University of Duisburg-Essen
Faculty of Engineering
Department of Computer and Cognitive Sciences

Doctoral Dissertation

Visualizing and understanding large regular data

Thomas Fogal
Matriculation Number: 300306200

Department of Computer and Cognitive Sciences
Faculty of Engineering
University of Duisburg-Essen

March 27, 2016

Supervisor:
Prof. Dr. rer. nat. Jens Krüger
Reviewers:
Prof. Dr. rer. nat. Jens Krüger
Prof. Chris Johnson
Prof. Dr. Jürgen Ziegler ??
????

Eidesstattliche Versicherung / Statement in lieu of an oath:
Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
I hereby confirm that I have written this dissertation on my own and that I have not used any media or materials other than the ones referred to in this dissertation.

Santa Clara, CA, USA
March 27, 2016

Zusammenfassung / Abstract

auf Deutsch: lorem ipsum dolor sit amet ...

Visualization has emerged as a critical component in deriving understanding from the vast amounts of data generated from both simulations and modern scanning technologies such as computed tomography. The organization of these data dictates how they are algorithmically processed and thereby the performance of processes that operate on the data. For these performance reasons as well as simplicity of implementation, a regular *N*D grid organization has heretofore dominated in the simulation, medical, and visualization domains.

Yet the regular organization of data alone is not enough. The pace of data growth has exceeded that of hardware growth for many years now, and the ensuing performance gap creates difficulties for visualization algorithms. As basically all sciences move to a data-centric approach, these performance limitations become the limiting factor in forward scientific progress.

To deal with this delude of data, many have turned to *in situ* visualization: coupling simulation and visualization software together in an effort to minimize delay. This is presently a daunting process, one that cannot be sustained at large scale with the dearth of software engineering resources across the research community.

This dissertation presents a number of community-vetted ideas aimed at removing these barriers. The algorithm targetted is volume rendering, a popular method for data understanding in a number of scientific disciplines. As we dissipate these challenges, we turn to the related problem of integrating volume rendering solutions with simulation software *in situ*, specifically focusing on ways to minimize the engineering investment.

Acknowledgements

Certainly, the primary acknowledgement should go to my advisor and friend through this whole endeavor, Jens Krüger. It would be impossible to overstate his contribution to this work, from developing arguments to coding to writing and presenting; you would surely not be reading this work today, if not for Jens.

A special thanks is also due to Hank Childs. In addition to significant contributions in writing the paper associated with Chapter 4 and help integrating code with VisIt, Hank's constant encouragement and praise were often exactly what I needed to push on after setbacks.

I was fortunate to enjoy a summer at Oak Ridge National Laboratory, the highlight of which was discussing parallel rendering research with Sean Ahern, a Chromium author. I was that kid that set up Chromium in his dorm room just because 'he thought it was cool'; years later, Sean was gracious enough to pretend I was an equal. Thanks, Sean.

I am grateful to Gunther Weber and Mark Howison, who helped us identify the research challenges and questions we wanted to address when our HPG volume rendering paper was an inkling of an idea.

Chuck Hansen has been especially helpful in helping me to formulate relevant ideas and present them effectively. Thanks!

Chris Johnson deserves a special thanks. He was the perfect manager while I was at SCI: his door was always open, the research challenges were always beyond measure, the queue of collaborators grew faster than it shrank, direction was available but never prescribed, and a golden road of funding grew wherever I wandered. Perhaps the best testament to his style is that it is only now, looking back, that I realize I was working for him and not myself.

I think Matt Might for his insights into program understanding and related work in relation to the *in situ* approaches developed in this work. Chris Johnson & Chuck Hansen provided fruitful early review of these approaches.

Thanks to Fabian Proch for his help with *PsiPhi*, used in Chapters 6 and 7, in addition to some writing and editing in the former.

I thank Ethan Burns for some much-needed implementation review of the work presented in Chapter 7.

Some computations described in this work were performed using the Enzo code that is the product of a collaborative effort of scientists at many universities and national laboratories. I especially thank Matthew Turk and Sam Skillman for their help interfacing with `yt`. I thank Burlen Loring for help with ParaView scripting. Thanks are also warranted to Paul Navrátil for his assistance running on

and scheduling exclusive access to the *Longhorn* machine.

Special thanks are due to friends in my research group, notably Alexander Schiewe and Andrey Krekhov, for copious supplies of sanity-inducing conversation and, where necessary, beer. Wouldn't have been the same without you guys—thanks.

This research was made possible in part by the Intel Visual Computing Institute; the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10; Award Number R01EB007688 from the National Institute Of Biomedical Imaging And Bioengineering; the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET); by the Cluster of Excellence 'Multimodal Computing and Interaction' at Saarland University; by the Center for the Simulation of Accidental Fires and Explosions at the University of Utah, which was funded by the U.S. Department of Energy under Contract No. B524196, with supporting funds provided by the University of Utah Research fund. Resources were utilized at the Texas Advanced Computing Center (TACC) at the University of Texas at Austin and at the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. We thank John Blondin for some of the data pictured in Chapter 2, Numira Bioscience for the AltaViewer screenshots in the same chapter. We thank the visible human project for the visible human scans and Siemens Corporate Research for the 'Wholebody' data set.

The content is under sole responsibility of the author.

Contents

Chapter 1
Introduction

> The purpose of computing is insight, not numbers.
> *Richard Hamming*

The growth in data-based science has exploded in recent years. As Figure 1.1 shows, the size of data is increasing exponentially. While the increased fidelity enables new levels of understanding, this is only possible if we can interrogate and analyze the increased scale of data.

Humans are effective pattern recognizers, but error-prone with repetitive tasks and inherently visual thinkers. Among even small sets of data, it is much easier for us to apprehend a change in color than a change in repeated sequences of numbers. Mapping raw numbers to color or another visual representation is thus an effective way to take advantage of the pattern recognition capabilities of the human visual system. The growth of data only increases the need for data visualization, else we will be left with massive piles of numbers with no insight into the mechanisms or processes that sourced them.

There are a variety of data types we might apply our efforts to. In this dissertation, we concentrate on *regular gridded data*, as might be output by medical (CT, MRI) scanners or used in simulation software. These kinds of structured grids account for a disproportionately large subset of data types used in the scientific and medical domains. It is one of the 'important' data types highlighted by a report discussing the future scalability of computing in science [1]. Furthermore, operations on such data are easily mappable to promising data-parallel hardware, unlike many other data types, suggesting its continued future use at scale.

There are a number of visualization techniques that are widely appreciated for such volumetric data. Much effort has been focused on the performance aspects

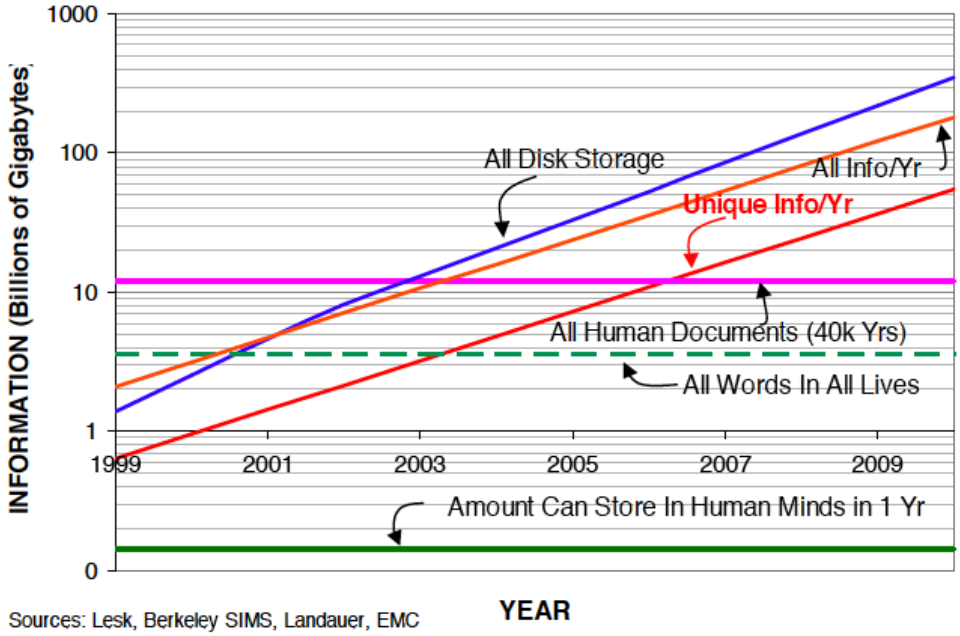

Figure 1.1: The growth of data in recent years. Hardware capabilities have not kept pace with the exponential pace of data growth.

of a few well-known techniques: mapping the algorithms to novel architectures, identifying special cases of interest, and reducing constants in the algorithms' execution. This work builds on the previous performance-oriented efforts in volume visualization.

1.1 Volume visualization

Direct volume rendering produces visualizations that accurately model tenuous mediums such as fire, smoke, and clouds. However, it is seldom applied for its ability to produce photorealistic imagery. Scientific and medical researchers use volume rendering to see *inside* a data set and highlight its internal structure. This can be extremely useful both in interrogating data as well as communicating known features.

The precision instrument used to control volume rendering is called the *transfer function*. A transfer function maps the value at every datum to colors *and opacities*. The control of opacity gives users an effective mechanism for filtering out data irrelevant to the point of interest. Color control enables the user to flexibly highlight different regions. In contrast to other volume visualization methods, this gives comparatively more power and flexibility to the user to communicate a feature exactly as intended.

Unfortunately this increased flexibility comes at the cost of considerable computational complexity as compared to other volume visualization techniques. In the general case, volume rendering is an $O(n^3)$ algorithm, requiring the consideration of every voxel in a three-dimensional dataset. Each of these samples typically requires a trilinear interpolation coupled with a set of multiply-adds needed to compute the volume rendering integral. For data of even small sizes, a naïve CPU-based computation is far from interactive.

1.2 Systems opportunities

There are a number of systems-oriented challenges in accelerating volume visualization techniques.

1.2.1 Hardware & programmability

The end of Moore's law necessitates a reorganization in software architecture to continue to see performance gains on novel hardware architectures. Serial algorithms must be amended to identify and exploit parallelism, often at multiple scales. Whereas old optimization principles emphasized eschewing computation, new strategies must reduce communication instead.

Some of the recent research in high-performance computing (HPC) has been focused on elucidating the appropriate architecture for future HPC systems. In part due to the results of this thesis, this architecture is now known to be based on the 'fat node' idea: fewer nodes with very high levels of shared memory-based concurrency on each node. Both companies leading the HPC space, NVIDIA and Intel, have invested heavily in an architecture based on accelerator cards: boards with a large number of shared memory computational units. While this is due in part due to promising performance results, such as those contained herein, much of this architecture is dictated by the practical concerns of energy and the ability to dissipate heat from processing units.

The exact characteristics of accelerators is a current topic of industry competition, but the general characteristics are large numbers of low-power cores connected to limited but high-bandwidth memory. High-powered cores need more power and there are issues dissipating the heat they generate, so it is unlikely that this basic component of the architecture will change soon. Memory faces similar problems, and thus it seems that the trend of reduced memory per core will continue.

The challenges these new architectures present gives rise to a number of programmability concerns. There are a number of proposals to ameliorate this problem, but as of yet no solution has arisen as a clear winner. Conflating the problem are a number of languages, libraries, or approaches that are intimately tied with particular hardware—including CUDA, OpenCL, Threaded Building Blocks (TBB), OpenMP, OpenAcc, Cilk, and UPC—forcing programmability issues to be inextricably linked with hardware choices.

1.2.2 I/O

The storage hierarchy is the single most limiting factor in high-performance visualization. Long term storage is slower than other subsystems by many orders of magnitude: far wider than the gap between processor and memory speeds, for example. Furthermore, there is little hope that this gap will shrink in the coming years. This poses a pernicious problem for visualization, whose primary interaction

with the storage hierarchy is via synchronous read operations. Worse, this type of workload is unique to visualization: other high-performance computing consumers are unlikely to be allies when negotiating architectural trade offs for HPC resources.

While parallel storage continues to make significant progress, the problems elucidated in recent years are a cause for concern. Parallel filesystems must intelligently partition and provide access to a distributed set of disks, but the decomposition of the parallel job is frequently opaque and only a traditional serial API is presented. This presents difficulties in avoiding issues such as false sharing, or central and thus contentious repositories for metadata. At the core of the issue is the rapidly increasing cost of data movement.

There are no imminent advances on the horizon for the storage hierarchy problems plaguing modern high-performance visualization and analysis software. While solid state drives ameliorate some of these concerns, from a scalability perspective they have only a minor impact. Given the realities of current and future architectural limitations, we must consider alternate approaches to the visualization and analysis problems. Efforts that eschew data movement in favor of redundant computation will be favored in future algorithms.

1.2.3 *In situ* visualization

In situ visualization addresses the 'too big to read' problem in large-scale visualization and analysis. The simple act of loading data significantly changes the visualization and analysis work flow. There is an inflection point in the relationship between data sizes and the performance of the storage hierarchy. Past the inflection, the impracticality of loading the full dataset precludes exploratory visualization and analysis. *In situ* visualization solves this by foregoing such use cases entirely: instead, all visualization and analysis is 'baked into' the simulation run itself, running concurrently with the parallel simulation. The advantage of this approach is that it obviates the most expensive step in the majority of visualization pipelines: reading data from disk.

To achieve this coupling of visualization and simulation requires a combination of software from multiple communities, notably simulation and visualization & analysis. This creates sizable technical and social challenges. First and foremost, these are often distinct groups, with varying research motivations, methods for assessing engineering contributions, and software processes. Data models are often radically different and data must often be shared across multiple programming languages. Finally, scalable visualization software is typically large, and both sides tend to have unique but overlapping dependency stacks.

Prior effort into *in situ* visualization is often focused on scalability and performance. At the largest scale—the most promising candidates for benefiting from *in situ* techniques—there is great concern that visualization will severely impact the execution time of highly parallel simulation software. The scalability of these results is promising, but often comes at large increases in software complexity and thereby technical debt.

This dissertation is perhaps the first to address the concerns associated with coupling increasingly complex visualization software with the simulations it uses as input. The starting thesis is that embedding visualization into simulation code should not require a visualization expert. It should be possible for the simulation author or even an independent user to perform this task. For the monumental task of integrating *in situ* visualization into the thousands of available simulation packages, coupling efforts must be measured in minutes, not weeks or months. Starting from these premises requires a radically different approach to *in situ* visualization than traditional solutions permit.

1.3 Contribution

The contributions of this dissertation fall along two broad lines. First, this work redefined the data sizes and methods in use to volume render and understand regularly-gridded three-dimensional data. Second, this dissertation is the first to radically rethink *in situ* visualization approaches for the purpose of accelerating the task of coupling simulation and visualization.

Prior to the ImageVis3D desktop volume rendering tool developed in relation to the research appearing here, 'large data' visualization was limited to 'big iron' multi-million dollar supercomputing facilities. Chapters 2 and 3 demonstrate that it is possible to use workstation computing resources to visualize the multi-terabyte datasets normally processed on supercomputers, as shown in Figure 1.2. Despite using vastly less hardware, the performance and user experience is even better than on the large-scale infrastructure. Chapter 4 demonstrates that these techniques can still be applied at scale and uses them to produce the largest-ever volume renderings, and Chapter 5 takes that concept further by investigating parallel IO concerns at scale.

Volume visualization is a means to an end: that of understanding the structures and processes present in a volume data set. However, massive data sizes preclude *post hoc* visualization for extremely large data sizes, even when taking advantage of the methods highlighted above. In recent years, many have turned to *in situ*

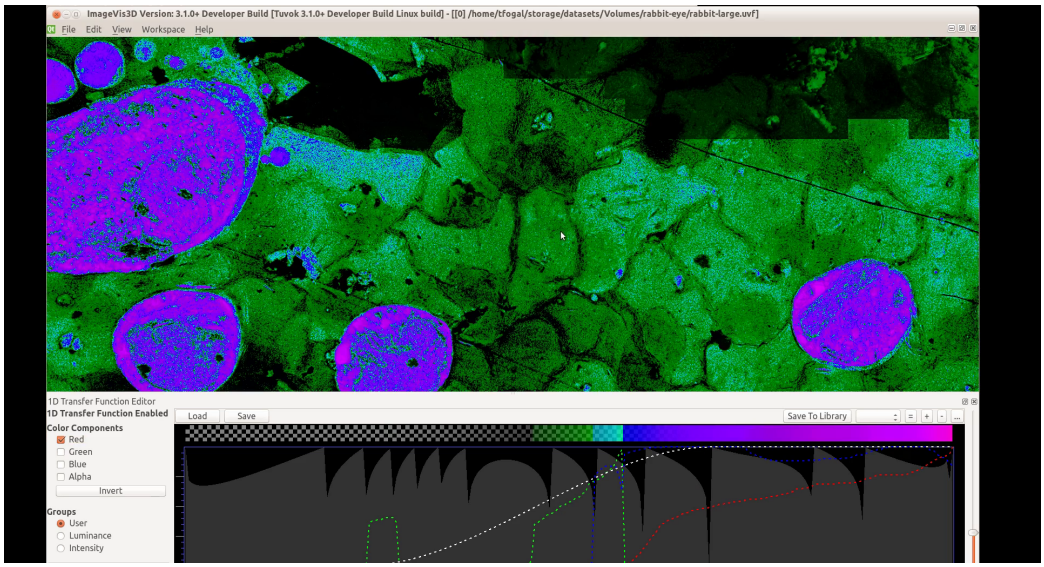

Figure 1.2: ImageVis3D interacting with a 5 terabyte data set of a rabbit eye. The research in this thesis demonstrated to the community that what was previously considered ‘large data’ could in fact be processed by workstations using commodity hardware.

visualization to tame the growing performance concerns of large-scale visualization. However, engineering groups have long coupled *ad hoc* visualization and analysis routines into their simulations to understand their simulation *while* it runs. The work developed here in Chapters 6 and 7 is the first to consider *in situ* visualization from the engineers' point of view: quick and easy visualization of their data.

Much of this work has been vetted in field-leading conferences and journals.

- Chapter 2 is based on, "Tuvok, an Architecture for Large Scale Volume Rendering" that appeared in VMV 2010 [2].
- Chapter 3 is based on, "An analysis of scalable GPU-based ray-guided volume rendering" that appeared at LDAV 2013 [3].
- Chapter 4 is based on, "Large Data Visualization on Distributed Memory Multi-GPU Clusters" that appeared at HPG 2010 [4].
- Chapter 5 is based on, "Efficient I/O for Parallel Visualization" that was presented at EGPGV 2011 [5].
- Chapter 6 is based on, "*Free*processing: Transparent *in situ* visualization via data interception" that was presented at EGPGV 2014 [6].
- Chaper 7 substantially appeared as "An approach to lowering the *in situ* visualization barrier" in ISAV 2015 [7].

There are also a number of publications by the author that are not considered 'core' components of this thesis [8, 9, 10, 11, 12, 13, 14, 15]. They are mentioned in this text only peripherally.

1.3.1 How to read this dissertation

The chapters in this dissertation can be read in any order. However the author recommends digesting the content in three groups, roughly concerned with GPU parallel volume rendering, distributed memory parallelism, and *in situ* visualization.

1. **GPU volume rendering** is covered by Chapters 2 and 3. The former chapter sets the stage for the problems that need to be solved, while the latter chapter delves deep into performance aspects. These chapters are therefore best read in the order presented.

2. **Distributed parallelism** is covered from multiple angles by Chapters 4 and 5. Understanding the work presented in the first group will be useful in following that presented in Chapter 4.

3. ***in situ* visualization** is covered by Chapters 6 and 7. Within this group Chapter 6 sets the stage for the overall goal—simpler *in situ* visualization—and thus the author recommends reading at least Chapter 6's introduction before moving on to Chapter 7.

Chapter 2

An architecture for large-scale volume rendering

2.1 Modern GPU-based volume rendering

In the past decade texture-based volume rendering on graphics hardware has positioned itself as a powerful tool for interactive visual analysis of modestly sized data sets. In earlier years slice-based approaches [17, 18] were utilized due to the limited capabilities of older graphics hardware, with the drawback of distracting visual artifacts. Later, GPU-based ray casting became possible on consumer GPUs, producing superior image quality and allowing for the integration of various acceleration strategies [19]. In addition to improvements in volume traversal methods, various approaches have been presented to efficiently render data larger than the video or even the system's main memory.

As data sizes grow, however, an efficient rendering system only solves part of

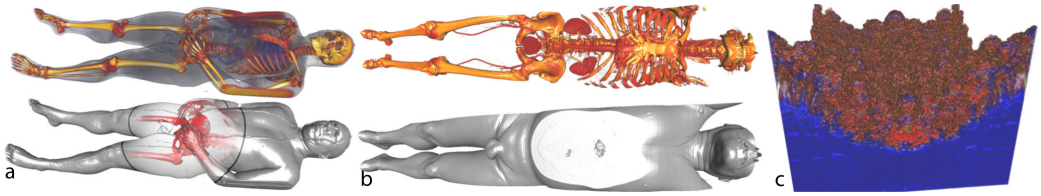

Figure 2.1: Large data sets rendered with the *Tuvok* framework. The Visible Human [16] CT scan (a), the Wholebody data set (b) and a Richtmyer-Meshkov instability (c).

the visualization problem. Along a different line of research, novel methods have been proposed to effectively interrogate, search, highlight and present data with an increasing number of high resolution features. In the course of this research multi dimensional- [20] spatialized- [21], size-based- [22], motion controlled- [23], topology-based [24], and style transfer functions [25], as well as other focus and context enhancing techniques [26, 27, 28] have been developed. For a complete and detailed survey on volume rendering we refer the reader to the state of the art report, course, and books by Engel et al. [29, 30, 31].

