulting prediction curve, caused by ReLU activation. Figure 17 depicts extrapolation results of a network comprising two hidden layers of 16 and 8 neurons with ReLU activation

Technique	**MSE loss**	**R^2 score**
Fitting the whole training set		
Standard linear regression (LR)	$2.512 \cdot 10^5$	0.66
LR with pixel/voxel number features	$1.986 \cdot 10^5$	0.73
LR with pixel/voxel number features, predicting FPS	n/a	0.86
LR with generated polynomial features	$\mathbf{4.978 \cdot 10^3}$	**0.99**
Neural network, 1 layer, tanh activation	$8.649 \cdot 10^3$	**0.99**
Extrapolation to a larger number of nodes, trained with half of the data		
LR with generated polynomial features	$3.071 \cdot 10^4$	0.89
Neural network, 1 layer, tanh activation	$4.502 \cdot 10^4$	0.84
Neural network, 2 layers, ReLU activation	$\mathbf{2.141 \cdot 10^4}$	**0.93**

Table 1: comparison of developed performance prediction models. First, w.r.t. to their ability to fit the whole dataset (top). Next, w.r.t. to their ability to extrapolate performance to a larger number of nodes, being trained with only half of the data (bottom).

and L2 regularization (lambda of 1.0); on the validation dataset, the network achieves an MSE loss of $2.141 \cdot 10^4$ and an R^2 score of 0.93.

As can be seen, the model is capable of limited extrapolation, however, as it is sampled further away from the training data, the quality of prediction degrades significantly. The effect is further aggravated when I decrease the maximum number of nodes used for generation of the training data, which presents a problem for further generalization of the model, i.e. being able to predict performance of the whole cluster from data obtained on just a single node.

Neural networks are inherently an *interpolation* (approximation) technique and, notwithstanding limited potential improvements, are ill-suited for *extrapolation* purposes. In order to generalize a neural network model to arbitrary clusters I need to obtain training data from executing the volume renderer on multiple clusters. This, however, presents a problem, since I only have access to a single cluster and am unable to collect the data for training a general model. A solution lies in changing the way I define the cluster scalability model and separating parts of it that I *can* train using data obtained on a single cluster. This process is presented in more detail in the next chapter.

4. Hardware-agnostic cluster scalability model

The purpose of the work presented in this chapter is to construct a general model capable of predicting performance of a distributed volume rendering application on an arbitrary cluster given information about the rendering task and the cluster hardware. In more formal terms, I seek a model that would map image resolution (R), dataset (D), number of nodes in the cluster (N) and the GPU of a single node (G) to cluster render time (T_c).

$$(R, D, N, G) \rightarrow T_c \tag{27}$$

Although the neural network model presented in section 3.3 shows reasonable performance at *interpolating* (technically, approximating) the experimental data, its shortcomings become apparent when doing *extrapolation*, evaluating the model for input points that do not lie in the vicinity of the training data. Thus, to train a model capable of predicting performance on a wide range of hardware I need to perform measurements on that hardware, and then feed the resulting data to the model. The more general I want the model to be, the more hardware I would need for collection of training data. This presents a problem, since there are only two clusters at my disposal, one of which is reserved for model validation, leaving just one.

A solution to this problem is to abstract away the application-specific details of local computation and only model the scaling behavior of the cluster, i.e. the communication overhead. This way I can encapsulate the hardware characteristics of a node and application-specific parameters such as dataset size into a 'local computation time' input variable. More specifically, instead of the model from Equation 27, I use a model mapping node number, image resolution and local node render time (T_n) to cluster render time (Equation 28). Although image resolution could be considered an application-specific parameter, it also determines the communication load and thus remains among the model parameters.

$$(R, N, T_n) \rightarrow T_c \tag{28}$$

Since the model no longer depends neither on the hardware of a cluster node, nor on application-specific parameters, I can simulate local computation phase by simply stalling for a specified amount of time on each node of the cluster. The composition step is performed as usual, because the overhead introduced by the communication is precisely the effect that I am trying to predict. Varying the local computation time parameter, I can perform a large number of experiments, simulating

various hypothetical node hardware (but not network hardware: since experiments are run on a single cluster, the model implicitly assumes a specific network hardware and topology).

It is important to note, that the local computation time parameter does not represent time it takes to perform the whole computation on a single node (a 'single-node cluster' time). It represents the time it takes for a single node of a cluster, which comprises N nodes, to perform its part of computation. So, this parameter is influenced by both the number of nodes and the partitioning scheme used by the application.

Furthermore, the local computation time is not a scalar but a vector of values, each corresponding to one of the nodes of the cluster. Usage of a vector allows for modeling load imbalance that occurs in a distributed application and might have significant impact on the performance of the cluster.

The decision of using local computation time as an input parameter for the model has not only been made out of necessity of acquiring a larger training dataset, but also in consideration that the values of this parameter for concrete rendering applications can be measured on a single node of a cluster. This holds true even though a vector of values might need to be obtained: local computation time of a node does not depend on results from other nodes, thus the computation of each node of the cluster can be performed and measured sequentially on a single machine. Additionally, the values could also be predicted by yet another statistical model that maps application-specific details and node hardware to local computation time:

$$(R, N, D, G) \rightarrow T_n \tag{29}$$

Combining it with the hardware-agnostic cluster model (Equation 28) would result in a complete cluster model (Equation 27). I also discuss this possibility in the future work section (Chapter 5).