Due to this vast body of research a large variety of different volume rendering systems and prototypes exist both in academia as well as in industry. Yet researchers and developers often reimplement the same basic fundamentals for each new volume rendering application. It may seem that there are many different good reasons for not reusing existing, proven code, but one can usually categorize the decision into one of three cases:

- **System**: Often, the integration of new ideas and methods into large monolithic rendering systems proves to be a bigger issue than re-implementing the entire environment from scratch.
- **Software Environment**: The existing code may be implemented in the wrong environment, such as for an old operating system or graphics API. For instance, a DirectX implementation will not be suitable for a cross platform project. Further, many research prototypes are tailor-made for one system due to the lack of time and need for a more general implementation.
- **Licensing**: while largely irrelevant in the academic environment, license issues often prevent developers in commercial environments from reusing existing code. Even code that is released under Open Source conditions may come with untenable requirements for some commercial entities, such as the GPL's stipulation that related yet non-derivative code be released under the GNU license.

Research groups and companies often release their work and thus a number of systems for volume rendering structured data exist as free or open source programs. One of the earliest examples of such an open source volume rendering system is Stanford's VolPack software [32]. Unfortunately it has not been under development for two decades. A more recent example is the Simian system developed by Kniss *et al* [33]. Released under a very liberal open source license, it features both a very polished user interface as well as multi-dimensional transfer

function support. Unfortunately it falls short as far as data import is concerned and development ceased years ago; therefore no novel render modes are implemented. Other such discontinued frameworks and toolkits are the OGLE [34] system, optimized for large data, and OpenQVis [35], optimized for fast GPU rendering. A program tailored for 3D Microscopy, Voxx [36], has been released by Indiana University; while it has very promising features, including support for 4D data, it is only published in binary form. While Bruckner and Gröller's 'volumeshop' [37] implements unique GPU accelerated illustrative render options, its development ceased in 2005 and no current version is available. Further, it only supported their proprietary volume format and the current license disallows the use of the code in commercial environments.

For medical applications the MITK toolkit [38] delivers many interesting features, including support for large data sets and data manipulation routines, but it offers only basic transfer function support and slow performance compared to highly optimized out-of-core GPU volume rendering systems. Solely on the Apple Mac OS X platform, OsiriX [39] offers unmatched DICOM support in an open source application, but as the tool is tied closely to Apple's Cocoa framework and implemented in Apple-extended Objective-C, it is nigh-impossible to port to any other platform.

Instead of using a specialized volume rendering application, existing visualization toolkits can be utilized to render volumetric data. The most prominent examples are the VTK [40] and ITK [41] systems, which allow for extremely versatile and flexible rendering and modification of many types of data sets. The major drawback is the lack of support for out-of-core processing, forcing application developers to concoct external strategies to handle large data sets. Built on top of VTK, ParaView [42] addresses the large dataset issue with a distributed memory approach but—like the underlying toolkit—does not efficiently utilize the capabilities of modern graphics cards, resulting in interactive performance only at very low quality even for modestly sized data sets. Recently, the VisCG at the Universität Münster developed the Voreen system [43], a prototyping environment for volume visualization. The interface provided exposes the underlying data flow network and many visualizations require knowledge as to how they are technically realized, which we found was not suitable for a large segment of our user base. Other non commercial visualization toolkits are the OpenDX system that is no longer under active development, and finally the SCIRun [44] and VisIt [45, 13] systems. As these systems suffered some of the problems of previously mentioned solutions (e.g. outdated render modes, slow performance, or limited support for large data sets) Tuvok is currently being integrated into these solutions. Besides

these free & open source solutions, a number of commercial products exist such as AVS2, Amira, Ensight, syngo, VGStudio Max, or AltaViewer. As these systems are closed source, obtaining detailed information on their operation is difficult; the possibility of integrating Tuvok into these systems is intriguing, but we do not discuss them in detail for this work.

In order to address the aforementioned three issues and to overcome the limitations of existing systems, we present *Tuvok*, a system built of cleanly separated components that can be used together, such as in the *ImageVis3D* application, or stand-alone. The entire system is implemented in C++ with OpenGL graphics and is designed to be completely platform independent. When necessary, Tuvok's components can be compiled into a shared library and accessed from another programming language. Tuvok is also released with a modest open source license that allows unrestricted academic and commercial use of the code. Specifically, *Tuvok* offers the following benefits:

1. **Large Data Support** Given sufficient storage space, the system can theoretically handle data sets of up to 16 Exabytes in size.

2. **Modular Design** While the application ImageVis3D presents itself to the end-user as a single application, it is composed of a collection of independent Tuvok frameworks.

3. **Self contained** While ImageVis3D requires Nokia's Qt library as an external dependency, *Tuvok* itself does not rely on external libraries at all.

4. **Cross platform support** *Tuvok* as well as *ImageVis3D* support all major platforms, including various versions of Microsoft Windows, Apple Mac OS X, and many Linux variants.

5. **Legacy hardware support** Tuvok has been extensively tested to work even with the very limited GPU capabilities of older or less capable systems.

6. **Up To Date Rendering algorithms** Besides its support for 2D and 3D texture based slice based volume rendering—mostly for older graphics hardware—*Tuvok* features GPU based ray casting to interactively render images of the highest quality.

7. **Provenance Support** *Tuvok* and *ImageVis3D* provide provenance hooks, with provenance recording and playback realized via VisTrails [46].

8. **Open Source** Tuvok and ImageVis3D are released under the very liberal MIT license, meaning that practically no usage restrictions exist—including the use of ImageVis3D or its components in commercial applications.

The remainder of this chapter is organized as follows. In Section 2.2 we discuss the design of *Tuvok*, focusing on the ways that the library handles large data. To demonstrate the versatility of *Tuvok* and *ImageVis3D*, we describe extensions to the system in Section 2.3, and projects that have incorporated *Tuvok* in Section 2.4. We conclude with a summary of the presented system and future research directions.

2.2 Design

The ImageVis3D system is composed of three major components, the Tuvok Volume rendering library, the Tuvok IO library, and the Qt based UI toolkit. Note that these components are designed to work well together but can also be used separately or replaced by other external libraries (see Section 2.4 for examples). In fact, during the compilation process of Tuvok the subcomponents are compiled as separate libraries that are simply linked together. During the design of these components care has been taken to create flexible and simple interfaces between the subcomponents. As an example of this decoupled design, the communication from the UI to the rendering and IO systems happens through a single entity, named the `MasterController`. This concept makes it easy to intercept all the communication to and from the UI (see Section 2.3) and is also the heart of the scripting interface built into ImageVis3D that allows programmatic control over the application.

2.2.1 The volume rendering library

The Tuvok volume rendering library contains the core graphics algorithms to render volumetric data. Currently, a slice based volume renderer as well as GPU based ray casting renderer are available via OpenGL. For pure software based rendering the system currently relies on the Mesa library.

2.2.2 Interactivity and quality

One of the primary design goals of Tuvok is that it should be able to visualize data sets of incredible size on almost any commodity system. We have previously scaled

the renderer to data sizes greater than 2 terabytes [4], including the 5 terabyte rabbit eye from Figure 1.2.

This is achieved using a streaming, progressive rendering system guaranteeing interactive frame rates with adaptive quality. The generation of full quality imagery is also guaranteed on all configurations, with any data set, but may not happen interactively.

To achieve this goal Tuvok utilizes a multiresolution level of detail (LoD) data representation. It queries the volume parameters from the Tuvok IO Library—or an external IO framework through a documented API if the IO library is not used—and uses that information together with the current viewing parameters and system performance history to compute a work order for the current render task. More details are available in Section 2.2.3.

To achieve goals 4-6 in the list above, renderers contain a variety of extra code paths for compatibility settings, as a means to address a number of issues discovered in OpenGL drivers. Tuvok contains multiple renderers, based on ray casting, 3D slicing, and 2D slicing, that span a range of quality versus portability across GPUs and drivers. This has been important to support a breadth of collaborations, as less technical users tend to have integrated graphics chips that lack support for even 3D textures. One feature driven by this requirement is the ability to select the bit width of the framebuffer object (FBO) used for rendering, because we found that some drivers would switch to a software path when rendering into a 32-bit FBO.

Table 1 gives timings for multiple data sets on different systems, demonstrating the system's compatibility and scalability. For these timings the progressive rendering has been disabled: only the time to render the maximum quality image for the given view was measured. With the progressive rendering turned on all data sets render at the chosen refresh rates on all systems. Note that the systems used in the test cover chipset integrated GPUs as well as also high end PC configurations. Timings are presented for small data sets as well as reasonably sized CT scans and simulations. Using even larger data sets does not significantly impact the performance of the system, as the amount of data accessed is bounded by the screen resolution.

2.2.3 Large scale data handling

While Tuvok can take advantage of recent advances in hardware capabilities, it is still true that data are growing and have been growing faster than hardware capabilities allow. Thus, while the size of data sets that we can interactively render is increasing with each hardware revision, we still find that a larger percentage of

data set	Air	Pro	Vista
C60 Molecule 128x128x128 8bit = 2 MB See Figure 2.2	110 / 184	80 / 124	12 / 14
VH Male CT 512x512x1884 8bit = 471 MB See Figure 2.1a	380 / 500	526 / 744	48 / 76
Wholebody 512x512x3172 16bit = 1586 MB See Figure 2.1b	680 / 700	587 / 984	126 / 301
RM Instability 2048x2048x1920 8bit = 7680 MB See Figure 2.1c	5523 / 6112	3112 / 3520	196 / 321

Table 2.1: Tuvok timings in **milliseconds** for various data sets and configurations. "Air": MacBook Air, 2GB RAM, onboard GeForce 9400, "Pro": MacBook Pro, 4GB RAM, GeForce 9600, "Vista": PC running windows Vista, 24 GB RAM, Quadro 5800. All tests were performed in isosurface mode (first value) and in 1D transfer function mode (second value), using the ray casting renderer and sampling twice per voxel into a 1024x1024 viewport. The camera was zoomed such that the data set covered the entire viewport, and the datasets were divided into bricks of size 256^3.

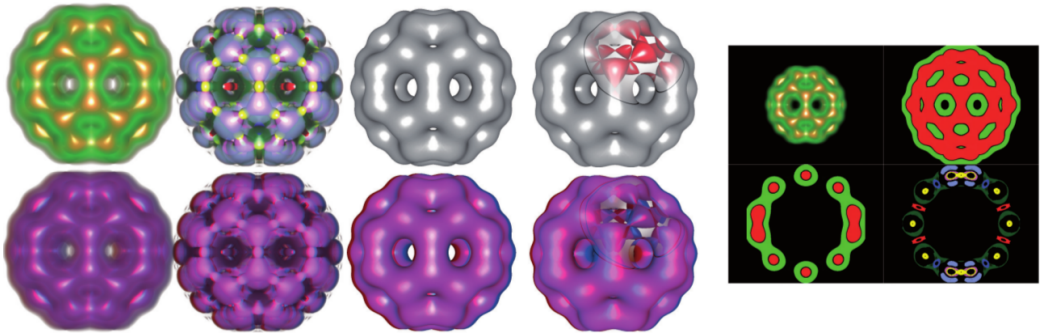

Figure 2.2: Various render modes applied to the C60 dataset. In the top row 1D and 2D transfer functions, isosurface extraction, and ClearView are shown. The bottom row shows the same views in anaglyph stereo mode. On the right is two by two mode featuring a 3D view, a MIP view (top right) and two slice views (bottom).

our data sets cannot be rendered interactively. It would be unreasonable to assume this trend would reverse in the coming years. Therefore, it is critical that interactive visualization systems incorporate progressive renderers.

Tuvok's progressive renderer is based on overloaded concepts of frames and subframes. In the context of Tuvok, a frame is a single, complete rendering of the data at native screen and data resolution. A subframe is an intermediate state between no rendering and a frame that includes the full spatial range of the data and any annotations present in the visualization. The quality of successive subframes monotonically increases. A sequence of these subframes are rendered before the final frame is displayed, detailing different approximations of the complete rendering much more quickly than a frame can be displayed. We guarantee that there is always at least one subframe that can be displayed interactively (within a hundred milliseconds). The system turns to such a subframe when the user is actively interacting with the data.

To model the concepts of frames and subframes, Tuvok uses a multiresolution, level of detail representation of data. For the most part, a subframe corresponds to the data at a particular level of detail. At the coarsest level of detail, the data are small enough that they can easily be read from disk under our real time requirements. However, we found that older GPUs could not always render such data quickly enough for our needs. Therefore Tuvok always makes available up to three additional subframes. These are generated by lowering the screen resolution of the rendering (and upscaling before display to the user), lowering the sampling rate used by the renderer, or both. Lowering resolution and sample rate significantly reduces the strain on the fragment processing stage of the graphics pipeline, allowing Tuvok to respond quickly even on low end hardware. We do not know any OpenGL 2.0-capable GPU that Tuvok does not perform acceptably on, and (through extensions) Tuvok can render even on some cards that do not report OpenGL 2.0 capabilities.

Preprocessing

Most data are not fed to visualization software with multiple levels of detail included. To accommodate such data, Tuvok's IO subsystem implements a preprocess that generates a multiresolution hierarchy. The data at their native resolution form the finest level of detail, and we subsample by two recursively until a level of detail exists that is less than or equal to a predefined user-configured limit. We also use this opportunity to perform other operations on the data, such as ensuring a consistent endianness. In most cases preprocessed data can be loaded from disk

directly into GPU memory.

The primary issues we face when loading large data are 32-bit address spaces, limitations on GPU 3D texture sizes, and managing the IO in an efficient manner. The address space limits us to only handling two gigabytes of data at any one time. Limitations on texture sizes prevent us from 'simply' loading the data into a single, large 3D texture. Typical IO performance on desktop-class and predicted future hardware informs our strategy for how we access and consume data.

To tackle these issues, the preprocess divides each level of detail into a set of bricks, with each brick small enough to fit into the texture memory of any modern GPU. The rendering core will render each level of detail in an out-of-core fashion: a brick will be loaded, rendered, and discarded as a single atomic operation. This allows the renderer to load data of virtually unlimited size with very little available memory, as the required amount of memory is independent of the data set size. To achieve the IO performance we require, the IO library uses large reads (by default, 16 megabytes) that make seek times virtually irrelevant.

A simple survey of modern disk drives finds reported seek times ranging from 3.75 up to 8.9 milliseconds. Sustained transfer rate capabilities can be as low as 65 MB/s; see Table 2.2. While there are of course differences across drives and manufacturers, multi-megabyte reads very quickly overtake seek times as the predominant factor in disk transfers. At 65 MB/s, it takes almost a quarter of a second to read 16 megabytes of data, yet only 8 milliseconds to seek to the position of that block. Even as one gets into the higher end drives, the story is the same; a Cheetah 15K.5 would take 0.12 seconds to read a 16 megabyte chunk of data, and only 3.75 milliseconds to seek to the appropriate location on disk. In relative terms, seek time makes up approximately 3% of the time required to read the data block. Based on these simple calculations, it is clear that transfer rates will have to improve drastically before seek times become a relevant parameter.

We have also benchmarked our I/O subsystem using solid state drives. Table 2.3 shows the time spent on I/O when loading a 648-brick data set via Tuvok. The SSD boasts vastly better seek times, on the order of microseconds instead of the normal milliseconds for mechanical drives, and a factor of two to three improvement in bandwidth. Using large reads, the seek time matters little in this case, but as shown in Table 2.3 Tuvok benefits from the improved transfer times offered by SSDs.

Paging strategy

Transfer time forms the majority of our pipeline execution time when using high end GPUs. Therefore, by maintaining a cache for individual bricks, we can improve

Drive name	Seek time (ms)	Sustained transfer rate (MB/s)
Cheetah 15K.5 SAS	3.75	73 to 125
WD Caviar RE2-GP	8.9	84
Barracuda 7200.8	8	65
WD 740GD	5.2	72

Table 2.2: Relevant disk performance characteristics for disks ranging from high-end server drives (Cheetah 15K.5) to an aging model released 11 years ago (WD 740 GD)

3-disk SATA RAID5	Solid state drive
64.8704	27.6723

Table 2.3: I/O component (seconds) for rendering a 9 gigabyte timestep from a simulation of a Richtmyer-Meshkov instability.

the overall rendering time by obviating the transfer time for oft-requested bricks.

A straightforward paging strategy for such a cache would be Least Recently Used (LRU), however this strategy delivers poor performance in many situations. Consider a dataset with 10 bricks, and a brick cache capable of storing 9 bricks. In the first frame, all ten bricks must be paged. Further, loading the final brick of the first frame will evict the first brick of that frame. Assuming any reasonable amount of frame-to-frame coherence, the next frame will again need the same 10 bricks, and they are likely to require a similar depth ordering. Thus, in the second frame, the first brick we will need is the brick we just evicted at the end of the last frame; further, the second brick we need will be evicted while loading the first brick of the second frame, and so on throughout the entire frame.

We have implemented a custom paging strategy that takes into account our progressive rendering system. In this strategy, we evict bricks *within* a frame using the Most Recently Used (MRU) strategy; we evict bricks *between* frames using the LRU strategy. The rationale for the former is that once we have used a brick in a subframe, it will not be used in the rendering of that frame again until the progressive renderer starts over, and we may service a large number of bricks in the interim. However, if we do start the frame from its earliest subframe again, particularly before finishing the frame, we are likely to need the oldest bricks that are present in the cache. Between frames, we rely on frame-to-frame coherence. If

a brick was not used in the previous frame, and is not used in the current frame, it is likely to not be required in subsequent frames as well; a common example is if the user has enabled a clip plane: any viewing transform will not affect the bricks that are clipped away by the plane. Therefore the LRU strategy will tend to evict bricks that are not visible under the current transfer function, isosurface, or viewing parameters.

2.2.4 UI and networking library

To facilitate rapid development of other visualization applications, all those components built on top of Qt that are not specific to the application level were separated, allowing them to be shared and reused in future applications. These components can be roughly categorized as the UI and networking components. The independent networking components include the bug reporting, update checking, and data set sharing subsystems, while the UI components include the base classes that define the look and feel of ImageVis3D, such as dialogs, tool widgets, user interaction, and persistence.

2.3 Extensions to *Tuvok* and *ImageVis3D*

In this section we present a couple of examples to demonstrate how simple it is to add new features or extend existing functionality. We present examples from research projects implementing a prototypic environment to experiment with new methods (Section 2.3.1) as well as new features to ImageVis3D to use it for other research.

Due to the modular design, the scripting interface, and the `MasterController` concept, integration with external software is simple. As the UI and execution layer communicate strictly through a single class, the `MasterController`, any type of external communication channel can simply attach itself to this class and track changes. Control of the library can also happen through the `MasterController` via script commands that allow programmatic modification of all of Tuvok's features.

2.3.1 Extensions to the rendering subsystem

ImageVis3D has been extended to provide domain specific visualization capabilities. In some domains, it is necessary to visualize multiple data sets simultaneously.

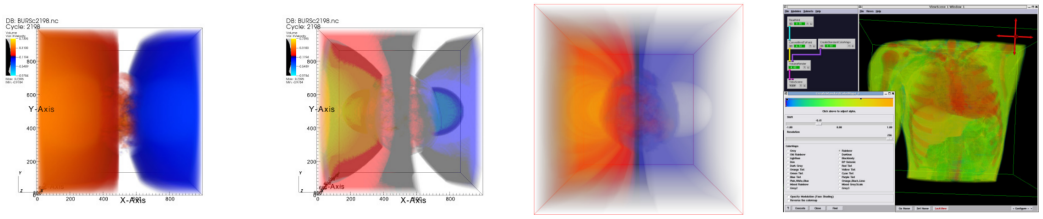

Figure 2.3: 3D texture, SLIVR, and *Tuvok* volume renderers in VisIt (left); *Tuvok* rendering a torso in SCIRun (right).

A student has modified ImageVis3D to render multiple data sets that live in overlapping space, and added domain-specific widgets for ease of use in a particular scientific domain. One such example is a dialog to automatically create transfer functions, based on external knowledge of characteristic data distributions within data sets common to that field. A second example is repurposing the 2D transfer function editor to utilize different metadata along each axis.

2.3.2 Extensions to *Tuvok*'s controller

For provenance tracking, we have integrated VisTrails, a production provenance framework with well-developed APIs for integration with external systems. The integration of VisTrails provenance tracking features required a two way communication from and to Tuvok. Interactions made by the user need to be communicated to VisTrails to track the provenance, but also VisTrails needs to be able to control Tuvok to perform undo/redo operations. Thus, this example is prototypic for any type of recording or remote control of Tuvok, such as cluster extensions or connections to novel input devices.

2.4 Use cases of *Tuvok*

In the following we present examples where Tuvok—or only some of it components—have been integrated into rendering environments other than *ImageVis3D*. Figures 2.3 and 2.4 demonstrate the integrations presented here.

2.4.1 SCIRun

SCIRun is a problem solving environment for modeling, simulation, and visualization of scientific data. It is an example of what we refer to as a legacy application, in that it was developed without the ideas implemented by Tuvok in mind. In particular, this means that the system must work with in-core, 'unbricked' data sets of a single resolution.