4.1 Rendering simulation

Switching to the hardware-agnostic cluster model requires modifications to the distributed renderer application. The renderer should be able to take a vector of local computation time values as input, and simulate local computation by stalling a node for a specified amount of time.

Although the new cluster model takes abstract computation time as input and uses

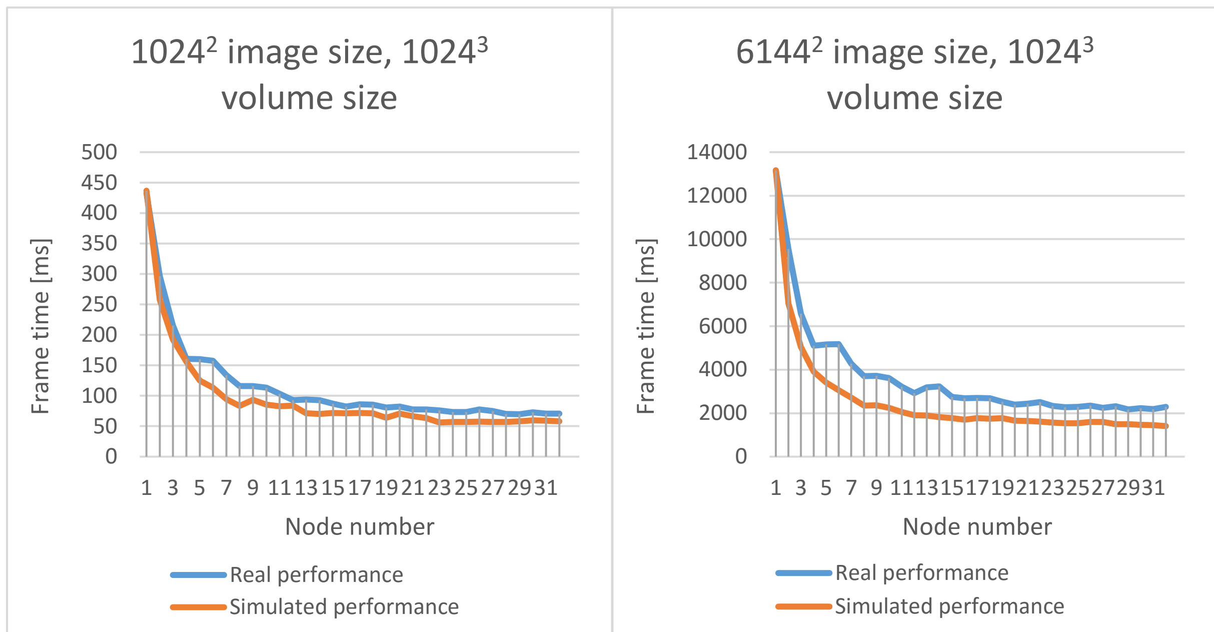

Figure 18: Comparison of real and simulated performance. Simulation using local computation time vectors, with each node stalling for the average amount of time taken by this node during a real renderer run.

data obtained through simulation, it should still serve the underlying goal of predicting performance of concrete rendering applications. Therefore, I must make sure that the simulation technique accurately mimics the performance of concrete rendering applications, so that when the model predicts performance of a simulation, the result can be used for actual applications.

To this end, I have measured local computation time vectors by running a concrete volume rendering application on a cluster. Specifically, for each combination of renderer input parameters (node number, image size, dataset size) I have performed a renderer run, during which 72 frames have been rendered, while orbiting the camera twice around the dataset. For each node, its local computation time has been measured and averaged over all the frames, resulting in a vector of N values. Then, this vector is used to perform a simulation run under the same conditions, the only difference being that no actual local computation is performed and the nodes are stalling instead.

This experiment allows collecting the cluster performance of both a real and a simulated rendering application, and assessing the accuracy of the simulation technique. The results are presented on Figure 18. Albeit the curves are similar in character, there is a severe disparity between the simulated and the actual performance, which is especially pronounced with larger image sizes. Most importantly, the simulation does not reproduce the characteristic ‘steps’ in performance of a

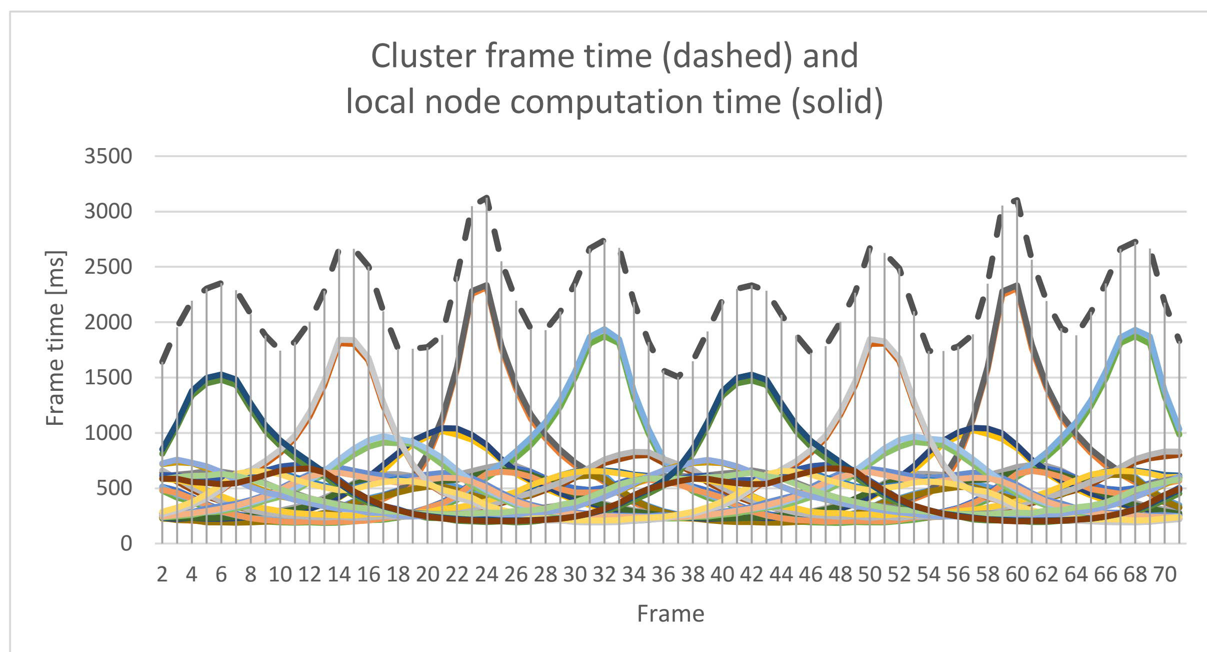

Figure 19: Local computation time of each node of the cluster during a single run (solid lines). Note how overall cluster frame time (dashed line) depends on the slowest node.

real application, which are an important feature of cluster scaling behavior.

The problem occurs because the technique does not properly simulate load imbalance. In a distributed rendering application with static object-space partitioning every node is assigned a fixed volume brick, which does not change during execution. As the perspective-transforming camera orbits the volume, distance from a volume brick to the image plane changes, so does its footprint, i.e. the number of screen pixels covered by the volume brick. Increase in the footprint causes an increase in node's local computation time, affecting the overall load balance. Profiling of the rendering application shows (see Figure 19) that the overall cluster frame time is defined by a few nodes with large load, as the whole cluster must wait for all the nodes to finish their local computation, before proceeding to the composition phase.

When using average node time for simulating local computation this load imbalance effect is lost, since every node takes the same amount of time each frame to perform its computation. The net result is that the cluster frame time of a simulation is lower than that of a real rendering, which is precisely the problem seen in Figure 18.

Since the overall cluster frame time is defined by the node that takes the longest to perform its local computation, a naïve attempt at improving the current technique would be using maximum and not average local computation time for each

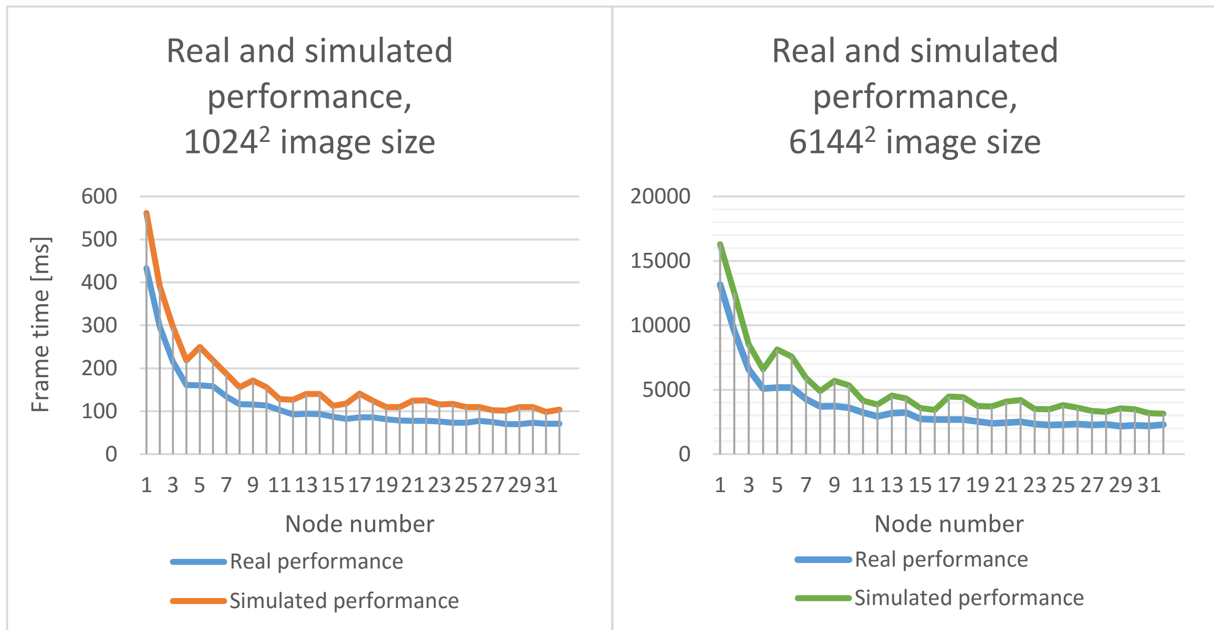

Figure 20: Comparison of real and simulated performance. Simulation using local computation time vectors, with each node stalling for the maximum amount of time taken by this node during a real renderer run.