To support such an environment, Tuvok has a simplified API for existing systems that do not include level of detail or bricking concepts. The information flows one way from the controlling application to Tuvok, and includes a reference counted smart pointer to the data, as well as metadata and rendering parameters. For small changes in rendering parameters, data shared from previous frames is retained and simply re-rendered. When changing or passing a new data set to Tuvok, the old data set is removed and replaced by a new reference counted smart pointer. This scheme allows us to avoid data copying between the host application and rendering library. In these kinds of systems, Tuvok does not have access to a multiresolution form of the data, and thus cannot guarantee interactive performance.

2.4.2 VisIt

VisIt is a data visualization and analysis application that is well-suited to large scale data processing on leadership computing platforms. We have integrated the underlying rendering core as an option alongside VisIt's existing volume renderers. Since VisIt already supported domain-based data set decomposition, it can easily take advantage of an additional Tuvok feature: bricking. This allows VisIt to volume render data of arbitrary size on the GPU, whereas it was previously limited to resampling the data or utilizing software rendering.

Though data do not come directly from a data file in this and other integration work, the abstraction provided by Tuvok's IO layer allows the rendering core to remain ignorant about the source of the data. The metadata that must be supplied to Tuvok scales with the complexity of the application: in the unbricked, SCIRun case, Tuvok can be told only the brick size (assuming the brick lies centered on the origin); with decomposed data, Tuvok must be informed of the world space location of the bricks; for progressive rendering applications, such as ImageVis3D, the LoD that a brick belongs to must also be given. Should an application choose, it can also supply additional metadata to allow advanced rendering optimizations.

An issue that arose specifically in the VisIt integration was state management

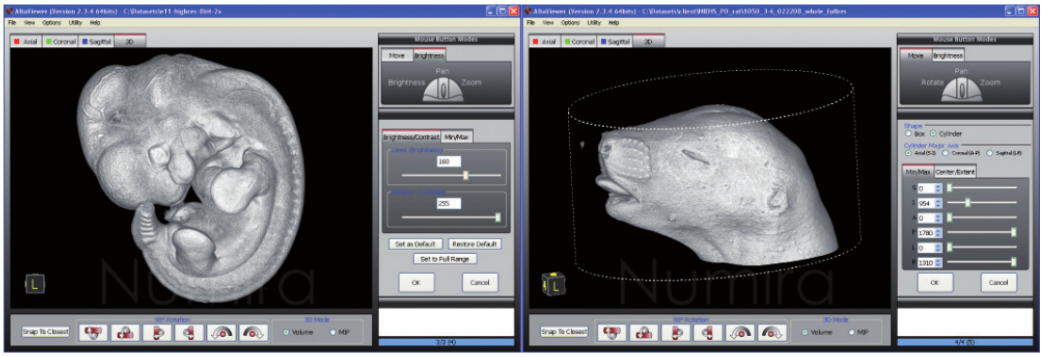

Figure 2.4: The left image shows an E11 mouse embryo (2.4GB) while the right image depicts a P0 newborn rat (7.6GB). Both specimens were stained using Numira's custom protocol and scanned using microCT. Images courtesy of Numira Biosciences. Copyright 2009 © Numira Biosciences. All rights reserved. AltaViewer Software available at `http://www.numirabio.com/`

in large, established software systems. The OpenGL API is a global state machine, and VisIt has many sub-libraries that can and will change the global state in ways we cannot predict. Tuvok therefore makes very few assumptions about OpenGL state. During the 'setup' stage, Tuvok takes state information—camera and viewing reference points, data, etc.—and stores it locally. A single method then uses all that information to configure OpenGL state once before moving on to per-brick rendering. For efficiency reasons, the system leaves the OpenGL state 'as-is' when finished rendering, much like other VisIt subsystems and libraries do. Until OpenGL establishes an object model, we have found this to be the best method for managing OpenGL state.

2.4.3 AltaViewer

Finally, we demonstrate the usability of Tuvok's components in a commercial environment. Numira Biosciences is a specialty contract research organization that focuses on high-resolution imaging and analysis of small animal specimens, provides researchers with quantifiable, visible evidence of disease progression, as well as drug efficacy and drug side effects in their animal models. For the next generation of their visualization suite 'AltaViewer' (see Figure 2.4) they have chosen to replace their proprietary IO library in part by Tuvok's IO components to achieve significantly better performance.

2.5 Conclusion and future work

In this paper we have presented the Tuvok framework as well as ImageVis3D, an application built with Tuvok. We gave insight into large data support in a production volume renderer. We also gave a couple of examples of research projects and commercial use of components of Tuvok. We are currently working on three major extensions to Tuvok. First, the support of time dependent data sets, in particular we are working to extend the progressive rendering concept to this data as well. Secondly, we are extending Tuvok to render multiple data sets in overlapping 3D space; due to the out-of-core nature of the system an efficient implementation of this feature is non-trivial. Finally, we also plan to add purely software based as well as OpenCL based ray casters to allow for fast rendering of ultra large data sets on headless clusters with and without GPUs.

Chapter 3

Ray-guided volume rendering

Volume rendering continues to be a critical method for analyzing large-scale scalar fields, in disciplines as diverse as biomedical engineering and computational fluid dynamics. Commodity desktop hardware has struggled to keep pace with data size increases, challenging modern visualization software to deliver responsive interactions for $O(N^3)$ algorithms such as volume rendering. We target the data type common in these domains: regularly-structured data.

In this work, we demonstrate that the major limitation of most volume rendering approaches is their inability to switch the data sampling rate (and thus data size) quickly. Using a volume renderer inspired by recent work, we demonstrate that the actual amount of visualizable data for a scene is typically bound *considerably* lower than the memory available on a commodity GPU. Our instrumented renderer is used to investigate design decisions typically swept under the rug in volume rendering literature. The renderer is freely available, with binaries for all major

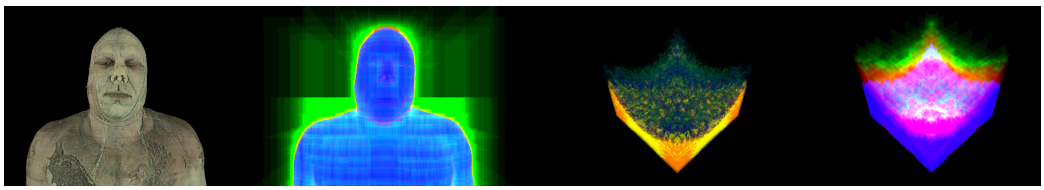

Figure 3.1: The Visible Human male full color (~12 GB) and a Richtmyer-Meshkov instability (~8 GB) render in 34 ms and 58 ms, respectively, using our ray-guided volume rendering implementation. On right are views that highlight the areas that take advantage of empty space leaping (green) and early ray termination (blue).

platforms as well as full source code, to encourage reproduction and comparison with future research.

3.1 Introduction

Modern volume rendering is heavily focused on the concepts of empty space skipping and the fast detection of ray saturation. Both of these concepts have extensive effects on the amount of compute work required. However, even more relevant is their ability to reduce the working set of extremely large datasets down to a small kernel, which can significantly reduce the amount of data that must be loaded from a slow network or local disk resources. This has enabled interactive volume rendering for very large data on commodity hardware [47, 48, 49].

There are a variety of trade-offs in the development of a modern volume renderer. The choice of brick size, for example, can significantly impact the effectiveness of empty space skipping. We note that the presentation of most volume rendering systems lacks detailed insight into these parameters. Further, these factors can interact in complex ways. As an example, empty space skipping works considerably better with smaller bricks sizes, but disk throughput drops sharply with small requests. Compression can further complicate the issue.

We seek to rectify this situation by performing a thorough study of the interaction of these parameters within the context of GPU-based ray driven volume rendering. We have surveyed recent volume rendering literature and implemented a renderer by piecing together the best ideas from a multitude of systems. These ideas were extended with notions required for our environment—for example, by removing the requirement that datasets fit in GPU memory. Along the way, we instrumented every corner of the renderer and utilized this instrumentation to exhaustively explore relevant options. The final result achieves better performance than previous work and provides a guided tour through the maze of design choices available in a modern volume renderer.

3.2 Related Work

Volume visualization on consumer graphics hardware has become widely utilized as a means to cope with the growing sizes of data. GPUs have proven useful in both ray-tracing and rasterization techniques [50, 51], rendering of diverse scenes [52], as well as considerably more general tasks [53].

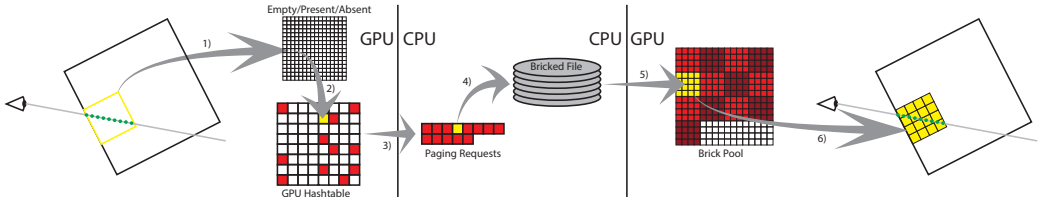

Figure 3.2: The missing brick reporting / paging subsystem of our volume rendering approach. Missing bricks are recorded into a hash table (1, 2), to be paged in (3, 4, 5) and rendered in subsequent frames (6).

Volume rendering accelerated by GPU hardware was established in the mid-90's [17, 18], initially based on hardware compositing of volume slices. The ability to do raycasting came later [19]. Since the time of the initial GPU-based volume renderers, researchers have been concerned with methods to work around the limited memory available on GPUs. The prominent technique for volume rendering large data on a GPU is to use a multiresolution representation [54, 55, 56]. This method hinges on the concepts of empty space leaping and early ray termination [57], two techniques developed early on that demonstrate that sampling can be significantly reduced in many instances of volume rendering.

There has been much work on accelerating ray-traced volume rendering in recent years. Voreen implements a more general architecture, including GPU-based raycasting [43]. Tuvok implements a flexible volume rendering system with support for very large datasets [2, 8]. Knoll et al. utilize a bounding volume hierarchy and optimized SSE to achieve very fast volume renderings [47]. Gobbetti et al. and Boada et al. detail methods for traversing tree structures on the GPU for the purpose of volume rendering [58, 59]. The Gigavoxels [49] system traverses N^3-trees on the GPU to choose an effective resolution. With the large gap between processing power and data sizes, some communities have turned to distributed memory systems for large-scale volume rendering [60, 61, 4, 62].

Our algorithm employs a lock-free data structure on the GPU for feedback information. Highly-concurrent Lock-free structures are ideal for the manycore GPU environment, however they have previously been challenged by the lack of concurrency primitives available for the OpenGL platform. We make use of a lock-free hash table very similar to that of Michael's [63], implemented in a manner similar to Lux and Fröhlich's implementation for terrain rendering [64].

Hadwiger et al. presented a volume renderer similar to ours [48]. Their system is aimed at volume rendering highly anisotropic data as it is streamed real-time

from a high-resolution microscope. Our renderer improves upon theirs in a number of ways:

- We perform brick lookup each brick, instead of every sample, maintaining the simple and familiar ray-marching core that is well-documented in volume rendering literature.
- We expound on how to use modern GPU features to implement our lock-free feedback data structure. This enables the implementation to spend more time computing on the GPU and less time pushing data around.
- We utilize an out-of-core, progressive rendering methodology, breaking the GPU-memory-size barrier that limits data sizes from Hadwiger et al.'s work. This also allows us to gracefully scale down to consumer-level graphics cards.

While we believe these to be novel additions, we do not consider them to be this work's major contribution. Rather, we provide new depth to the discussions of a variety of parameters that are relevant in the development of a ray-guided direct volume renderer:

- The strategy to be used to load higher resolution data when a variety of intermediate choices are possible;
- an understanding of the miasma of issues surrounding bricking and brick sizes;
- empirical evidence demonstrating that the working set for direct volume rendering is indeed bound more by the screen resolution than the dataset;
- a novel method for ray-guidance storage and propagation to the input system's logic;
- how to effectively handle real-time updates to the transfer function; and
- the effect of brick layout strategies on large volume access times.

In contrast to previous renderers, ray-guided volume renderers couple the rendering process with the identification of which subvolumes ('bricks') must be loaded. We describe the operation of ray-guided volume renderers, in Section 3.3. In Section 7.4.5 we detail a plethora of benchmarks that demonstrate the performance of the renderer.

In many prior volume renderer evaluations, results are generally limited to the raw performance of the renderer. However, we note that—for some reason—users of our volume renderer rarely ask how many milliseconds it takes to render the visual human. One thing users *do* ask is how large the data can get before the renderer becomes unusable. For this reason, we have engineered our renderer so that it does not require that the volume fit in core. Furthermore, users generally value a responsive system over a performant system. They are curious if money should be spent upgrading a video card or buying a solid state drive. Design elements are carefully expounded and conclusions are drawn in Section 3.5.

Finally, Section 3.6 gives our final remarks, and note both limitations and opportunities for future work.

3.3 Ray-Guided Grid Leaping

At the macro level, our algorithm is reminiscent of the recent work of Hadwiger et al. [48], as well as Engel's CERA-TVR [65] that is in turn based on the Gigavoxels system [49].

With Hadwiger et al. we share the requirement of a set of simple multiresolution Cartesian grids, along with an OpenGL-based table to report missing bricks. A multiresolution hierarchy is built as a preprocess for input data that exist at only one resolution (details are in Section 3.5). From the CERA-TVR system we inherit the idea to only recompute and request grid cells at boundaries.

3.3.1 Overview

We endeavor to create a volume renderer that can render massive datasets extremely fast on commodity GPU hardware. The major issues in such a renderer are:

1. Identifying regions that must be sampled densely.
2. Precisely locating the transition between these regions and regions that exhibit considerable homogeneity.
3. Terminating a ray as soon as possible.
4. Efficiently communicating regions to be rendered in the future to the IO layer.

Points (1) and (2) ensure we concentrate the computational effort on the areas that require it. Point (3) is critical because it means we do not have to load the data

beyond the point of early termination, significantly reducing costly disk traffic. If point (4) is not sufficiently addressed, the renderer will load large amounts of data that are not needed for rendering, at severe costs in performance.

To the first point, we employ an efficient metadata structure that allows us to quickly identify these regions. Points (2) and (3) are handled through an educated choice of brick size that is discussed more thoroughly in Section 3.5. A major component to modern volume renderers is how they address point (4), now by and large based on *ray guidance*. That is, the sampling characteristics of the ray determine which data to load. Stated differently, the future data requirements are computed *in concert* with standard ray traversal and accumulation.

The entire operation is detailed in Figure 3.2. For each ray we compute the level of detail required to maintain a pixel error of less than one. With this level and the position in the volume we compute a brick index. This brick index is used to fetch information from a lookup table (Figure 3.2.1) to identify whether the brick is a) empty, b) non-empty and present on the GPU, or c) non-empty and absent. When it is empty, we skip the brick and repeat the process at the brick's exit point. When it is non-empty and present, we ray-cast that brick. When the brick is non-empty *and* not resident in GPU memory, the system returns the finest coarser level available and the missing entry is added to a GPU hash table (Figure 3.2.2). This table is read back to the host memory at the end of the frame (Figure 3.2.3), and used to page in bricks from disk or cache (Figure 3.2.4). A paged-in brick is then uploaded to a GPU texture pool (Figure 3.2.5), and a subsequent frame will use this portion of the brick pool for sampling (Figure 3.2.6).

The key component is that both ray-accumulation *as well as* identification of the bricks that are needed should occur on the GPU. The latter is natural to compute during standard ray-casting operations. Doing both operations on the GPU means brick identification comes very cheap, as it parallelizes very effectively. More importantly, performing this during ray-casting ensures that it is optimally accurate: the program never loads data that will not be used.

The basic algorithm is given in Algorithm 1. Briefly, the appropriate sampling rate is identified and we look for the data at that resolution (lines 6, 7). `GetBrick` will always return some data, but the data may be at a lower resolution than request; this is communicated through the `samplingRate` and the situation is handled on line 8. If our data are too coarse, we note that we are missing a brick (`ReportMissingBrick`) and where we are in the volume (`rayResumePos`) when this *first* occurred (`terminated`).

Every iteration through the outer loop, we perform this identification of the appropriate resolution. This satisfies our first goal as mentioned above: we identify

Algorithm 1 Ray-guided volume rendering. Each ray identifies the set of bricks that it needs for rendering independently, and reports this information for use in subsequent rendering passes.

```
1:  color = rayResumeColor
2:  terminated = true                                   ▷ assume ray will finish
3:  rayResumePos = FINISHED
4:  repeat
5:      LoD = ComputeLOD(Depth(ray))
6:      brick, samplingRate = GetBrick(ray)
7:      offsets = PoolOffsets(brick)
8:      if samplingRate ≠ RequiredSamplingForLOD(LoD) then
9:          ReportMissingBrick(brick)
10:         if terminated then                          ▷ first missing brick?
11:             terminated = false
12:             rayResumePos = ray
13:         end if
14:     end if
15:     Raycast(ray, samplingRate, offsets)
16: until ray ≥ exit ∨ Saturated(ray)
17: rayResumeColor = color
```

the appropriate sampling resolution at every brick boundary. With small bricks, this means we will do few integration steps before early ray termination is recognized. Furthermore, we detect empty bricks at this stage as well. The standard raycasting inner loop is hidden in the `Raycast` call.

3.3.2 Missing Data

As noted above, it is possible that data are undersampled while rendering. When this occurs, we display a coarser version of the data initially, but progressively refine those regions with finer resolution data until they are sampled at a rate of a single voxel per pixel, or the maximum data resolution available. This information is collected by the GPU as it renders, but must be communicated back to the CPU to coordinate disk access and update the appropriate area of the volume pool.

One solution for this would be to use multiple render targets to store information on which bricks are missing [49]. The limitation of this method is the limited mapping operation from the ray to the target buffer: there are only so many available render targets. Furthermore, this approach ignores the inherent spatial coherency between rays. Two neighboring rays are highly likely to request the same set of bricks, or at least have substantial overlap within the sets they require. With the multiple render targets approach, both pixels will encode the same value, and we will need to read back larger textures that consist of predominantly duplicate values.

Instead of utilizing extra render targets, we take advantage of an OpenGL extension that was promoted to core in version 4.2, `GL_ARB_shader_image_load_store`. This extension allows the creation of an image buffer that is independent of the current rendering buffer. Using the atomic load/store operations the extension provides, we implement a set based on a linearly-probed lock-free hash table stored in an `image_load_store` buffer. Since we are hashing based on the brick, multiple rays requesting the same brick hash to the same position. This allows us to keep the table—and therefore how much information we read back per-frame—quite small. We discuss sizing of the hash table in more detail in Section 3.5.3.

3.3.3 Brick Classification

Considering our target goals (1) through (3) given at the beginning of this section, one could classify a brick into one of three categories:

- skipped due to empty space skipping,

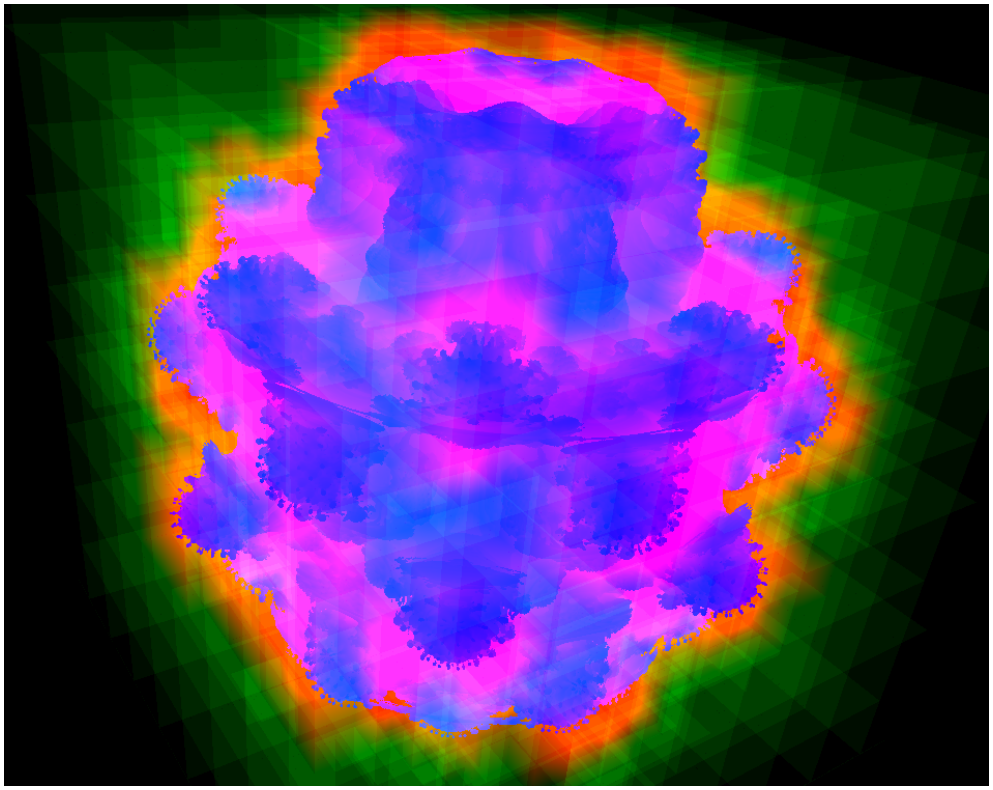

Figure 3.3: Volume rendering behavior for the Mandelbulb dataset. Green indicates bricks that were skipped via empty space skipping. Red indicates bricks that were sampled densely. Blue indicates bricks that were sampled but saturated quickly.