node. More precisely, when constructing a local computation time vector, instead of averaging computation time of a node over all frames, take the maximum computation time that occurred on that node during a run of the renderer.

The results of a simulation with such parameters are presented in Figure 20. Every node of the cluster during every frame simulates its own worst-case scenario, i.e. conditions under which its brick has the largest footprint. This results in overall cluster frame time being exaggerated compared to the baseline of the real measured performance. Nevertheless, this technique has smaller absolute disparity than the average node computation time approach, and has yet another advantage: it preserves and amplifies the effects of the communication overhead on the overall frame time. One can see the 'steps' in the frame time graph, which occur due to interplay between cluster size and the "2-3 swap" composition scheme, with frame time being lower for cluster sizes resulting in favorable communication patterns, e.g. power-of-two cluster sizes (see section 2.3 for the description of the "2-3 swap" composition scheme).

After studying the results of the above-mentioned simulation approaches, it becomes evident that to eliminate the cluster frame time disparity between the real and the simulated performance, I need to move away from using a static predefined amount of time that a node is being stalled for during the whole simulation run, and instead use a representation of local computation time that is capable of

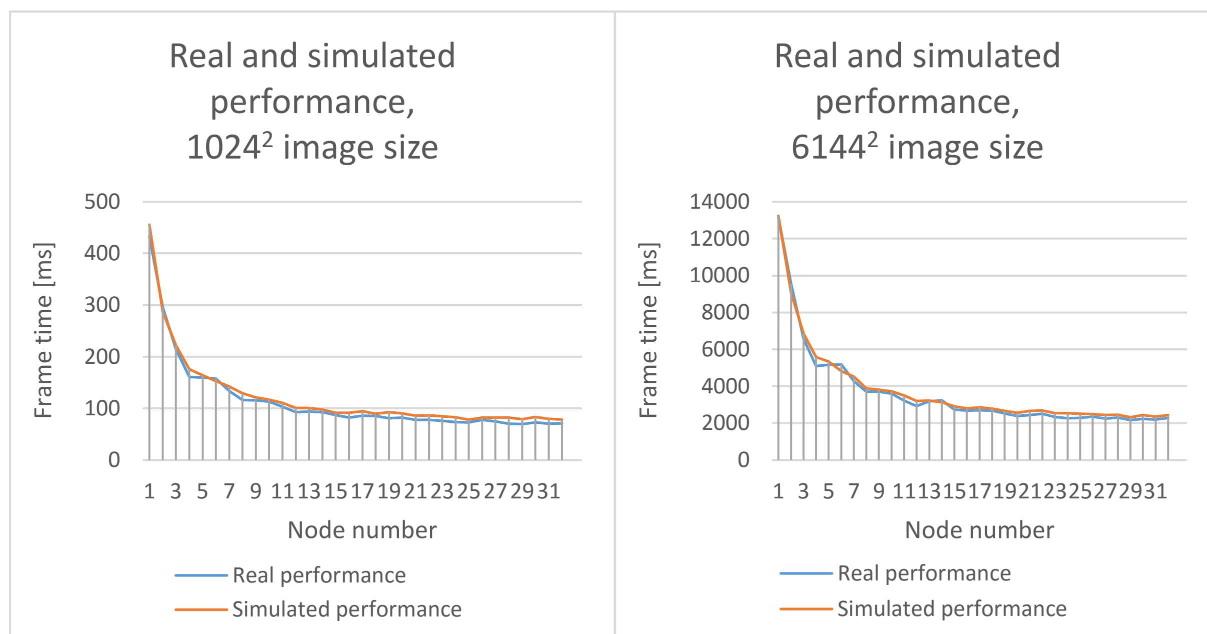

Figure 21: Comparison of real and simulated performance. Simulation using local computation time distributions, with each node sampling the distribution during runtime to determine the amount of time it should stall.

capturing dynamic load imbalance present in a given rendering application.

One such representation is *a distribution* of local computation time, which can be expressed as a histogram and randomly sampled during simulation, to replicate the varying character of local computation time and maintain an approximate proportion of load-heavy nodes during each frame.

To collect this local computation time histogram, I have run an actual rendering application on a cluster. (Remember, although not implemented in this project, local computation time can be measured even on a single machine, since it does not depend on inter-node communication.) Then, the observed local computation time of each node, during each frame, is interpreted as a sample of a random variable that is local computation time. Next, all these samples are aggregated into a histogram of M bins, which is subsequently normalized to represent a probability distribution. The histogram is stored as M bin values, together with a minimum and a maximum local computation time values that capture the domain of the histogram. Thus, for each run of the renderer, i.e. for each combination of renderer input parameters (node number, image size, dataset size) I obtain a local computation time histogram, which captures the unique load balance conditions that occurred during the run.