- early termination due to ray saturation, or
- sampled densely without saturating.

An important observation is that—in a very large number of cases—bricks fall into *either* the 'empty' or 'saturating' categories, and only *rarely* in the 'non-saturating' category. The factor that has the greatest effect on performance is how quickly a renderer can classify data into one of the first two categories, and therefore bypass a large set of the work.

To make this identification effective, ray-guided volume renderers maintain the state of each brick, shared on both the GPU and host memories. During rendering, one uses the table to identify if a brick is empty. If so, the renderer leaps over that space instead. We store this as an array consisting of one 32-bit integer per brick of the dataset.

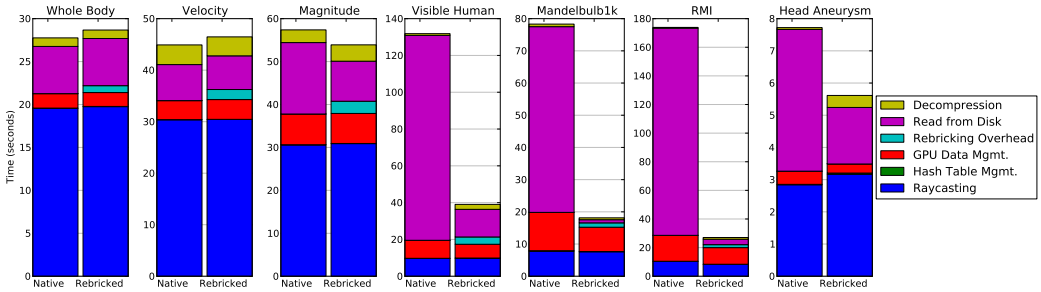

Figure 3.4: Time spent at various stages of our pipeline, aggregated over the generation of a rotation sequence. Comparisons are made between data stored with the ideal brick size for that dataset ('Native'), and data stored at a large brick size of 256^3 with the ideally-sized bricks created at run-time ('Rebricked'). 'Whole Body', 'Velocity', and 'Magnitude' suffer from a lack of ray saturation.

Figure 3.3 visualizes this classification for a large dataset under a typical view and transfer function. As shown there, the majority of the visualization falls into either the 'blue' (saturated quickly) or 'green' (skipped) sets. This also demonstrates how little data is required for a typical volume rendering. A similar rendering is given in the rightmost image of Figure 3.1, in which only the rays in the middle of the volume require extensive computation.

Of course, this classification depends largely on the transfer function and viewing parameters. In practice, however, transfer functions that produce *informative* visualizations tend to exhibit such ternary classifications.

When the transfer function is changed, this metadata information must be recomputed. For datasets with many bricks, this can induce a noticeable delay. Our current test platform can process about 7.5 million bricks per second, but even a 1 second delay between interactions is too much. Therefore, we offload this update to a background thread. Until the thread completes its work, the renderer considers all unprocessed bricks to be 'missing', causing it to request bricks that might be empty. Those bricks' metadata are directly updated and they are only loaded if they fail the empty check. The overall performance effects may be large, but the system remains responsive during this period.

3.4 Performance

In this section, we give an overview of the various stages of the renderer and how they perform. Unless otherwise noted, all timings were performed on a dual quad-core Xeon 2.2 GHz system using an NVIDIA GeForce GTX 680, with 24 GB of system memory and 4 GB of GPU memory. We mostly report results from commodity hard drives, explicitly noting some specific relevant uses of SSDs. In many cases, results were obtained from multiple screen resolutions, but we report results from an HD viewport (1920×1080) unless noted otherwise. Details of the data utilized and renderer timings are given in Appendix 3.7.

3.4.1 Benchmarks

We have chosen a variety of benchmarks to evaluate the performance of our renderer, and we elucidate the logic behind those choices here. First, the choice of HD resolution is motivated by voxel-to-pixel error ratios. All modern high-performance volume renderers try to maintain a 1-to-1 ratio between projected voxels and pixels. Adaptive resolution selection is used to ensure this ratio. Without this feature, results will be aliased, too much information will be compressed to a single pixel, and performance will suffer. Adaptive resolution means that small viewports will not stress renderers: a 512×512 viewport can get along fine with a paltry few hundred megabytes of memory, irrespective of the input dataset size.

We utilize zoom-ins, as in the accompanying video and results such as those in Figure 3.5 and some in Table 3.1, to accentuate these high resolution issues. When the volume is far away, a very coarse resolution is utilized that maintains accurate voxel-to-pixel error ratios. As the camera comes closer, higher resolutions of the source data must be utilized. We terminate zoom-ins slightly after they fill the screen; beyond this point, frustum culling's effect dominates (see Figure 3.5). The most challenging cases for a volume renderer are when data are close enough to be seen at native resolution, but far enough away that no data can be culled by the frustum.

Rotations are used to demonstrate that the renderer does not rely solely on early ray termination. As described in Section 3.3.3 and depicted in Figure 3.3, most rays either skip large parts of the volume, or terminate very quickly. With a transfer function that produces a dense volume, bricks in the front will prevent bricks in the rear from ever being paged in, effectively meaning the volume renderer need only cope with the front *half* or even less of the volume. Barring pathological volumes and transfer function choices, rotations ensure all of the data has a chance

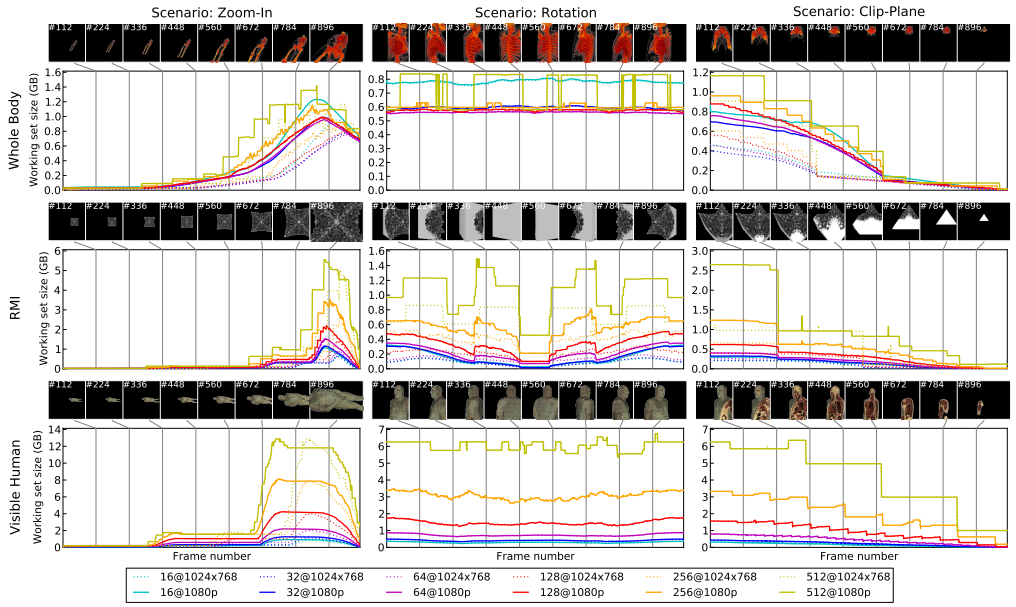

Figure 3.5: Working set sizes across three different scenarios for multiple datasets. Smaller brick sizes approximate the working set better.

to contribute to a sequence.

Transfer functions. Changing a transfer function is also an important benchmark in any volume rendering system. Doing so invalidates our brick metadata concerning which bricks are empty, causing some hash table entries in the next frame to make little sense (i.e. request bricks that are visible under the old transfer function but empty under the new one). Furthermore, the bricks in the GPU volume pool may be inappropriate for the new transfer function.

Renderer performance as measured by response time during such an interaction actually changes very little, and can even improve. However, quality suffers rather drastically. This is evident in the time to convergence after a change in the transfer function: in a typical case with the RMI data set (see Section 3.7), time to convergence increased over 6x after changing the transfer function (from $\sim$380ms to $\sim$2300ms).

3.4.2 Results

To evaluate our renderer in different scenarios, we used a standard rotation scenario with a variety of datasets, measuring the length of each pipeline stage. Figure 3.4 has these results. As IO is the prime bottleneck in many cases, we implemented a 'rebricking' scheme to mitigate the amount of IO performed. Using large reads and caching, this significantly lowers the time spent doing IO. We used 'LZ4' compression when recording this performance data, which trades CPU time for IO time.

The majority of the time is spent ray-casting, pulling data from disk, and uploading the bricks to the pool. Our novel hash table approach keeps the table small, and so reading it is very cheap: even for large data, this component does not factor in to the overall performance. The other GPU data to manage is metadata information for our volume pool (i.e. which bricks are resident), but at a single machine word per brick it costs very little to push it down to the GPU, even for very large data.

Interestingly, the time spent managing GPU data is an increasing function of volume size *until* it peaks around the size of the RMI ($2048 \times 2048 \times 1920$). This reinforces our assertion that there is only so much data visible in a given frame—dependent only on the view frustum, and *not* the dataset size—and so at some point we saturate the set of visible data. Figure 3.5 and Section 3.5.1 include more discussion about working set sizes.

3.5 Design Tradeoffs

In this section, we try to explore aspects that have not been thoroughly addressed by previous literature. Details on trade-offs and the reasoning behind our final implementation choices are given.

3.5.1 Subdivision

How a system subdivides the volume into manageable pieces can have a large effect on the performance of the renderer. The primary considerations are in regard to early ray termination and empty space skipping: small bricks are much more likely to be composed of a small range or even uniform values, which will make it in turn more likely that the brick can be skipped under a large set of transfer functions. Further, small bricks means one will detect ray saturation much more

quickly, as this is checked only when exiting a brick.

Internal Overhead The primary drawback is reduced disk throughput due to utilizing many small requests. A further drawback is the data size overhead: each brick needs two voxels of ghost data in each dimension, for sampling and gradient computation purposes. This is negligible for large bricks, but grows sharply as the brick size approaches one, as shown in Figure 3.6. Figure 3.8 demonstrates that this is not strictly a theoretical result: a small brick size greatly increases not just size overhead, but also the time to reorganize the data on disk. From these Figures we can derive that for large datasets a brick size of less than 16^3 is impractical.

External Overhead We have performed a number of experiments to identify the working set size for multiple different brick sizes. Starting with the smallest practical size of 16^3, we increase the brick size up to 512^3.

As can be seen in Figure 3.5 the working set is bound not by just the data size, but the screen resolution as well. It can also be seen that the brick size heavily influences the working set size: larger bricks allow for less efficient utilization of empty regions. From the images we can derive that a brick size of less than 128^3 is desirable to reduce the working set to roughly the memory size of a GPU.

We note that the working set size is not a strict function of the brick size, however. Figure 3.5 and Table 3.1 also show that the choice of brick size is not clear-cut. The Visible Human male performs best with 16^3 bricks, for example, whereas the ideal brick size for the 'Magnitude' data is 64^3. For the 'Whole Body' dataset, using brick sizes of 16^3 actually resulted in *larger* working sets than 32^3. This occurs when the transfer function produces large regions of semi-transparency but never reaches saturation. Indeed, when datasets contain large swaths of semi-transparent regions, the conventional wisdom is reversed: large brick sizes are generally preferred, since they significantly improve disk throughput.

If we begin to consider secondary metrics, such as the response time of the system, the choice of brick size becomes even more complex. Since bricks are the atomic building blocks in a volume renderer, one cannot load less than a single brick from disk. Therefore a larger brick size imposes a larger response time on the system. These concerns would generally push a designer to choose smaller bricks.

However, disk performance falls very sharply with small requests [5]. It is nice for a system to respond within a few tens of milliseconds, but such concerns should not dictate the design to the point that end-to-end performance suffers drastically. Furthermore, small brick sizes are accompanied with significant over-

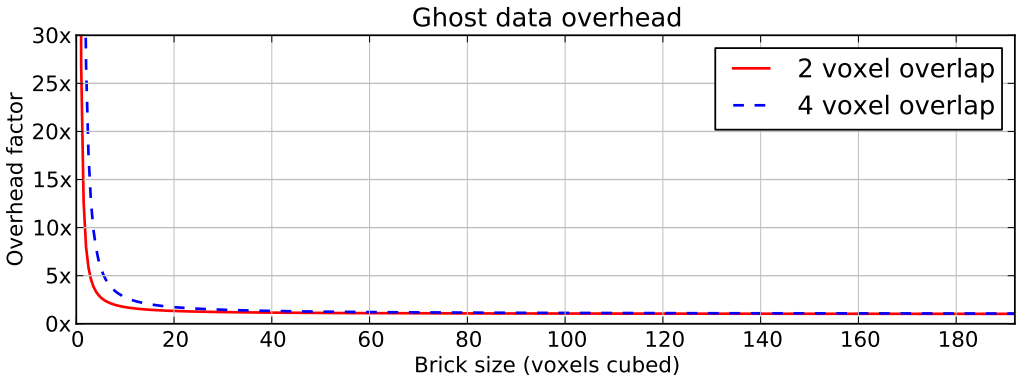

Figure 3.6: Brick size overhead. As bricks get smaller, the overhead for the additional ghost data grows significantly. At a larger brick size of 128^3, the overhead with 2 ghost voxels per dimension amounts to a few percent, whereas with 32^3 bricks this increases the dataset size by almost 50%.

head, as discussed in Figure 3.6, and do not compress as effectively as their larger counterparts.

Systems such as Reichl et al.'s hybrid surface rendering, CERA-TVR, and Gigavoxels utilize a static brick size of 32^3 [50, 65, 49]. This brick size exhibits few extremes of the performance issues mentioned above. However, it is certainly not the ideal choice for all circumstances.

3.5.2 Disk IO

Brick Layout

Figure 3.7 demonstrates how this changes with the brick size. Both disk IO times as well as decompression times are displayed there. As shown in the figure, reading data from disk becomes quite severe with small brick sizes. However, as brick sizes grow to 64^3 and beyond, decompression time becomes more important and overall time plummets. This effect is even more pronounced using a hard disk in place of the SSD used here. Intelligent layout strategies purport to minimize seek times; our results corroborate this, with the important caveat that seek times are not relevant with larger brick sizes.

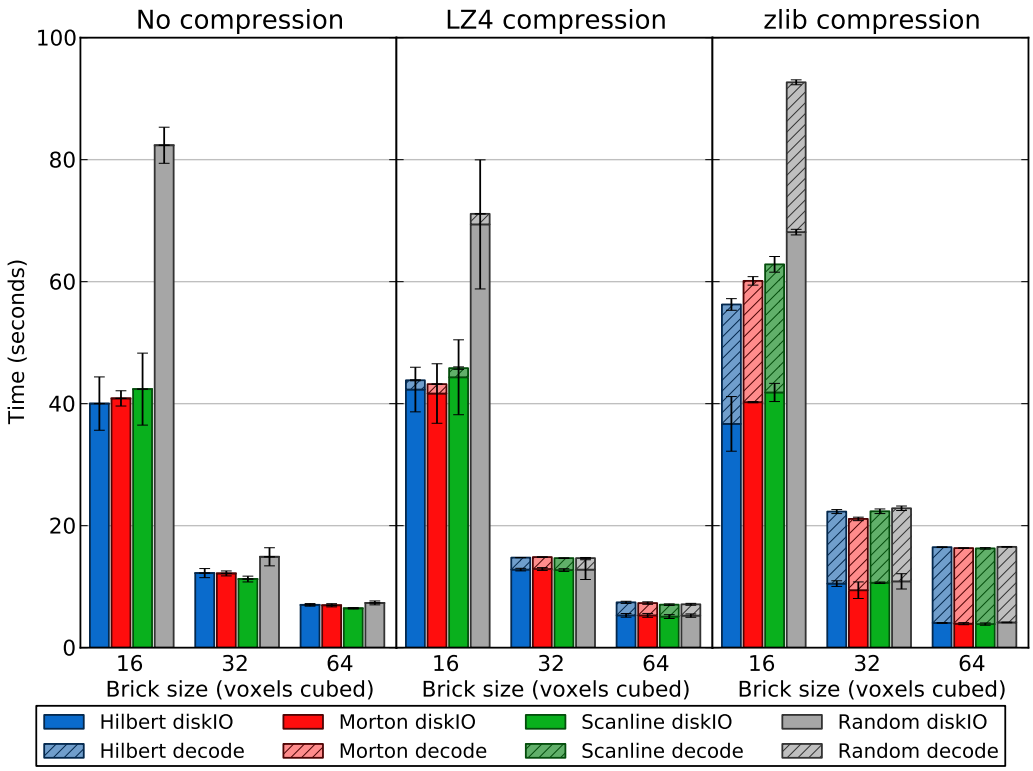

Figure 3.7: Time spent with IO-related tasks using an SSD for the RMI dataset's zoom-in scenario, sampled with 100 frames and a 1024×768 viewport. Layout strategies only see utility at small brick sizes.

Dynamic Rebricking

The renderer desires small bricks, as discussed in Section 3.5.1, as small bricks will help with early ray termination and empty space leaping. However Figures 3.7 and 3.6 demonstrate that large brick sizes are preferable for disk performance and overhead reasons. To provide the best of both worlds, we implemented a 'dynamic' bricking scheme, whereby bricks are stored on disk in a rather large size (e.g. 256^3) but presented to the renderer as if they exist at some small resolution (32^3). The small bricks are dynamically generated from the large ones on request.

Since requesting a large brick for every small brick would only increase the disk traffic, we keep an additional brick cache in memory to source these copies from. Our cache uses a standard LRU strategy. This is advantageous when the working set of the data fits into the host memory, however when the working set

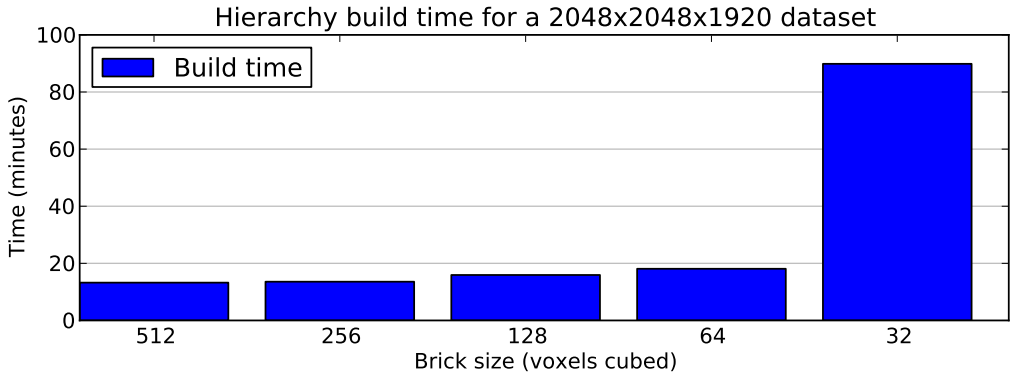

Figure 3.8: Time to build bricked representation for a medium-sized dataset, as a function of brick size. Renderers desire small bricks to perform efficiently, but generating such bricks takes significant preprocessing resources.

exceeds the host memory we will evict entries before finishing a rendering. We stuck with this strategy since the working set often *does* fit into host memory, as established by Figure 3.5. If the renderer is to be used in an environment in which working sets are routinely larger than memory, an MRU strategy would be more appropriate.

Hierarchy Generation Reorganizing data into a set of bricks is mostly ignored in volume rendering literature, but becomes a significant bottleneck in real-world usage. Figure 3.8 shows the time our preprocess needs to generate this hierarchy, which increases sharply for small brick sizes. This time also increases with respect to dataset size. At the extreme scale, such data reorganization is completely infeasible: merely reading every datum might take months. We believe such reorganization will be feasible up to a few tens of terabytes. In practice, the authors and collaborators thereof tolerate this for up to 5 terabytes at present.

Rebricking the data at run time alleviates this problem. The data can be generated at very large brick sizes, enabling fast conversion and effective disk throughput, and then dynamically rebricked to very small sizes. Both disk and renderer deal with their ideal cases, then. The ‘Rebricking’ case of Figure 3.4 shows performance in this mode.

Algorithm 2 Greedy algorithm: request all bricks at all resolutions.

```
ReportMissingBrick(b)
repeat
    LoD++
    b = LookupBrick(ray, LoD)
    if Missing(b) then

ReportMissingBrick(b)
    end if
until ¬Missing(b)
```

Algorithm 3 Global algorithm: only request bricks required to satisfy the final rendering request.

```
ReportMissingBrick(b)
repeat
    LoD++
    b = LookupBrick(ray, LoD)
until ¬Missing(b)
```

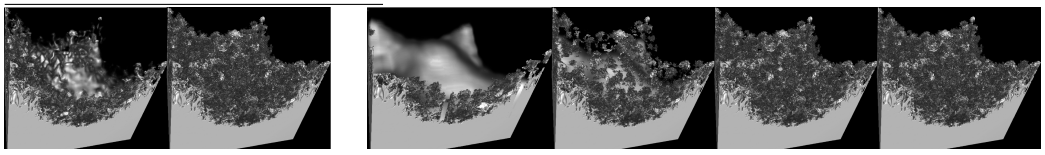

Figure 3.9: The effect of multiple brick replacement strategies. Renderings are select intermediate frames from the corresponding strategy. 'Greedy' strategies converge quicker and produce more densely-packed intermediate progress.