During the simulation, the local computation time histogram is sampled once for

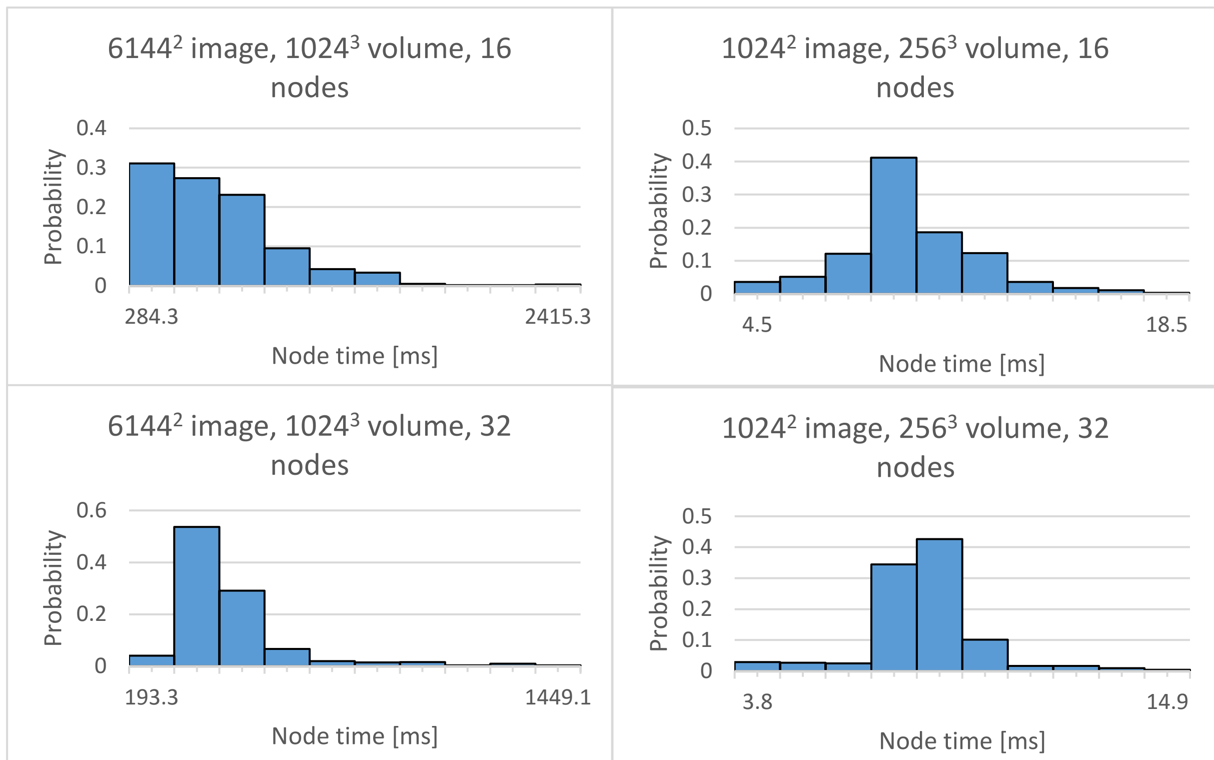

Figure 22: Distributions of local computation time measured by running an actual rendering application.

each node, to decide how long the node will stall during the current frame. The histogram simulation approach (Figure 21) has a very small disparity between the real and the simulated results, and thus, it is suitable for the purposes of decoupling the cluster scalability model from cluster node hardware. It allows simulation of different hardware and rendering applications to obtain more training data for the prediction model.

4.2 Prediction model

Now that I have defined the hardware-agnostic scalability model and have developed a reasonably accurate simulation technique, one step remains before training the model: I need to use the simulation to obtain more training data to make sure that the model 'learns' general effects of cluster communication, and not peculiarities of a particular cluster or application. Measurement of additional training data in turn requires more node time histograms. The histograms can be obtained either by measuring them on different hardware, or constructing artificially, representing some hypothetical combination of node hardware and a rendering application. I follow the latter approach, since it allows usage of the existing hardware