3.5.3 CPU/GPU Interface

Point (4) in our overview is the efficient communication of the ray guidance information from the location it is generated—the GPU—to the location it is utilized—the IO layer of a volume renderer. This section details how that communication happens.

We utilize a GPU-based hash table to store this data, though we note that we really only require a set. That is, our keys (brick IDs) are our values, and we only care about their *presence* in the table, which we will read back and process as a list later. A list would work as well, but a hashing scheme allows concurrent inserts to proceed with less contention. During rendering, a ray may write into this table to indicate that it needs a non-resident brick to continue (see Figure 3.2, (c)). This small table will be read back from the GPU at the end of a frame and utilized to fill the volume pool with new data.

As locks do not exist in current GLSL versions (and potentially never will), lock-free structures are the only hazard-prone data structures that can be correctly implemented. Crassin et al.[49] workaround this by using multiple render targets: each pixel has its own unique set of memory to write into, and so there are no write hazards. Our scheme requires significantly less memory, but we must deal with these write hazards.

Hash Table Parameters

We map from the 4D index of the requested brick (spatial index + LoD) to a unique 1D index in the hash table. The mapping we utilize is simply converting the 4D index into its equivalent 1D form, as if it were stored in a 1D array. We increment the index by 1 so that we may use 0 to indicate that there is no entry at a location.

In a normal concurrent hash table, a lock is acquired for a table or bucket before an access. In lock-free data structures the primitives used to implement locks are instead used directly on the data values in question. Inserts into our table proceed mostly as described in previous work [63]. In the face of concurrent writes, this operation fails, and we attempt to probe a few times (presently: 10) before giving up.

The critical piece to note is: *it is not an error if a missing brick is not recorded*. As long as *some* missing bricks are recorded, the next frame *will* make progress. Each ray is either: finished, able to make progress, or unable to make progress due to a lack of bricks that it requires. Since our hash table only contains entries for bricks that were requested by a ray, then an invariant of our system is that: volume

rendering is done, or there exists at least one ray that can make progress.

Strategies for Loading Coarser Bricks

When the resolution required is missing during ray-casting, a ray's brick requests can be what we call 'greedy' or 'global'. In the 'greedy' case, the ray requests intermediate levels of detail along the way, flooding the hash table with requests that *this* ray wants. In the 'global' case, each ray only requests what it absolutely needs, leaving space for other rays to request what they need. These cases are visually depicted and expounded in Figure 3.9.

The intuitive interpretation is that the 'greedy' approach will produce a more responsive, iteratively-refined image, whereas the 'global' approach will generate the final correct image quickest. However, the authors were surprised to find that the 'greedy' approach both produces more pleasing progress information *and* converges in the fewest number of frames. This is because it allows a ray to sample at its final resolution quickly, which can cause earlier ray termination.

3.6 Conclusions, Limitations, & Future Work

In this work, we have introduced an efficient, out-of-core, ray guided GPU volume renderer that scales to extremely large data. The system pulls inspiration from a patchwork of recent renderers, combining the advantages of many and reimplementing some ideas in light of modern GPU features. We have also contributed an evaluation and discussion of the tradeoffs inherent in the development of a modern ray-guided volume renderer.

Based on the data here, we conclude that a ray-guided volume renderer should work with bricks that are, on disk, 64^3 or larger. This minimizes time spent doing IO (Figure 3.7), and makes data layout irrelevant, obviating the need for a complicated component of the code. Since the required memory shrinks with the brick size, generating 32^3 or even 16^3 bricks on-the-fly is desirable, though exactly which size is unfortunately too data-specific to answer generally. While 'bzlib' gives ideal compression ratios, it is very slow to decompress, and therefore most implementations will want to utilize 'LZ4' compression. A cache is a boon when data will not fit in GPU memory but will fit in the host's memory.

We have made a best-effort attempt to design both favorable and unfavorable conditions with which to test a volume renderer, but it is possible some considerations have been omitted. In particular, this renderer and many others rely heavily

on the assumption that rays will saturate quickly. Subjectively, we have found this to be overwhelmingly valid for all our work in volume rendering, but this is not a rule and has not been thoroughly evaluated.

A second issue is the rendering modes evaluated. While our system supports 2D transfer functions as well, all performance results presented here utilized the 1D transfer function mode. Advanced rendering effects as well, such as those similar to ambient occlusion [66], are omitted. Such effects should have a variable impact, positively correlating to the proportion of rendering vs. IO times presented in Figure 3.4. Screen-space methods may provide acceptable quality without (comparatively) impacting performance.

Finally, reformatting the data into a bricked hierarchy continues to be the bane of high-performance volume rendering. This result is not expounded often enough in the literature. We hope this paper helps to reiterate to the community that the FLOPs may be free, but data movement will kill performance.

Most importantly, we have contributed an evaluation and discussion of the issues inherent in the development of a ray guided volume renderer. As has been demonstrated, many of these choices are not as clear as previous reports may have inadvertently implied. The results presented in this work depict the tradeoffs, to aid system designers in creating volume renderers that suit their particular environment.

We hope to extend this work to more diverse visualization scenarios. Ray-guidance-based isosurface generation is a natural candidate for these ideas. Furthermore, a common use case is combining an isosurface with volume rendering, which has the potential to significantly change such aspects as the working set size. The general idea that rendering should drive the visualization pipeline—as opposed to passively consuming the output of earlier operations—is one that is applicable in a much wider sense than that presented here.

3.7 Data and Performance Details

We tested our renderer with a plethora of datasets, both real and artificially created. For space reasons, we discuss only a subset that proved to be a reasonable sampling of our available data. Renderer performance is depicted for a variety of datasets in Table 3.1. We discuss these in order of increasing size here.

Two small datasets are the Bonsai tree (“Bonsai”) and “Aneurysm” datasets (Figure 3.10, top, left & middle). While small by today’s standards, effective empty space leaping and early ray termination still double the performance (Table 3.1, note how performance doubles with smaller brick sizes).

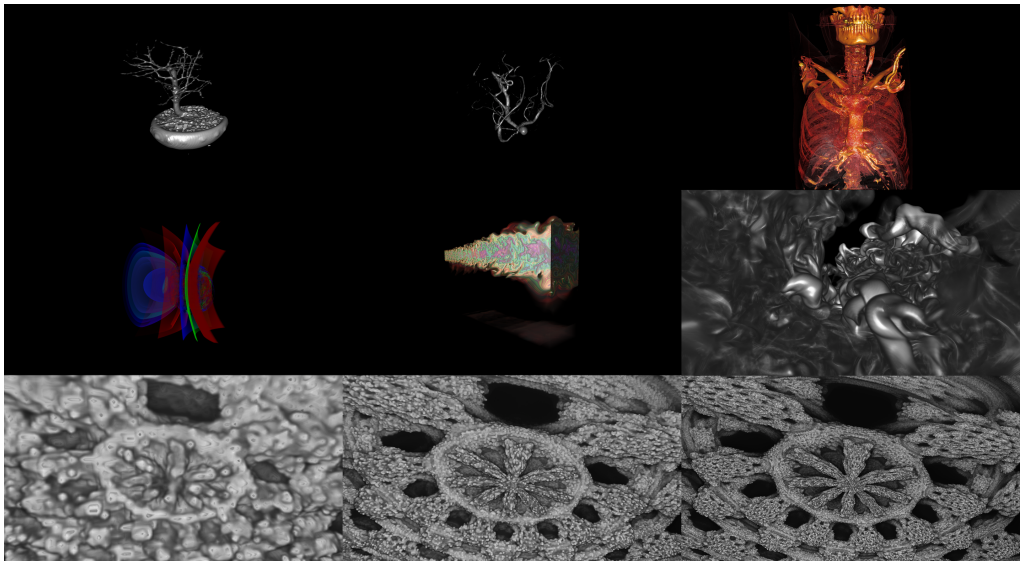

Figure 3.10: Selected frames from interactions used to record data for Table 3.1 or Figure 3.4

The "WholeBody" dataset (Figure 3.10, top, right) is a contrast-enhanced CT scan of a human body. As sometimes happens in the biomedical domain, these data have limited slice resolution but a plethora of slices. Coarser resolutions must be careful to downsample anisotropically, else the in-plane resolution washes out too quickly.

"Velocity" (center, left) comes from the simulation of an exploding star; we chose this dataset because our ideal transfer function for it is quite transparent, preventing the renderer from taking advantage of early ray termination. Highly transparent transfer functions that still produce informative results are a rarity but still occur. For these data, the additional overhead of small bricks can have a fairly drastic effect on performance. This dataset is one of the rare datasets for that lighting actually makes the visualization *more difficult* to interpret, and so we always render this dataset with lighting off.

The "magnitude" dataset (center, middle) comes from a combustion simulation and represents another intermediate step towards larger data. The lower half of this dataset actually has a very faint trace of data, which causes the renderer to sample densely. The expense of computing lighting information for fragments that ultimately contribute very little has a notable effect on performance.

The Richtmyer-Meshkov Instability ("RMI", Figure 3.1 right and Figure 3.10,

Table 3.1: Per-frame rendering time at 6 different brick sizes, for a variety of datasets depicted in Figures 3.10 and 3.1. **Optimal brick sizes** are dataset dependent.

	Rendering Time (ms)				
Dataset	16^3	32^3	64^3	128^3	256^3
Bonsai	**16**	20	26	31	28
Head Aneurysm	**27**	34	40	55	85
Whole Body	140	94	82	77	**67**
Velocity	376	208	146	118	**110**
Magnitude	132	93	**80**	82	85
RMI	**60**	64	61	67	67
Visible Human	**34**	37	47	67	123
Mandelbulb1k	**21**	**21**	**21**	22	25
Mandelbulb4k	**27**	30	37	47	47
Mandelbulb8k	**33**	37	45	60	78

center, right) and the Visible Human (Figure 3.1 left) are popular datasets in the volume rendering literature; details can be found in previous work.

We created a series of "Mandelbulbs" at various resolutions ($1k^3$, $4k^3$, $8k^3$). These are an extension of the mandelbrot fractal into 3 dimensions. This has many of the same properties of the data used in Crassin et al. [49], in which Perlin noise was added to a large bone scan to increase the sampling requirements. We create the high-resolution features *a priori*, so no GPU features were used to accelerate this process. At equivalent resolutions to that work, we see double to an order of magnitude improved performance, but for this work we report results at 1080p HD resolution. A descriptive view of the Mandelbulb is given in Figure 3.3 and there are close-ups visible in Figure 3.10 (bottom row; center, right).

3.8 Source Code

The renderer used in this work is freely available, as part of the ImageVis3D [2] package. We encourage others to reproduce and build upon our results.

Table 3.2: Dataset properties for test datasets.

Dataset	**Resolution**				**Size**
Bonsai	256	× 256	× 256	8 bpp	16 MB
Head Aneurysm	512	× 512	× 512	16 bpp	256 MB
Whole Body	512	× 512	×3172	16 bpp	1.5 GB
Velocity	1000	×1000	×1000	16 bpp	1.9 GB
Magnitude	2025	×1600	× 400	16 bpp	2.4 GB
RMI	2048	×2048	×1920	8 bpp	7.5 GB
Visible Human	1728	×1008	×1878	32 bpp	12.2 GB
Mandelbulb1k	1024	×1024	×1024	8 bpp	1 GB
Mandelbulb4k	4096	×4096	×4096	8 bpp	64 GB
Mandelbulb8k	8192	×8192	×8192	8 bpp	512 GB

Chapter 4

Multi-scale-parallel volume rendering

Data sets of immense size are regularly generated on large scale computing resources. Even among more traditional methods for acquisition of volume data, such as MRI and CT scanners, data that is too large to be effectively visualized on standard workstations is now commonplace.

One solution to this problem is to employ a 'visualization cluster,' a small- to medium- scale cluster dedicated to performing visualization and analysis of massive data sets generated on larger scale supercomputers. These clusters are designed to fit a different need than traditional supercomputers, and therefore their design mandates different hardware choices, such as increased memory, and more recently, graphics processing units (GPUs). While there has been much previous work on distributed memory visualization as well as GPU visualization, there is a relative dearth of algorithms that effectively use GPUs at a large scale in a distributed memory environment. In this work, we study a common visualization technique in a GPU-accelerated, distributed memory setting, and present performance charactersitcs when scaling to extremely large data sets.

4.1 Introduction

Visualization and analysis algorithms, volume rendering in particular, require extensive compute power relative to data set size. One possible solution is to use the large scale supercomputer that generated the data. However it can be difficuilt to

reserve and obtain the compute resources required for viewing large data sets. An alternative approach, one explored in this work, is to use a smaller scale cluster equipped with GPUs. Such a cluster can provide the needed computational power at a fraction of the cost—provided the GPUs can be effectively utilized. As a result, a semi-recent trend has emerged to procure GPU-accelerated visualization clusters dedicated to postprocessing the data generated by high-end supercomputers; examples include ORNL's Lens, Argonne's Eureka, TACC's Longhorn, SCI's Tesla-based cluster, and LLNL's Gauss.

Despite this trend, there have been relatively few efforts studying distributed memory, GPU-accelerated visualization algorithms that can effectively utiliaze the resources available on these clusters. In this work, we report parallel volume rendering performance characteristics on large data sets for a typ[ical machine of this type.

Our system is divided into three stages:

1. *An intelligent pre-partitioning* that is designed to make combining results from different nodes easy.

2. *A GPU volume renderer* to perform per-frame volume rendering at interactive rates.

3. *MPI-based compositing* using a sort-last compositing framework.

Müller et al. presented a system similar to our own that was limited to smaller data sets [67]. We have extended the ideas in that system to allow for larger data sets, by removing the restriction that a data set must fit in the combined texture memory of the GPU cluster and adding the ability to mix in CPU-based renderers, enabling us to analyze the parallel performance on extremely large data sets. The primary contribution of this component of our work is an increased understanding of the performance characteristics of a distributed memory GPU-accelerated volume rendering algorithm at a scale (256 GPUs) much larger than previously published. Further, the results presented here (data sets up to 8192^3 voxels) represent some of the largest parallel volume renderings attempted thus far.

Our system and benchmarks allow us to explore issues such as:

- the balance between rendering and compositing: a well-studied issue with CPU-based rendering, but currently with unclear performance tradeoffs for rendering on GPU clusters;

- the overhead of transferring data to and from a GPU;

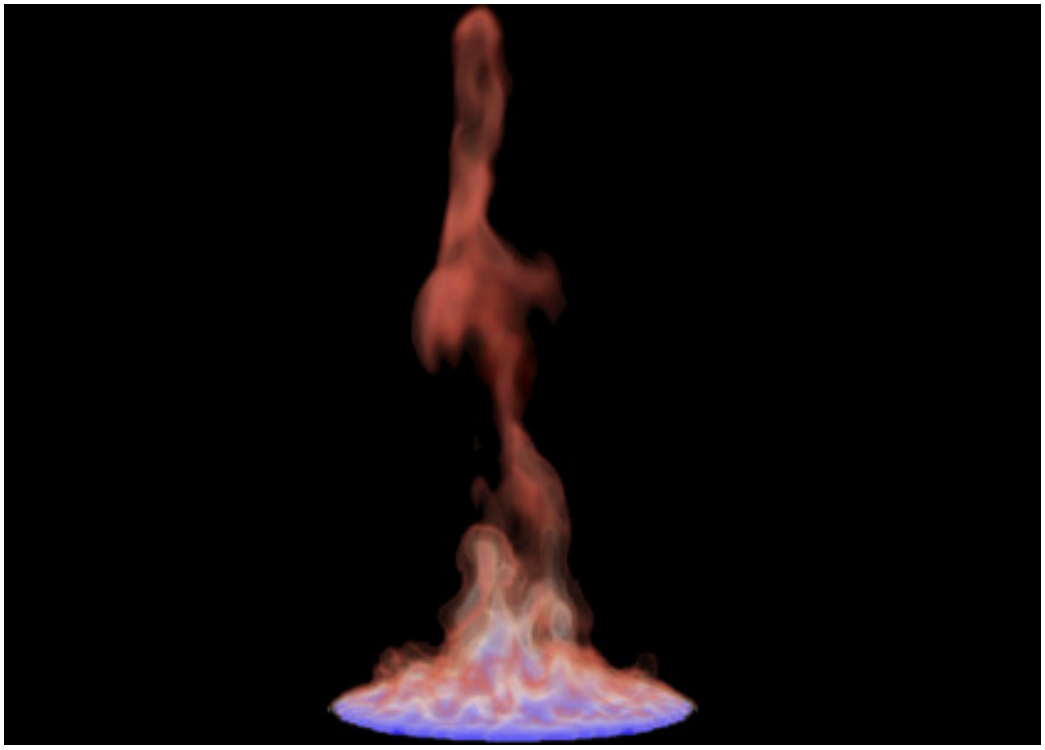

Figure 4.1: Output of our volume rendering system with a data set representing a burning helium flame.

- the importance of process-level load balancing; and
- the viability of GPU clusters for rendering very large data.

This chapter is organized as follows. In Section 4.2, we overview previous work in parallel compositing and GPU volume rendering. In Section 4.3, we outline our system in detail. Section 4.4 discusses our benchmarks and presents their results. Finally, in Section 4.5 we draw conclusions based on our findings.

4.2 Previous work

Volume rendering in a serial context has been studied for many years. The performance of the basic algorithm [68] was improved significantly by incorporating empty space leaping and early ray termination [57]. Max provided one of the

earliest formal presentations of the complete volume rendering equation in [69]. Despite significant algorithmic advances from research such as [57], the largest increase in performance for desktop volume renderers has come from taking advantage of the 3D texture capabilities [18, 17, 70] and programmable shaders [19] available on modern graphics hardware.

Extensive research has been done on parallel rendering and parallel volume rendering. Much of this work has focused on achieving acceptable compositing times on large systems. Molnar et al. conveyed the theoretical underpinnings of rendering performance [71]. Earlier systems for parallel volume rendering relied on direct send [72, 73], which divides the volume up into at least as many chunks as there are processors, sending ray segments (fragments) to a responsible tile node for compositing via the Porter and Duff *over* operator [74]. These algorithms are simple to implement and integrate into existing systems, but have sporadic compositing behavior and the potential to exchange a large a number of fragments, straining the network layers when scaling to large numbers of processors. Tree-based compositing algorithms feature more regular communication patterns, but impose an additional latency that may not be required, depending on the particular frame and data decomposition. Binary swap and derivative algorithms are a special case of tree-based algorithms that feature equitable distribution of the compositing workload [75], Despite advancements in compositing algorithms, network traffic remains unevenly distributed in time, and thus high-performance networking remains a necessity for subsecond rendering times on large numbers of processors.

In the area of distributed memory parallel volume rendering of very large data sets, the algorithm described by Ma et al. in [73] has been taken to extreme scale in several followuip publications. In [60], data set sizes of up to 3000^3 are studied using hundreds of cores. In this regime, the time spent ray casting far exceeds the composite time. In [76, 77] the data set sizes range up to 4480^3, while core counts of tens of thousands are studied. In [78] the benefits of hybrid parallelism are explored at concurrency ranges going above two hundred thousand cores. For both of these studies, when going to extreme concurrency compositing time becomes large and dominates ray-casting time. This suggests that a sweet spot may exist with GPU-accelerated distributed memory volume rendering. By using hardware acceleration, the long ray casting times encountered in [60] can be overcome. Simultaneously, the emerging trend of composite-bound rendering observed in [77] and [78] will be mitigated by the ability to use many fewer nodes to command the same compute power.

Numerous systems have been developed to enable parallel rendering in existing

software. Among the most well-known is Chromium [79], a rendering system that can transparently parallelize OpenGL-based applications. The Equalizer framework boasts multiple compositing strategies, including an improved direct send [80]. The IceT library provides parallel rendering with a variety of sort-last compositing strategies [81].

There has been less previous work studying volume rendering on multiple GPUs. Strengert et al. developed a system that used wavelet compression and adaptively decompressed the data on small GPU clusters [82]. Marchesin et al. compared a volume that ran on two different two-GPU configurations: two GPUs on one system, and one GPU on two networked systems [83]. The use of just one or two systems, coupled with an in-core renderer, artificially constrained the data set size. Müller et al. developed a distributed memory volume renderer that ran on GPUs [67]; their system differs from ours in a few key ways. First, we use an out-of-core renderer and therefore can exceed the available texture memory of the GPU by also utilizing CPU memoryor disk. To further reduce memory costs, we compute gradients dynamically in the GLSL shader [19] obviating the need to upload a separate gradient texture. This also has the benefit of avoiding a pre-processing step that is normally software-based in existing general-purpose visualization applications (including the one we chose to implement our system within) and can be time consuming for large data sets. Further differentiating our system and in line with recent trends in visualization cluster architectures, we enable the use of multiple GPUs per node. Müller et al. used a direct send compositing strategy [72, 73] whereas we use a tree-based compositing method [81]. Finally, and most importantly, we report performance results for substantially more GPUs and much larger data sets, detailing the scalability of GPU-based visualization clusters. We therefore believe our work is the first to evaluate the usability of distributed memory GPU clusters for this scale of data.

4.3 Architecture

We implemented our remote rendering system inside of VisIt [45, 13] which is capable of rendering data in parallel on remote machines. The system is comprised of a lightweight 'viewer' client application, connected over TCP to a server that employs GPU cluster nodes. All rendering is performed on the cluster, composited via MPI, and images (optionally compressed via zlib) are sent back to the viewer for display. Example output from our system is in Figure 4.1.

Although VisIt provided a good starting point for our work, we needed to make

significant changes in order to implement our system. In this section, we highlight the main features of our system, taking special care to note where we have deviated from existing VisIt functionality.

4.3.1 Additions to VisIt

Multi-GPU access

At the outset, VisIt's parallel server supported only a single GPU per node. We have revamped the manner in which VisIt accesses GPUs to allow the system to take advantage of multi-GPU nodes. When utilizing GPU-based rendering, each GPU is matched to a CPU core that feeds data to that GPU. Additionally, when the number of CPU cores exceeds the number of available GPUs, we allow for the use of software-based renderers on the extra CPUs. This code has been contributed to the VisIt project.

Partitioning

VisIt contained a number of load decomposition stratgies prior to our work. However, we found these stratgies to be insufficient for a variety of reasons:

1. **Brick-based** Equalizing the distribution of work in VisIt was entirely based on *bricks*, or pieces of the larger data set. Our balancing algorithms use the time taken to render the previous frame to determine the weighted distribution of loads.

2. **Master-slave** Dynamic balance algorithms in VisIt are based on a *master* node that tells slaves to process a brick, waits for the slaves' completion, and then sends them a new brick to process. We implemented a flat hierarchy, as seems to be more common in recent literature [84, 67].

3. **Compositing** *Most importantly*, for our object-based decomposition to work correctly, we needed a defined ordering to perform correct compositing. The load balancing and compositing subsystems in VisIt were independent prior to our work.

Our system relies on a *k*d-tree for distributing and balancing the data. The spatial partitioning is done once initially and can be adaptively refined by the rendering times from previous frames. The initial tree only considers the number

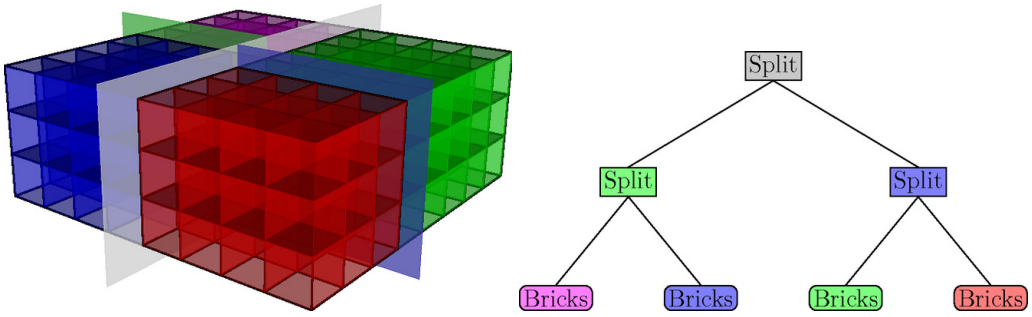

Figure 4.2: Decomposition and corresponding kd-tree for an 8x8x3 grid of bricks divided among 4 processors. Adjacent bricks are kept together for efficient rendering and compositing. A composite order is derived dynamically from the camera location in relation to the splitting planes. Note that the number of leaves in the tree is equal to the number of processes in the parallel rendering job.

of bricks in the available data set and attempts to distributed them evenly among processes, to the extent that is possible. When using static load balancing, this decomposition is invariant for the life of the parallel job. Figure 4.2 depicts a possible configuration determined by the partitioner, and shows the corresponding *k*d-tree.

When the dynamic load balancer is enabled, we use the last rendering time on each process to determine the next configuration. In our initial implementation, the metric we utilized was the total pipeline execution time to complete a frame. This included the time to read data from the disk, as well as the compositing time, among other inputs. However, we found that I/O would dwarf the actual rendering time. Further, compositing time is not dependent on the distribution of bricks. This therefore proved to be a poor metric. Switching the balancer to use the total render time for all bricks on that process gave significantly better results.

In order to compare different implementations, we implemented multiple load balancing algorithms, notably those described in Marchesin et al. and Müller et al.'s work [84, 67]. In both cases, leaf nodes represent processes, and each process has some number of bricks assigned to it. In the Marchesin-based approach, we start at the parents of the leaf nodes and work our way up the tree, searching for imbalance among siblings. If two siblings are found to be imbalanced, a single layer of bricks is moved along the splitting plane. This process continues up the root of the tree, at which time the virtual results are committed and the new tree dictates the resulting data distribution. In the Müller-based approach, we begin

Component	*Lens*	*Longhorn*
Number of nodes	32	256
GPUs per node	2	2
Cores per node	16	8
Graphics Card	NVIDIA 8800 GTX	NVIDIA FX 5800
Per-node memory	64 Gb	48 Gb
Processors	2.3 GHz Opterons	2.53 GHz Nehalems
Interconnect	DDR Infiniband	Mellanox QDR Infiniband

Table 4.1: Configuration of the GPU clusters utilized.

with the root node and use a pre-order traversal to find imbalance among siblings. Once imbalance is found, the process stops for the current frame. Instead of blindly shifting a layer of bricks between the siblings, the method derives the average rendering cost associated with a layer of bricks along the split plane, and shifts this layer if the new configuration is projected to improve rendering time.

In addition to achieving a relatively even balance among the data, the *k*d-tree is used in the final stages to derive a valid sort-last compositing order.

4.4 Evaluation

We implemented and tested our system on *Lens*, a GPU-accelerated visualization cluster housed at ORNL. However, we were only able to access 16 GPUs on that machine. In order to access a larger number of GPUs, we transitioned to *Longhorn*, a larger cluster housed at the Texas Advanced Computing Cluster (TACC). Specifications for each cluster are listed in Table 4.1. Due to machine availability and configuration, we were not able to fully utilize either machine.

4.4.1 Rendering times

The two dominant factors in distributed memory visualization performance are the time taken to render the data and the time taken to composite the resulting sub-images. These have the largest impact on usability because they comprise the majority of the latency a user experiences: the time between when the user interacts with the data and when the result of that interaction are displayed.

Our data originated from a simulation performed by the Center for Simulation of Accidental Fires and Exploisions (C-SAFE), desinged to study the instabilities

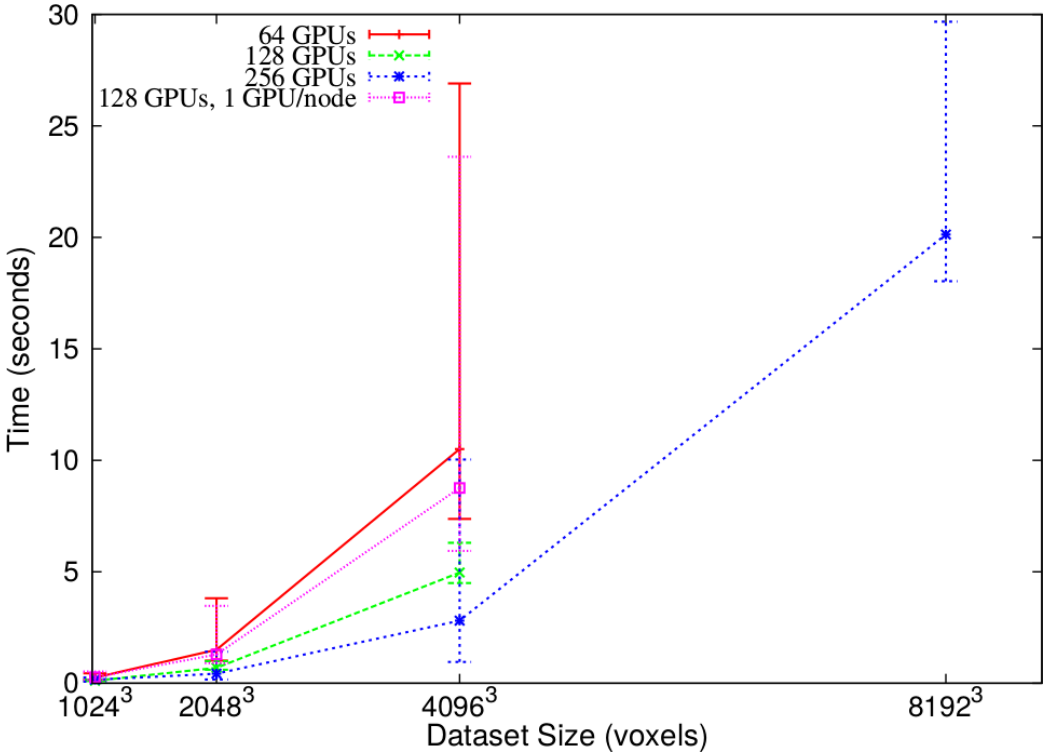

Figure 4.3: Overal rendering time when rendering to a 1024x768 viewport on Longhorn. This incorporates both rendering and compositing, and therefore shows the delay a user would experience if they used the system on a local network. Data points are the average across many frames, and error bars indicate the rendering timesor the slwoest and quickest frames, respectively. For these results we used a domain consisteing of 13^3 bricks (varying brick size) with the exceptions that all runs in the 128 GPU cases used 8^3 bricks, and the run for the 8192^3 data set was done using 32^3 bricks.

Figure 4.4: Rendering time as a function of brick size. Error bars indicate the minimum and maximum times recorded, across all nodes, for that particular brick size; high diswparity indicates the rendering time per-brick was highly variable, and load imbalance was therefore likely. All tests were done with a 4096^3 data set statically laod balanced across 128 GPUs on 64 nodes, using a scripted camera that requested the same viewpoints each run. Note that the choice of brick size matters little in the average case, but bricks using non-power-of-two sizes give widely varying performance. Though raw data shows it is only hundreths of a second faster than 256^3.

in a burning helum flame. In order to study performance at varying resolutions, we resampled this data to 1024^3, 2048^3, 4096^3, and 8192^3, at a variety of bricks sizes. We then performed tests, varying data resolution, image resolution, choice of brick size, and number of GPUs, up to 256. Unless noted otherwise, we divided the data into a grid of $8x8x8$ bricks for parallel processing (larger data sets used larger bricks), and rendered into a 1024x768 viewport.

Figure 4.3 shows the scalability on the *Longhorn* cluster. The principal input that affects rendering time is the data set size, as one might expect. These runs were all done using 2 GPUs per node, except the "128 GPUs, 1 GPU/node" case that was run on 128 nodes each accessing a single GPU. With very large data, there is a modest increase in performance for this experimental setup.

As can be seen in Figure 4.4, the brick size *generally* has little impact on performance. A parallel volume renderer's performance is, however, dictated by the slowest component, and therefore the average rendering time is less important than the maximum rendering time. Taking that into account, it is clear that brick sizes that are not a power of two are poor choices. Dropping down to 128^3, we can see that per-brick overhead begins to become noticeable, impacting overall rendering times. We found larger bricks sizes of 512^3 give the absolute best performance, with 256^3 a good choice as well, as the differences are minor enough that they may almost be considered sampling error. Of course, such recommendations may be specific to the GPUs used in *Longhorn*

We were initially surprised to find that the image resolution, while relevant, was not a significant factor in the overall rendering time. When developing single GPU applications that run on a user's desktop, our experience was the opposite: that image size does play a significant role in performance. We at first thought this was due to skipping bricks that were 'empty' under our transfer function—our domain

is perfectly cubic, yet as is displayed in Figure 4.1 very little of the domain is actually visible—but even after changing to a transfer function with no "0" values in the opacity map, rendering times changed very little. We concluded that the data sizes are so large compared to the number of pixels rendered that the image size is dwarfed by comparison.

In our initial implementaion on *Lens*, we noticed that we began to strain the memory allocators while rendering a 3000^3 data set, as we approached low memory conditions. Our volume renderer automatically accounts for low memory conditions and attempts to free unused bricks before failing outright. However, an operating system will thrash excessively before finally deciding to fail an allocation, and therefore during the time leading up to a failed allocation, performance will drop considerably. Worse, we are working in a large existing code base, and attempting to manage allocations outside our own subsystems would prove unwieldy. As such, we found the original scheme to be unstable; the rendering system would create memory pressure, causing other subsystems to fail an allocation in areas where it may be difficult or impossible to ask our volume renderer to free up memory.

To solve this problem, we render the data in a true out-of-core fashion: bricks are given to the renderer, rendered into a framebuffer object, and immediately thrown away. One might expect that out-of-core algorithms would have more per-block overhead and therefore be slower than an in-core algorithm. As shown in Figure 4.5, the out-of-core approach actually outperforms the analogous in-core approach even when there is sufficient memory to hold the data set at once. The reasoning turned out to be that bricks were searched for in a logarithmic data structure; the conservative approach taken by the out-of-core algorithm meant that the container maxed out at a single element, accounting for a minor performance improvement.

Readback and compositing

In earlier results, particularly with GPU-based rendering architectures, the community was generally concerned with the time required to read the resulting image data from the GPU and into the host's memory [83]. Our study did not provide corroboration of this concern, which we interpret as a positive data point with respect to evolving graphics subsystems. Our system did demonstrate that this time increased as the resolution grew, but as can be seen in Table 4.2, even at 1024x768 this step took only thousandths of a second.

The time required for image composition is significantly reduced when taking

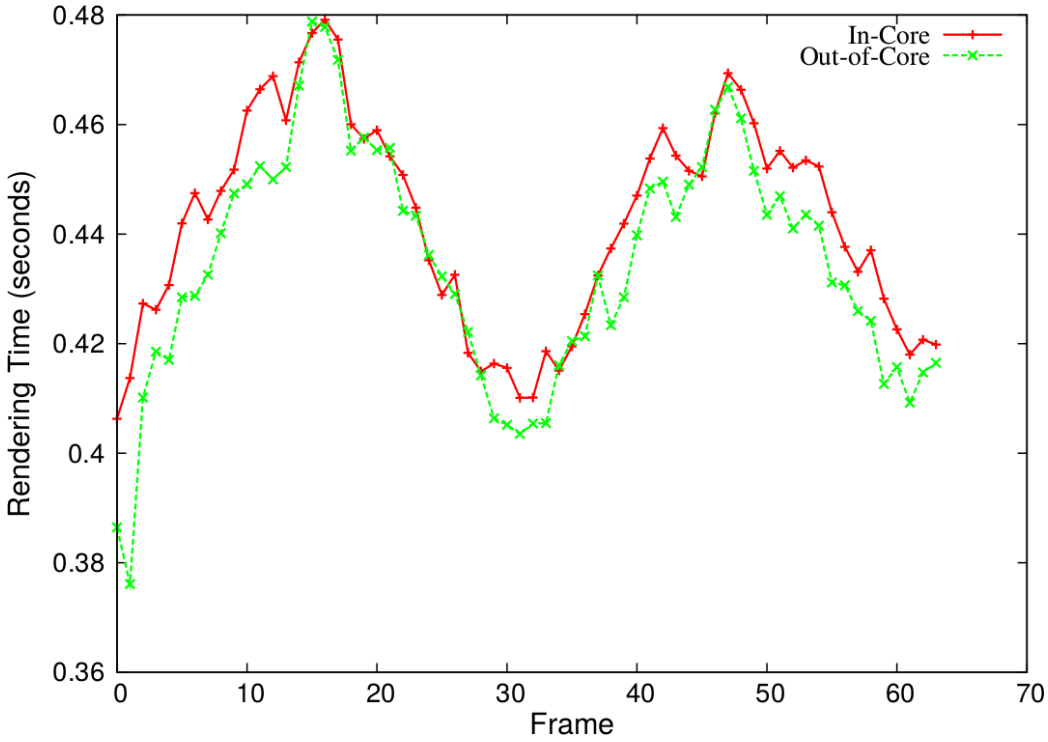

Figure 4.5: Per frame rendering times for the in-core and out-of-core approaches to rendering a 1024^3 data set (that fits comfortably in memory) across 16 GPUs. Additional processing in the out-of-core case does not negatively impact performance.

Dataset size	**Rendering (s)**	**Readback (s)**	**Compositing (s)**	**Total (s)**
1024^3	0.06141	0.00328	0.06141	0.12610
2048^3	0.35107	0.00377	0.07673	0.43157
4096^3	2.50984	0.00377	0.29533	2.80894
8192^3	19.60648	0.00373	0.51799	20.12820

Table 4.2: Breakdown of different pipeline stages for various data set sizes on 256 GPUs rendering into a 1024×768 viewport. All times are in seconds. The 1024^3, 2048^3, and 4096^3 case used 13^3 bricks (varying brick size); the 8192^3 case used 32^3 bricks, making each brick 256^3 voxels. Compositing time rises only artificially; if a node finishes rendering before other nodes, the time it must wait was included under 'Compositing' due to an artifact of our sampling code. Thus, the data imply that larger data sets see more load imbalance.

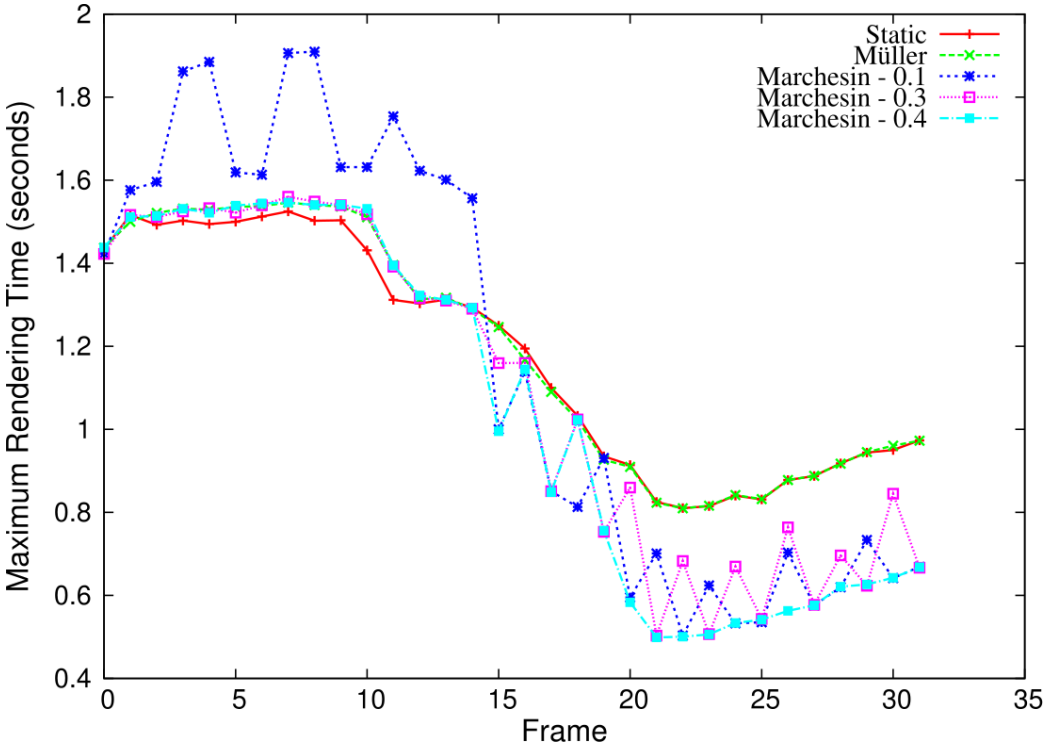

Figure 4.6: The maximum rendering time across all nodes under various load balancing algorithms. The numbers after the 'Marchesin' algorithms indicate thresholds: rendering disparity under these thresholds is ignored.

advantage of the GPUs available in the visualization vluster. Since a GPU can render much faster than a software-based renderer, one can achieve acceptable rendering performance using far fewer nodes. Compositing, as it scales with the number of nodes involved in the compositing process, improves significantly by utilizing many fewer nodes.

4.4.2 Load balancing

We also sought to examine the utility of load balancing algorithms for our system. We have implemented the algorithms as presented in two recent parallel volume rendering papers, and compared rendering times to each other and to a statically balanced case. Figure 4.6 illustrates the comparisons, where the times shown are the maximum across all processes.

We did a variety of experiments with multiple load balancer implementations, using 8 or 16 GPUs. Our initiali flythrough dequence proved to be inappropriate for the application of a load balancer, as there was not enough imbalance in the system to observe a significant benefit. We then attempted to a 'zoom out' flythrough, but rendering times decreasing on *all* nodes was not a case the the balancers we implemented could effectively deal with: we found many cases where the balancers would shift data to a node that was previously idle or at least doing very little work, and a frame or two later the workload on such nodes would spike. This occurred because these nodes had both 1) received new data as part of the balance and, 2) retained old data as part of the initial decomposition or previous balancing processes. The sudden additional workload of previously invisible bricks caused these nodes to overcompensate, sending data to other "idle" nodes—nodes that would experience the sample problem in subsequent frames.

In previous work, authors have praised the effect load balancing has when zooming *in* to a data set. This naturally creates imbalance, as some nodes end up with data that are not rendered under the current camera configuration, and therefore the node has no work to do.

With the implementations we recreated as faithfully as possible, we did find that zooming in to the data set was a task that was well-suited for load balancing. Still, we encountered issues even with this case. For the algorithm given in [84] we observed that the data would move back and forth between nodes quite frequently, having a negative impact on overall rendering time. We therefore introduced a 'threshold' parmaeter to the existing algorithm, in an attempt to limit this 'ping-pong' behavior. As we move up the tree, imbalance between the left and right subtrees is subject to this threshold; if it does not exceed the threshold, the imbalance is ignored. This is a very useful parameter for ensuring that we do not move data too eagerly. Generally, setting this threshold too high will yield behavior equivalent to the static case; setting it to low leads to a considerable amount of unnecessary data shifting, and we found that this in many cases overcompensated for minor, expected variations (such as those one might expect from differing brick sizes; see Figure 4.4). For example, see Figure 4.6, in which low thresholds display an obvious 'ping-pong' effect as nodes overcompensate for increased rendering load.