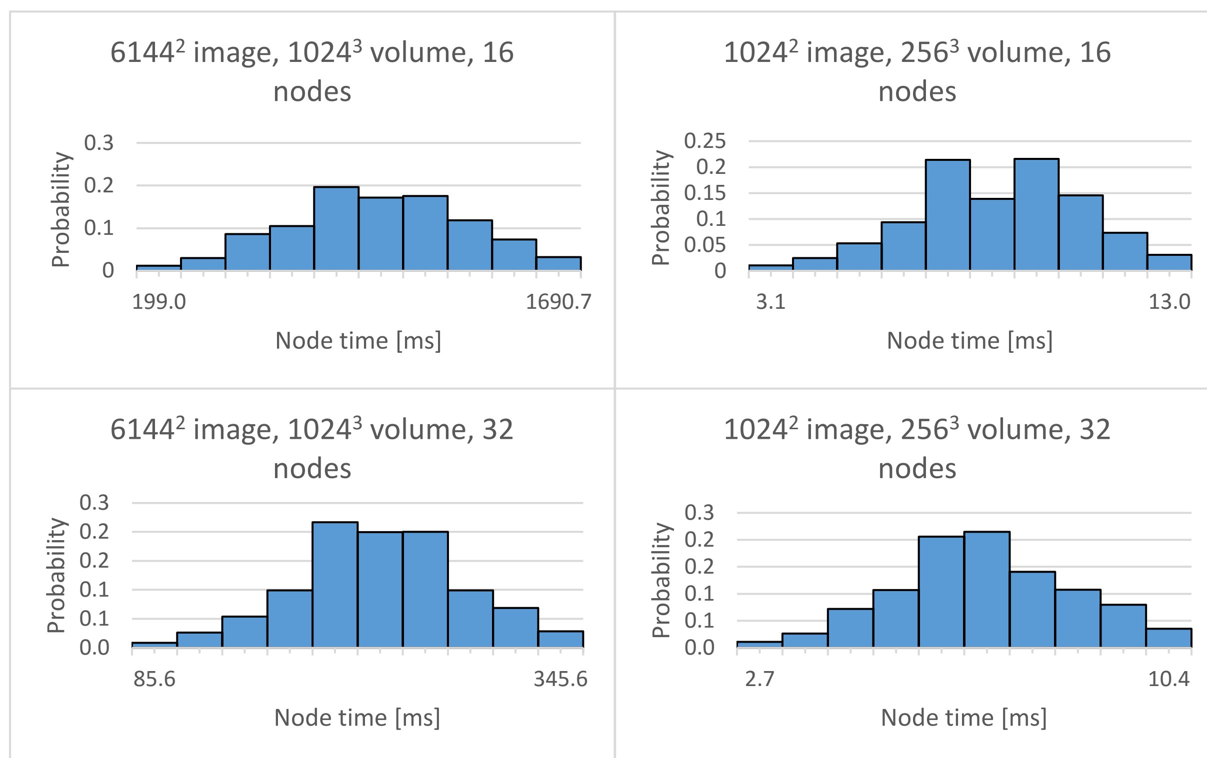

Figure 23: Artificially generated distributions of local computation time. A Gaussian with mean of 0.5 and standard deviation of 0.2 is sampled, and perturbed with a uniform distribution in the range (-30%,30%), to imitate histograms measured by running an actual rendering application.

and does not require any modifications to the rendering application.

4.2.1 Artificial generation of node time histograms

Before I begin generation of artificial node time histograms I first need to examine the histograms obtained from running an actual volume rendering application, presented on Figure 22. As seen from the graphs, most distributions produced by volume rendering tend to have a single peak, resembling normal distribution.

Although it is not strictly necessary to give any particular shape to the generated node time histograms, as I could attempt to train a neural network that can predict performance for any possible distribution, this would require generation of large amounts of training data, for many possible distributions. Instead, knowing that the goal is to predict performance of volume rendering, I generate only a few histograms, which resemble distributions observed while running the renderer.

Specifically, when generating artificial histograms, I sample a Gaussian curve centered near the middle of the histogram (mean of 0.5, relative to the histogram do-

Figure 24: Evaluation of the final model on the validation dataset, which was measured on a different cluster. Best out of three runs, with MSE loss of $5.127 \cdot 10^3$ and R^2 score of 0.96.

main; standard deviation of 0.2), and then perturb it slightly using a uniform distribution in the range (-30%, 30%). To define the domain of the histogram, i.e. the minimal and maximal local computation time, I simply take the value observed during a real renderer run under the same input parameters (image/volume size, node number) and scale it by a constant. Note, that the details of this method are fairly arbitrary, but ultimately not very important: I only aim to provide variation in the input features, so that the model can be trained to predict their effect on the overall cluster time. Examples of generated histograms can be seen on Figure 23.

Figure 25: Learning curves, showing how accuracy of prediction changes with the increase of the training set size.

I repeat this procedure for every possible combination of renderer input parameters, thus generating 594 node time histograms, which will be used in the simulation to measure 594 data points, with a full run of the rendering simulation. I refer to this set of data points as the *generated-histogram dataset*.

There exists yet another way of increasing the amount of training data I have. After changing the model to represent only the communication overhead of a cluster I do not need to treat the number of GPUs on a node as an input parameter for the model, since it is abstracted away behind the local computation time histograms. Now I can perform measurements on the 'training cluster' twice: once in double-GPU mode, and once in single-GPU mode, effectively giving me two sets of node time histograms. I can then use these histograms to perform simulation on the cluster, obtaining another two datasets, representing performance of two different clusters. I refer to these datasets as *measured-histogram datasets*.