Müller et al. describe a different balancing system [67]. This system calculates the average cost of rendering a brick, and therefore has a clearer idea of what the effect of moving a given set of bricks will have on overall system performance. Further, they introduce additional parameters that add some hysteresis to the system, and this helped reduce the 'ping-pong' effect of nodes sending data to a neighbor

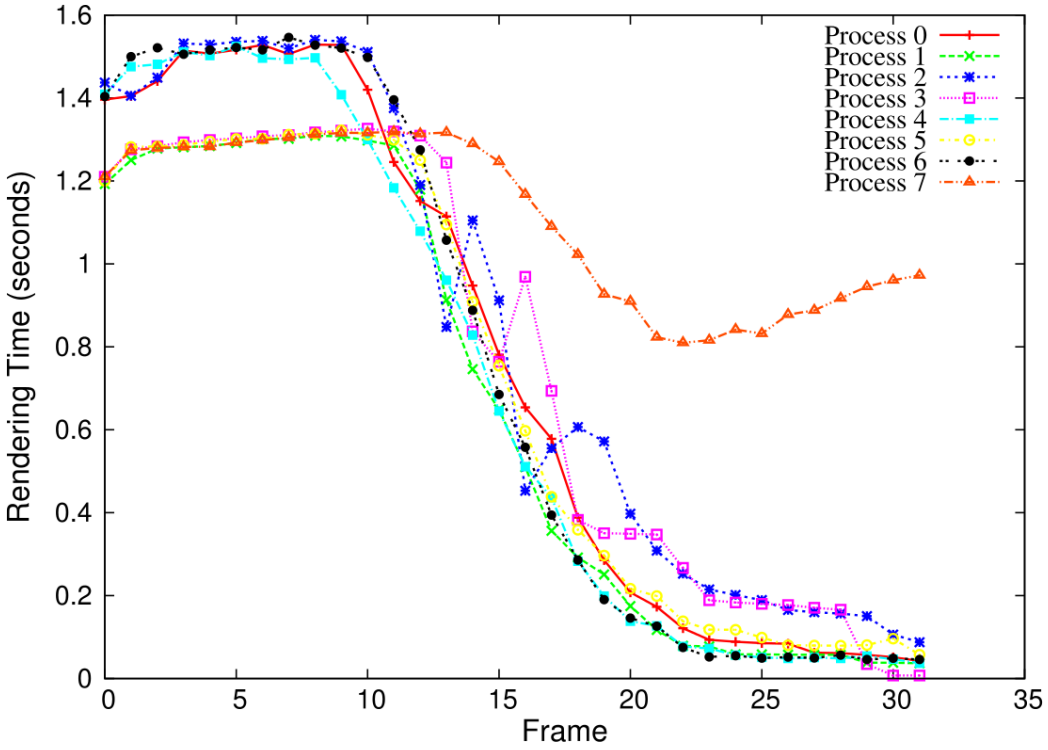

Figure 4.7: Per-process rendering times for the 'Müller' line given in Figure 4.6

just to receive it in the next frame when the neighbor becomes overloaded.

We found that this algorithm did do intelligent balancing for reasonable settings of these parameters, and the additional parametes could be successfully used to reduce excess data reorganization. Still we found two issues with the approach: for one, the assumnption that 'all bricks are equal' did not pan out for our work. Even assuming uniform bricks for a data set (true for our case, but likely not in a general system), one can see in Figure 4.4 that the time to render a brick sees variation on the orde rof a second. Secondly, despite experimenting with parameter settings, we found it difficult to get the algorithm to choose the 'best' set of nodes for balancing. In many cases, we found a particular node was an outlier, consistently taking the most time to render per frame. Yet it was common for this algorithm to balance different nodes. While rendering times would generally improve, the system's performance is determined by the slowest node, and therefore making the fast nodes faster does not help overall performance.

This was apparent in the tests described in Figure 4.6: the algorithm balanced

between some of the nodes, but the slowest node was never balanced, and therefore the user-visible performance fror this run was equivalenty to the static case. Figure 4.7 shows a more detailed analysis of the execution of the Müller algorithm that generated the data in Figure 4.6. The per-node rendering times in Figure 4.7 show that process 7 is usually the last process to finish and is often *considerably* slower than the next-to-last. As evident from the lack of sudden discontinuities in process 7's rendering times, however, no bricks from process 7 move to other nodes. Rendering times decrease across other nodes, but the maximum rendering time does not change.

We theorize that additions to the algorithm to learn weights for each individual brick would yield friutful results. Furthermore, the algorithm explicitly attempts to avoid visiting the entire tree, as an attempt to bound the maximum time needed to determine a new balancing. In our work, we did not observe cases where iterating through nodes in the tree had a measurable impact on performance, and feel that by doing so the algorithm could obtain the global knowledge it needs to balance data effectively. Both of these extensions are left for future work.

In Section 4.1 we noted a variety of quesitons that the design of our system allows us to answer.

- *Rendering vs. compositing*. As shown in Table 4.2, subsecond rendering times are achieved using a very small number of nodes, relative to previous work. This relieves a significant source of work for compositing algorithms.

- *Overhead of GPU transfer*. Table 4.2 shows readback time to be on the order of thousandths of a second for common image sizes. Measuring texture upload rates is difficult due to the asynchronous nature of OpenGL and current drivers, but we did not find evidence to suggest this was a bottleneck.

- *Importance of load balancing*. A dynamic load balancer can have a worthwhile impact on perfprmance. However, it can also lower the performance of the system. Load balancers generally come with some number of tunable parameters, and useful settings for these parametrs are difficult to determine *a priori*, and likely impossible for an end-user to effectively set. We observed that dynamic load balancing for volume rendering struggled in cases often encountered in real-world environments and, for this reason, believe there is still a gap before deploying these techniques in *production* systems. There is a great opportunity for future work in this area.

- *Viability*. As displayed in Figure 4.3 and Table 4.2, rendering extremely large data sets—up to 8192^3 voxels—is possible on relatively few nodes.

Further, data sets up to 2048^3 can be rendered at an interactive two frames per second.

4.5 Conclusions

With this study, we demonstrated that GPU accelerated rendering provides compelling performance for large scale data sets. Figure 4.3 demonstrates our system rendering data sets that are among some of the largest reported thus far, using far fewer nodes than previous work. This work shows that a multi-GPU node is a great fojundational 'building block' to compose larger systems cpaable of rendering very large data. As the price-performance ratio of a GPU is better (provided it can effectively parallelize the workload) than CPU-based solutions, this work makes the case for spending more visualization supercomputing capital on hardware accleration, and acquiring smaller yet more performant clusters.

Reports on the time taken for various pipeline stages demonstrate that PCI-E bus speeds are fast enough that readback performance is not as great of a concern as it was a few years ago. However, it remains to be seen if contention will become an issue if individual nodes are made 'fatter', utilizing additional GPUs. The 1 GPU per node given in Figure 4.3 suggest that multiple GPUs do contend for resources, but at this scale the differences are not yet significant enough for warrant moving away form the more cost-effective 'fat' node architecture. Given the relatively few nodes needed for good performance on large data, and external work scaling compositing worklaod out to tens of thousands of cores, it seems likely that the relatively 'thin' 2-GPU-per-system archietcture can be made to scale to even larger systems.

We would like to study our system with higher image resolutions, such as those available on a display wall, and larger numbers of GPUs. At some point, we expect compositing to become a significant factor in the amount of time needed to volume render large data, but we have not approached the cross-over point in this work, due to the use of 'desktop' image resolutions and low numbers of cores.

Our system allows substituting a Mesa-based software renderer when a GPU is not available. This provided a convenient means of implementation for an existing large software system, in particular because it allows pipeline execution to proceed unmodified through the rendering and compositing stages. However, tests very quickly showed that using software renderers when a GPU was available was not worthwhile, and usually ended up hurting performance more than helping. Therefore, for this work we traded access to more cores for the guarantee that we

will obtain GPUs for each core we do get.

An alternate system architecture would be to decouple the rendering process from the other work involved in visualization and analysis, such as data I/O, processing, and other pipeline execution steps. In this architecture, all nodes would read and process data, but processed, visualizable data would be forwarded to a subset of nodes for rendering and compositing. The advantage gained is the ability to tailor the available parallelism to the visualization tasks of data processing and rendering, which, as we have found, can benefit from vastly different parallel decompositions. The disadvantages are the overhead of data redistribution, and the wasted resources that arise from allowing non-GPU processes to sit idle while rendering.

Our compositing algorithm assumes that the images from individual processors can be ordered in a back-to-front fashion to generate the correct image. For this paper, we met this requirement by using regular grids that are easy to load balance in this manner. It should be trivial to extend this work to other grids, such as AMR or curvilinear ones. Extensions to handle unstructured grids would be difficult, however, but represent an interesting future direction.

Load balancing is an extremely difficult problem, and we have just scratched the surface here. The principal difficulty in load balancing is identifying good parameters to control how often and to what extent the balancing occurs. We would like to see ideas and algorithms that move in the direction of user-friendliness: determining the most relevant parameters and deriving appropriate values for them automatically.

Chapter 5

Large-scale data access

While additional cores and newer architectures, such as those provided by GPU clusters, steadily increase available compute power, memory and disk access has not kept pace, and most believe this trend will continue. It is therefore of critical importance that we design systems and algorithms that make effective use of off-processor storage. This work details our experiences using parallel file systems, details performance using current systems and software, and suggests a new API that has greater potential for increased scalability.

5.1 Introduction

Large scale parallelism is widely used not only to simulate complex phenomenon, but also to process the resultant data for understanding and insight. Parallel visualization and analysis applications exist to aid in this process, but I/O performance analysis generally takes a back seat to other metrics, such as renderer performance, with the justification that one only reads the data once and then spends much more time interacting with it. However, as we scale visualization tools up, we find that the time taken for the initial reading of the data is prohibitive, and becomes a significant barrier to the scientist's task: to understand their data and gain new insight in their science.

In developing any application, there are a number of practical concerns that must be considered to obtain acceptable performance. In the space of I/O, and especially distributed filesystems, many visualization and analysis developers pay little heed to these concerns. In this work we hope to elucidate some 'best

practices' for writing applications that will utilize parallel filesystems, as well as steer a convergence between application and filesystem developers.

5.1.1 Previous Work

Since the performance of most large scale visualization systems is clearly bound by I/O performance a significant body of literature exists to analyze and improve this component of parallel software. We provide a brief overview of a subset of that literature here.

The predominant file systems in use in modern supercomputers are the Network File System (NFS) filesystem and Lustre. NFS was originally developed by Sun and is now in its fourth revision. However, despite the third revision's release almost twenty years ago[85, 86], it is still in wide deployment. The "Linux Cluster" filesystem, Lustre[87], is a newer filesystem that distributes the I/O workload across multiple nodes, and thus has been demonstrated to scale considerably better. Both systems have characteristics that should inform how we develop software to run on such systems. We focus this work on these two filesystems due to their prevalence in high performance computing environments.

Collective I/O (CIO) [88, 89, 90] was introduced as a very versatile concept where the I/O bandwidth is increased by coalescing a number of I/O requests to be sent to the storage system as a single large request. Memik et al. [91] extended CIO as Multi-Collective I/O (MCIO) by optimizing I/O accesses to multiple arrays simultaneously. They show that optimal MCIO patterns require the solution to an NP-complete problem but are able to demonstrate up to 85% speedups over CIO using a heuristic approach.

A similar concept was recently presented by Kendall et al. [92]. They showed that, with a carefully chosen greedy algorithm, end-to-end access times of under a minute are possible in the visualization of terascale data. Their system accessed multi-file netCDF [93] data using the Parallel netCDF library [94] that is in turn built on top of MPI-2 [95].

Lofstead et al. [96, 97, 98] report that on current supercomputers, independent I/O tends to outperform collective I/O. They present the ADaptable IO System (ADIOS) and — in combination with MPI-IO and collective MPI-IO — report speedups of about an order of magnitude compared to a serial HDF5 access. To improve access to data stored in HDF5 Howison et al. [99] present optimizations for the Lustre File System.

Specifically targeting scientific visualization of large-scale earthquake simulations on parallel systems, Ma et al. [100] demonstrated that overlapping I/O with

rendering can significantly reduce inter frame delay. This concept was extended into a general parallel visualization pipeline for large earthquake simulations by Yu et al. [101].

Yu et al. [102] conducted an extensive characterization, tuning, and optimization of parallel I/O on Jaguar, a Cray XT based supercomputer at Oak Ridge National Laboratory that uses Lustre [87] for its IO subsystem.

Yu et al. [103] demonstrated general I/O solutions for the visualization of time-varying volume data in a parallel and distributed computing environment.

Peterka et al. [77] also present optimization strategies for the problem of volume rendering large time dependent datasets, focused specifically on the IBM Blue Gene/P system system. Their summary result is that even with optimized storage and access systems I/O still severely limits the overall performance and more research is required in this area.

Recently, Lang et al. [104] performed a comprehensive study of I/O on Intrepid, the IBM Blue Gene/P system at the Argonne Leadership Computing Facility. In their work they also give a broad overview of existing parallel file system evaluations and HPC system scaling studies.

Ching et al. contribute a a more modern take on file and range locking in distributed filesystems [105]. Using their distributed lock manager, they demonstrate scalability up to 32 servers, something the POSIX locking model cannot provide.

5.1.2 Contribution

Our primary goal with this work is to inform developers writing visualization and analysis applications on the characteristics of I/O systems at a multitude of scales. We desire to show methods by which parallel applications can be written to maximize performance for developers' constituency, without working directly with their user base or clusters that the application will run on. As a community, we will never have the resources required to address the specific machines that every supercomputing-based science group needs to utilize. Therefore we must design applications that perform well on such machines without investing weeks (or months) of a visualization or I/O expert's time to achieve that performance.

Most I/O studies focus on a particular machine and even a specific application on that machine. This approach would not, however, serve our purpose of identifying I/O best practices that are widely applicable. We contribute end-to-end scalability results of a typical analysis problem on volume data, for numerous clusters and a variety of I/O backends.

Finally, based on our work developing parallel visualization and analysis applications like the one in this work, we propose an extension to the ubiquitous POSIX API that has the potential to greatly improve the performance of parallel I/O systems.

The remainder of this paper is organized as follows. First, we review some disk and I/O characteristics that are common to both serial and parallel environments. In Section 5.3 we describe filesystems in common use in modern cluster computing environments. Then we expound the design of a program that has I/O as a major component, and describe implementations using numerous backend APIs, in Section 5.4. In Section 5.5 we use the knowledge gained in Sections 5.3 and 5.4 to enumerate an API that would allow improved scalability on current and future parallel filesystems. Finally, we conclude by highlighting the limitations, drawbacks, and opportunities for mistaken conclusions that arise due to our methods.

5.2 Data Access Time

The overall time to perform any I/O operation is well studied. Generally we consider this to follow the simple equation:

$$T_{total} = T_{access} + T_{trans}$$

that is, the total time to perform an I/O operation is equal to the time to seek to the desired track along with the time for the start of the needed sector to spin under the disk head, plus the time for the platter to spin until all the required sectors have passed under the head.

We will utilize a hypothetical modern disk with an average access time of 8 msec, and a sustained transfer rate of 100 MB/s. The access time time is a conservative median for current consumer level disk drives. The 100 MB/s sustained transfer rates are not yet possible with current consumer level disks, but the number is close enough and serves our purpose well.

5.2.1 Considerations for access time optimizations

The total time to transfer data of M MB can be described by the equation:

$$T_{total} = \frac{T_{\text{access}}}{1000} + \frac{M}{R_{\text{trans}}}$$

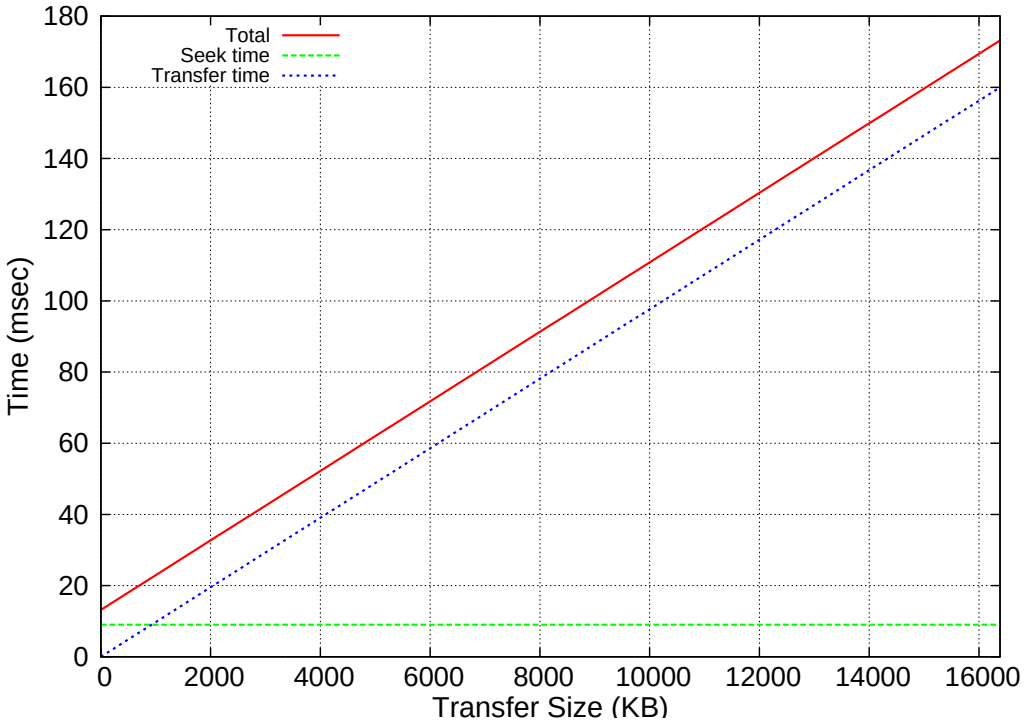

Figure 5.1: Total I/O time as a function of transfer size. Transfer rate quickly overtakes access time.

where R_{trans} denotes the transfer rate in MB per second. We divide T_{access} by 1000 to express it in seconds as it is normally given as milliseconds. Consequently, the access time represents a percentage P of the overall time that can be expressed as:

$$\begin{aligned} P &= 100 \cdot \frac{T_{\text{access}}}{T_{\text{total}} \cdot 1000} \\ &= \frac{T_{\text{access}}}{\frac{T_{\text{access}}}{100} + 10 \cdot \frac{M}{R_{\text{trans}}}} \end{aligned}$$

If we now insert the parameters of the above described hypothetical disk and assume that we are partitioning our data in 10 MB blocks we arrive at the consclusion that—when we need to do a random seek operation for every single block of data—the seek time accounts for only 7% of the overall time to access the data.

Now, to assess the gain of a specific layout scheme we consider the following equation. It measures the performance gain G in percent for a given scheme if that scheme reduces random access by a factor of F.

$$\begin{aligned} G &= 100 \cdot F \cdot \left(\frac{\frac{T_{\text{access}}}{1000} + \frac{M}{R_{\text{trans}}}}{\frac{M}{R_{\text{trans}}}} - 1 \right) \\ &= 100 \cdot F \cdot \frac{\left(\frac{T_{\text{access}}}{1000} \right)}{\left(\frac{M}{R_{\text{trans}}} \right)} \\ &= \frac{F}{M} \cdot \frac{T_{\text{access}} \cdot R_{\text{trans}}}{10} \end{aligned}$$

Again, assuming the hypothetical drive parameters from above and we get

$$G = \frac{F}{M} \cdot 80$$

For a scheme that reduces random access by a factor of 3, only a 2.6% improvement in runtime would be achieved for 10 MB blocks, while with the same scheme the performance would be almost tripled with 900 byte blocks.

From these numbers we conclude that in most environments, in particular those with structured data—where larger data blocks can be utilized more easily—a data layout optimization would only improve a very small fraction of the overall time and is most likely not worth the implementation and maintenance effort. For

environments that *must* break the data into tiny chunks a clever layout strategy to improve data access times can in the best case (in which the majority of disk operations involve seeks) double the data access performance.

It is worth noting that with the advent of solid state drives, in particular for consumer workstations, this minimal block size required to utilize unwrought data layout strategies while still obtaining good performance is bound to shrink even more, as those drives have a significantly smaller 'seek' time with only moderately higher transfer rates.

Finally, it should once again be stressed that the percentages given above account for maximum theoretically possible optimization potentials if all seek operations could be completely avoided and no other additional overhead would come from the layout. In reality the speedup that can be gained from access time optimization will stay below that value. In particular it is worth noting that accessing data via optimized layout schemes does not come for free. Kendall et al. [92] demonstrated for distributed memory systems that a random ordering scheme outperforms most space filling curve approaches.

The takeway:

- If the data is broken into pieces larger than 10 MB, then it is not worth worrying about the data layout for even consumer level disks.
- For kilobyte sized chunks a clever layout strategy can significantly cut the data access time on a standard HDD.

5.3 Parallel Filesystems

All distributed filesystems have unique characteristics that should inform the way we access and process data. In this section we will highlight some of the common pitfalls that may be found with applications designed to run in an NFS or Lustre environment.