4.2.2 Final model

The final model uses a neural network consisting of two hidden layers of 16 and 8 neurons with ReLU activation and L2-regularization. The model uses the following input features:

- Image size (width, height)
- Node number

Figure 26: Evaluation of the final model without the 'bin features' of node time histograms, to check if the model utilizes the node time histograms in its prediction. MSE loss of 2.179·10^4 and R^2 score of 0.83

- Avg., min and max node time (histogram domain)
- Ten histogram bins (histogram values)

Out of the *generated-histogram dataset* and two *measured-histogram datasets*, described in previous section, I use two datasets for training, reserving one measured-histogram dataset as a *test dataset* for automatically choosing the best value for the L2-regularization parameter. For final validation, I have also performed

measurements on a different cluster. Since the goal is to validate the quality of the developed performance prediction technique, not just how well the model predicts the simulation performance, I use the measurements from a different cluster directly, without putting them through simulation.

The results are presented on Figure 24. As can be seen, the model exhibits some jittering when predicting performance for smaller image/volume sizes, but the overall accuracy is reasonable, with an MSE loss of $5.127 \cdot 10^3$ and an R^2 score of 0.96 on the validation dataset. I also construct learning curves, plotting MSE and R^2 score against the training set size on Figure 25. As more data is added, one can see an expected increase of the error on the training set (as it gets harder to fit more data points) and an improvement in accuracy on the validation data set (as the model becomes more general with more data), confirming that adding additional simulation data improves the prediction.

Another interesting test to perform is to try removing the 'bin features' of the node time histograms from the input data, i.e. training the neural network which relies on node number, image size and min/max/avg. node time, to see if the final model is actually using the load balancing information contained in the histogram to improve the prediction. The results are shown on Figure 26, with an MSE loss of $2.179 \cdot 10^4$ and an R^2 score of 0.83 on the validation dataset. The model displays less jittering, trading off higher variance for higher bias, but resulting in overall lower accuracy. This result suggests that the model does rely on the node time histograms, when predicting the overall performance of the cluster.

5. Conclusions and future work

5.1 Summary

In this work, I aimed to construct a performance model that can predict performance of a parallel volume rendering application running on a cluster, given data obtained on one of its nodes.

To better understand the cluster performance and assess suitability of various machine learning methods, I began by studying and predicting performance of a single concrete cluster. First, simple *linear regression* has been considered. However, the model turned out to be far too simple for high-dimensional performance data that also deviated too much from the theoretical best-case scenario of linear performance scaling. Several slight improvements have been made, such as introducing extra features proportional to the number of GPU memory accesses, and using frames-per-second instead of frame time as the target variable for regression, but none of them yielded acceptable results, suggesting that the complexity of the model needs to be increased.

Next, I have extended the linear model with automatic generation of *polynomial features* from the original data, using linear regression solver to fit a non-linear function to the data. Careful adjustment of the model's hyper-parameters (such as the number and the degrees of generated features) yielded a model capable of representing performance of a single cluster. However, the process of manually selection model's hyper parameters proved to be cumbersome and ill-suited for datasets with a larger number of features (which would be inevitably introduced to generalize the model to arbitrary hardware). This motivated a search for a more automated data-driven technique, leading to investigation of neural networks.

Neural networks were utilized to train a model that matches the accuracy of linear regression techniques, while simultaneously not requiring feature generation and manual adjustment of many hyper-parameters. Yet, despite the benefits of neural networks, they also hold a disadvantage: unlike the linear regression model they exhibit very erratic behavior 'outside of training data', which makes it hard to apply them to performance extrapolation, and generalize to arbitrary hardware having only a small amount of training data. An obvious solution is to perform measurements on many different clusters, but only a single cluster was available for running the experiments. For this reason, before approaching generalization of the model, the cluster scalability model had to be reformulated.

Initially, I sought a holistic model that would map cluster size, node hardware and application-specific parameters (dataset size, image size, etc.) to overall frame time of the cluster. To address the training data problem, hardware and application-specific parameters have been abstracted away behind a *local computation time* parameter, which represents how much time each node of the cluster requires to perform its local rendering. This allows to avoid local rendering completely when running experiments, simply stalling the nodes instead. Various hypothetical combinations of hardware and application parameters can be simulated by choosing different local computation time values. And, the effects of communication and composition algorithms are still captured in the training data, enabling training of a model that predicts the 'cluster overhead'. Furthermore, the local computation time can be measured on a single node of a cluster by sequentially timing the rendering of each volume partition.

Noticing in simulation results, that the local computation time does not faithfully represent the load imbalance inherent for parallel volume rendering, it was reformulated into *local computation time histograms*, which represent a probability distribution of local computation time, rather than its average value. Sampling this distribution to determine how much each node of the cluster should stall during the local computation phase allows for situations, where a few nodes take significantly longer time to render their volume partitions than others, mimicking load imbalance.

Using this simulation technique, I have gathered more training data and trained a neural network, which is capable of predicting cluster performance. The model has been tested with a validation dataset acquired on a cluster with different node hardware. Although the model displays some jittering on smaller problem sizes, it shows high overall accuracy, achieving an MSE loss of $5.127 \cdot 10^3$ and an R^2 score of 0.96 on previously unobserved data.