5.3.1 Opening Files

Opening a file is one example of an operation that performs uniquely in a distributed environment. In NFS systems, this is implemented via the client sending an `ACCESS` or `GETATTR` remote procedure call. The operation asks the server if the client is allowed to access the file, or requests metadata for the file. The server responds with a small message containing the resulting permissions. The situation

in Lustre is similar: queries go to a global 'metadata server' (MDS) that grants or denies access. In both systems, *the file is not opened*. Doing so would consume resources on the server, particularly due to read-ahead caching, and the request to actually read or write the file may be significantly delayed in time—or might never come at all!

This has important implications for programs running on such filesystems. Any distributed filesystem is going to scale extremely poorly with a program that opens many files at one time. Since the `open` call must correctly report errors, the request and response must be entirely synchronous. There is no `openv` system call in POSIX, analogous to `readv`. Therefore every open file request must send a (very small) message to a server, and wait for a (very small) message to return. The network capacity for messages at these sizes is extremely poor. It is important to note that *Lustre does not scale any better than NFS* in this use case, as it has the singular bottleneck of one MDS per filesystem. Many sites split up their Lustre offerings into multiple filesystems as a way to mitigate this problem, but these must then be mounted under different locations in the filesystem hierarchy.

To prevent inducing poor performance in this manner, avoid opening more than one or two files per process; at large scale, even that will be a bottleneck. Furthermore, if at all possible, avoid synchronization points immediately before opening files: if one absolutely needs an `MPI_Reduce`, for example, try opening the file immediately before the reduce instead of immediately after. This should prevent a 'thundering herd' (to steal a term from the threading world) of processes that pound on the metadata server at the same time. It is interesting to note that the ADIOS middleware library already attempts to mitigate this effect [98].

The takeway:

- At large scale, eschew large numbers of files.
- Stagger synchronization points with `open` calls.

5.3.2 Closing Files

Distributed filesystems almost unilaterally implement what is referred to as 'close-to-open cache consistency'. To increase performance, writes are cached locally on the client filesystems. During regular intervals or in response to certain events, the client cache is flushed to the server.

This presents difficulties in implementing `writes`. The problem is in reporting errors when a write should fail; since the system only writes to a local cache, the

write never reaches its final destination and thus additional errors could still occur after the user process has proceeded beyond the write. It is possible for the write to be sent to the server machine, enter into the server's cache, and eventually be denied due to a transient error (e.g. exceeding quota). Yet the client system cannot report this error to the running process, because the process has long since moved on from the failing write call.

Distributed filesystems thus require a client cache to write-through all changes when the client application *closes* the file. Client operating systems must get a confirmation from the server that all data has been flushed *before* it returns from the client processes' `close` call; this is the last possible operation for the file, and thus the distributed systems' final opportunity to report errors that may indicate data loss.

It is therefore highly desirable to delay close operations that occur after writes. If a process is writing multiple output files, try to make it maintain two open files instead of one, and close the file from the previous iteration while writing in the current iteration.

Sadly many applications, even those designed to run on supercomputers, do not check the return value of the `close` system call. There is no reason to believe that what was written is consistent, given such applications.

The takeaway:

- *Always* check `close` for errors!
- Try to delay `close`s that appear after `write`s.

5.3.3 Locking

By 'locking' here we are referring to advisory file locking, a la the `flock` system call; mandatory file locking has its own set of issues in even a serial environment, and the utility of such locking in an HPC environment is nebulous. In our experience, few if any large scale visualization and analysis applications utilize file locking. However, it is worth noting that locking typically adds an I/O synchronization point, much like `close` would. For this reason it is not recommended that an application lock and unlock files unless there is an interaction with known external software that dictates it. If at all possible, a better solution would be to `close` the files on the writing process, and send a message to reading processes notifying them that the writer has completed—before they attempt `open`ing the files at all.

Locking can in theory provide the best mechanism for inter-process communication in a distributed environment (i.e. to coordinate with in situ visualization and analysis processes), however it is not in wide use, perhaps due to the issues mentioned here. As noted earlier [105], this is still an area of research in HPC systems and so we recommend the aforementioned explicit synchronization methods for now.

5.4 Parallel Data Access

To identify the ideal method for accessing data in numerous environments, we wrote test programs using a variety of APIs and API options, then evaluated their performance. Yet many scientific visualization and analysis packages, in addition to large scale simulation software, utilizes some I/O middleware for data access. These middleware packages offer complexity reduction, and typically provide a method for ascribing higher level metadata with data, such as the dimensionality and mesh information. After identifying the ideal low-level methodologies, we sought to quantify the differences between middleware libraries, and in particular their scalability on distinct clusters.

To quantify this, we developed the same analysis program using a variety of backend APIs. The program is simple: it is a threshold-based volume segmentation tool. The software reads in a large volume and outputs a binary mask volume that indicates the voxels that fall between the threshold values. The program is parallel, and out-of-core: the input volume is intelligently bricked, and each process is responsible for a set of bricks. Processes load up a brick and generate the output volume brick one at a time. We chose out-of-core as opposed to in-core because it models how future (even current) visualization and analysis software must be written, given the current trend of increasing processing-power-to-memory ratios.

5.4.1 Results

We ran our application on multiple distinct supercomputers. One cluster is specifically designed for visualization; another excelled at analysis; the third is a very large scale general purpose supercomputer designed for ‘leadership computing’. Installation dates were diverse: one cluster was commissioned in 2008, another went into production early in 2010, and a third was originally installed in 2005, receiving its most recent upgrade in 2009. All of these clusters are using Lustre

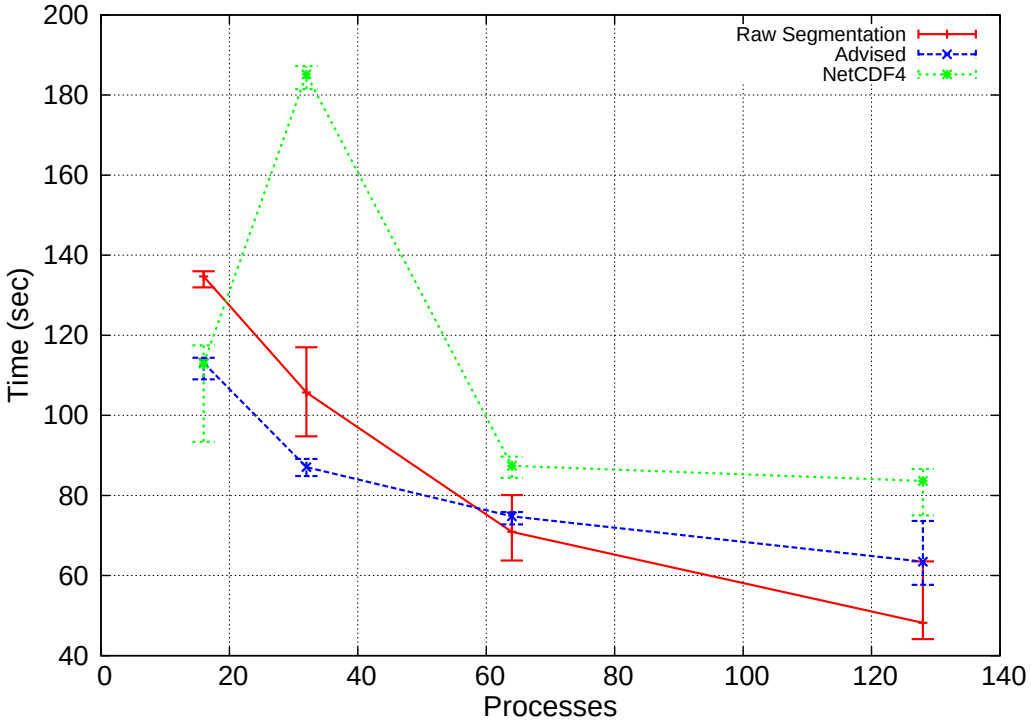

Figure 5.2: Strong scaling of our example segmentation program running on cluster #1, with a variety of I/O backends. 'NetCDF4' is NetCDF with an HDF5 backend. 'Raw' is our hand-generated simple I/O layer, and 'Advised' a minor modification on it. Error bars indicate maximum and minimum running times per process in the job.

for their backend filesystem. On all systems, we used the 'native' compilers and, where available, system-installed modules for the libraries we required.

For backend I/O we tested multiple configurations: NetCDF, HDF5/NetCDF, and a custom solution.

The hierarchical data format (HDF) is a data model that has seen significant uptake in the parallel computing world. It provides mechanisms for organizing complex data in an extensible manner. We did not look directly at HDF5, but instead considered it in concert with NetCDF.

The Network Common Data Form is a library that provides array-oriented data access. Like HDF5, NetCDF files endeavor to be partly self-describing. With recent releases of the NetCDF library, there are a multitude of options for backend I/O. The first is so-called 'classic' NetCDF files. These files have a limit of 2Gb

per variable, and thus were not considered for this study. The '64bit offset' format is an extension of the 'classic' format to allow use of 64bit indices, and thereby to address files of, for all practical purposes, unlimited size. The final format is the so-called 'NetCDF4' format – somewhat confusing because the '64bit offset' format debuted in 3.6.0, right before the 4.0 release, yet is a distinct backend that uses HDF5 as its backend. To disambiguate, we refer to the '64bit offset' format as "NetCDF-64" and the HDF5-backed format as "'NetCDF4" in this work.

We also developed a custom I/O layer based on our experiences on a variety of machines, including workstations. The approach is very simple: each process memory-maps a chunk of the large input data file, as well as the relevant portion of the output mask file. Data are processed out of the memory-map as is, without intermediate buffers. The source for this version is thus simpler than any other version of the program, containing no memory management code for data buffers. As such, this version required the least memory by a wide margin: the API dictated an approach that was naturally out-of-core.

The results on the first cluster can be seen in Figure 5.2. 'NetCDF4' is NetCDF backed with an HDF5 file. 'Raw segmentation' uses our custom I/O layer based on `mmap`. 'Advised' is the 'Raw' line, with the addition of just a single line of code, placed before we process a block of data:

```
posix_fadvise(fd,
  index * sizeof(float),
  buffer_size,
  POSIX_FADV_WILLNEED
);
```

That is, we are informing the operating system that we will need block $X+1$ in the near future, just before we begin processing block X. We had found that including this optimization increases our performance 3 to 4x on desktop systems. Results on the supercomputer do show an initial increase in performance, but the effect was unfortunately subdued at higher concurrency. We do not include results for the NetCDF-64 run in this figure, as it did not fit in the same scale as the pictured backends.

Results for this machine were somewhat difficult to report, because they varied so widely. We ran the scaling study for one particular format straight through, with no delays between runs, multiple times. In each instance the NetCDF4 result included a spike in the running time. For our raw segmentation, we would see results offset by 20 seconds or so, and the width of the error bars would change arbitrarily. The readahead version of the program experienced less variability, but

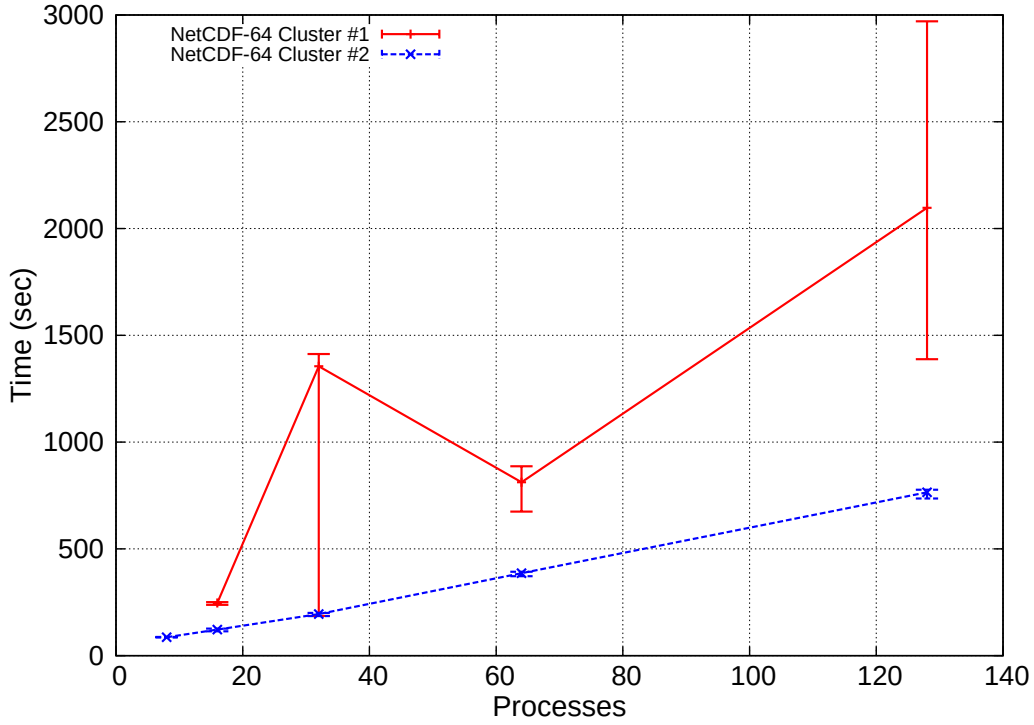

Figure 5.3: Strong scaling using the NetCDF '64bit offset' file format on multiple clusters. Higher levels of concurrency led to decreased overall performance when using this format. Of note is the high variability from cluster #1, characteristic of that machine's I/O subsystem.

we were unable to conclude whether this was a property of the program or simply luck. The data presented in figures represents the *set* of runs that performed best overall, for that I/O backend.

The NetCDF-64 results could not be plotted with the other results, due to the large difference in scale. Results using this format on multiple clusters is provided in Figure 5.3. Performance actually decreased with this backend. For this reason, we highly recommend forcing the HDF backend (using the `NC_NETCDF4` flag) when writing applications that make use of the NetCDF API.

Results from running on the second cluster are given in Figure 5.4. The HDF-backed NetCDF version could not be run on this cluster due to a software incompatibility.

Results from the third cluster are given in Figure 5.5. This machine is one of the largest scale supercomputers we have access to, and so we performed runs at

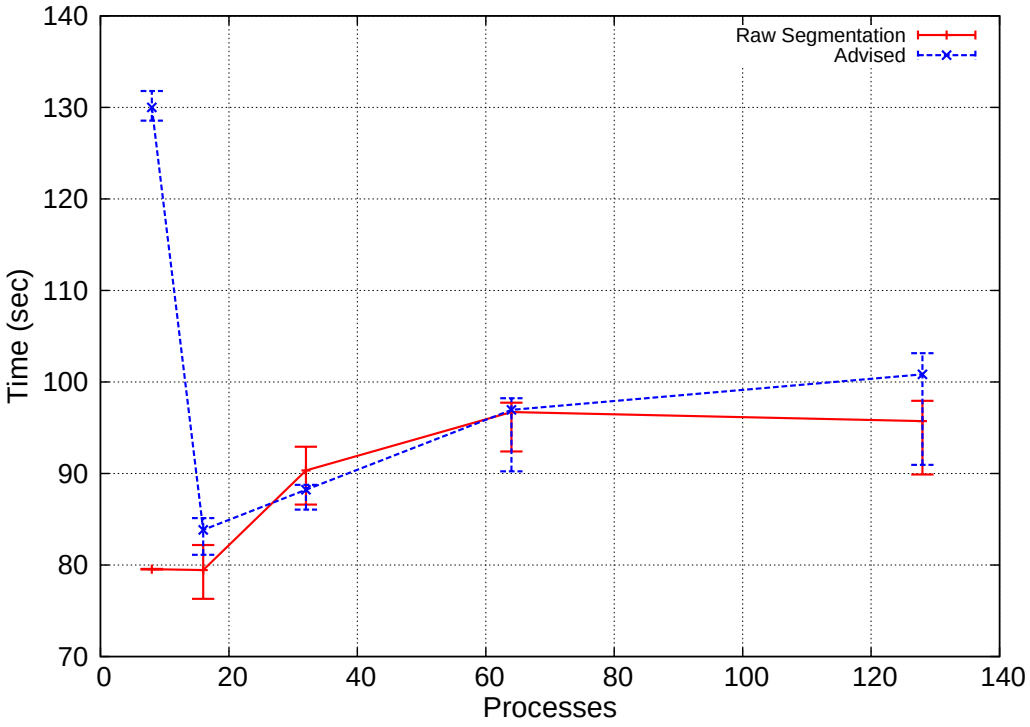

Figure 5.4: Raw segmentation strong scaling with and without explicit caching, on the second cluster. Explicit single-block readahead makes little difference, especially at higher concurrency levels.

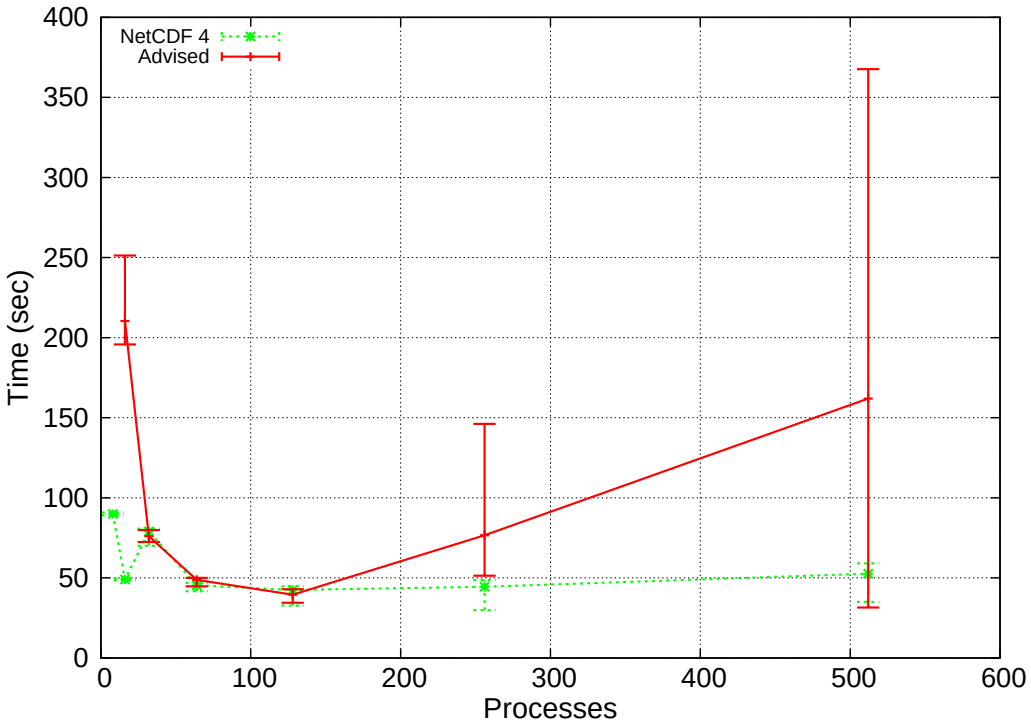

Figure 5.5: HDF-backed NetCDF and raw advised I/O method scaling on the third cluster. Performance is largely the same at first, hinting that explicit readahead is likely too limited to be effective. At higher levels of concurrency, our writes get very small, and the HDF backend is able to deal with the case much more effectively.

larger levels of concurrency, although we did not utilize the entire cluster. We only performed the 'advised' version of our raw algorithm for this cluster, as the simpler version gave essentially the same performance, and compute time was harder to obtain for this machine.

5.5 A Parallel File API

In the work presented for this paper, as well as our previous experience writing visualization and analysis programs targeting supercomputers, we have noticed that the available I/O models presented by the standard POSIX API is insufficient from both the producer and consumer vantage points. Implementers are not given enough information about access patterns that applications are utilizing, which prevents filesystems from optimizing for common tasks. Library and application developers, on the other hand, have no mechanism for communicating such information. The result is sub-par performance, with both parties feeling like there is little that can be done.

For this reason, we present an API that:

- models the way visualization and analysis application programmers think about their data,
- simplifies data access, and
- enables implementers to design effective filesystems.

A summary of the methods that the API provides is given in Table 5.1. Our primary goal with such an API is to encourage application developers to structure their code in such a way that it models a similar API, as it provides more information to lower levels. At the same time, we hope middleware libraries and even kernel code begin to offer APIs that allow application developers to provide this kind of information. In the end, there should be much more data about the patterns and intent of data access that the application provides to the levels that can make use of it.

`open_range` and `close_range`; `open` and `close` calls that work with byte ranges. One of the issues that plagues I/O concurrency at large scales is the inability to indicate that portion of the data a process intends to access: each process only needs some subset of the overall data, but cannot communicate this a priori to middleware or the runtime system. To perform effectively in the majority

System call	Description
open_range	open with an explicit range of accessible bytes.
close_range	clean up resources associated with a buffer
readanyv	accept a set of blocks and returns when any one full block is available
finished	asynchronous flush; return immediately, but mark buffers as unused.

Table 5.1: Summary of proposed new APIs.

of cases, caching, large stripe sizes, and readahead must be employed by the I/O system. However these techniques create false sharing when byte ranges overlap.

A popular method to combat this problem is to create a single file per process. Since only one process accesses the file, it is clear to the I/O subsystem that concurrent access is impossible. While this is effective at the small scale, at the highest levels of concurrency the method becomes untenable due to overwhelming amounts of metadata: listing all files in a directory would require making a hundred thousand requests to a server. Even if this were technically feasible, it presents significant data management difficulties; it would be much easier on users if we could contain results into a singular file.

Many analysis applications would be able to calculate the byte offset they will need on a given process given just the total number of processes