5.2 Current limitations and future work

I proceed to discuss current limitations and potential further improvements of the model. First, as mentioned before, the prediction on smaller image and volume sizes shows some jittering. Although usage of percentage-based loss functions, or dependent variable normalization somewhat alleviates the problem, it results in overall less accurate prediction, which is not acceptable. Looking for another solution, one can note, that removing the histogram 'bin features' yields a more stable

prediction curve, albeit less accurate. This warrants a further investigation of histogram representations, perhaps one could find a representation that is more suitable for a neural network, to give the result more stability, while maintaining its accuracy.

Second, the current model implicitly assumes 2-3 swap being used as the composition algorithm. As the rendering application already implements several other composition schemes, one could collect more data and try to generalize the model to accept composition scheme as one of its parameters.

Perhaps, it is even possible to devise a general way of succinctly representing the communication pattern of various composition schemes, and simulate their performance, similarly to how local computational load is captured in a node time histogram. This could allow an extension of the model from covering a fixed set of parallel volume rendering algorithms, to other parallel applications.

Another direction, in which further work could be conducted, is to create a more holistic model of parallel volume rendering performance. Currently, the model uses distributions of local computation time as its input, focusing on predicting communication overhead of the cluster. One could construct yet another model, which would predict local computation time, based on application-specific parameters, such as dataset size, transfer function, and hardware characteristics of a node. Then, the two models could be combined into a single holistic model of parallel volume rendering performance, that could predict performance of a cluster based solely on *a priori* data, without performing any measurements, even on a single node of the cluster.

Overall, performance prediction remains a challenging, but rewarding problem, due to its many useful applications in systems design, equipment procurement and performance optimization. Numerous other prediction techniques have been successfully employed for HPC applications, and might be applicable to parallel visualization domain. In this work, I have focused on machine learning. Although machine learning methods, especially neural networks, at time might feel as a 'black box', I have shown, that through careful iteration they can be successfully applied to parallel volume rendering, and yield useful results.

References

[1] R. A. Drebin, L. Carpenter and P. Hanrahan, "Volume rendering," in *ACM Siggraph Computer Graphics*, 1988.

[2] T. Peterka, H. Yu, R. B. Ross, K.-L. Ma and others, "Parallel Volume Rendering on the IBM Blue Gene/P.," in *EGPGV*, 2008.

[3] J. T. Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, 1986.

[4] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 1994.

[5] L. A. Westover, "Splatting: a parallel, feed-forward volume rendering algorithm," 1991.

[6] R. Fernando, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Pearson Higher Education, 2004.

[7] Y. Heng and L. Gu, "GPU-based volume rendering for medical image visualization," in *Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2005.

[8] T. Porter and T. Duff, "Compositing digital images," in *ACM Siggraph Computer Graphics*, 1984.

[9] S. Molnar, M. Cox, D. Ellsworth and H. Fuchs, "A sorting classification of parallel rendering," *IEEE computer graphics and applications,* vol. 14, pp. 23-32, 1994.

[10] W. M. Hsu, "Segmented ray casting for data parallel volume rendering," in *Proceedings of the 1993 symposium on Parallel rendering*, 1993.

[11] K.-L. Ma, J. S. Painter, C. D. Hansen and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications,* vol. 14, pp. 59-68, 1994.

[12] H. Yu, C. Wang and K.-L. Ma, "Massively parallel volume rendering using 2--3 swap image compositing," in *2008 SC-International Conference for High*

Performance Computing, Networking, Storage and Analysis, 2008.

[13] S. Marchesin, C. Mongenet, J.-M. Dischler and others, "Dynamic Load Balancing for Parallel Volume Rendering.," in *EGPGV*, 2006.

[14] C. Bishop, Pattern Recognition and Machine Learning, Springer, 2007.

[15] N. R. Draper and H. Smith, Applied regression analysis, John Wiley & Sons, 2014.

[16] C. Bishop, Neural networks for pattern recognition, Clarendon Press, 1996.

[17] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling,* vol. 5, p. 1, 1988.

[18] M. Zinkevich, M. Weimer, L. Li and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010.

[19] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, pp. 2278-2324, 1998.

[20] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 2010.

[21] L. T. Yang, X. Ma and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005.

[22] D. H. Bailey and A. Snavely, "Performance modeling: understanding the past and predicting the future," in *European Conference on Parallel Processing*, 2005.

[23] S. Sodhi, J. Subhlok and Q. Xu, "Performance prediction with skeletons," *Cluster Computing,* vol. 11, pp. 151-165, 2008.

[24] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[25] J. Zhai, W. Chen and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," in *ACM*

Sigplan Notices, 2010.

[26] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE micro,* vol. 28, pp. 39-55, 2008.

[27] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009.

[28] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM Sigplan Notices*, 2010.

[29] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[30] 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 *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization*, 2014.

[31] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. Meredith and H. Childs, "Performance modeling of in situ rendering," in *The International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT*, 2016.

[32] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable parallel programming with CUDA," *Queue,* vol. 6, pp. 40-53, 2008.

[33] W. Gropp, E. Lusk and A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, vol. 1, MIT press, 1999.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[35] F. Chollet, *Keras,* GitHub, 2015.

[36] T. D. Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints,* vol. abs/1605.02688, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature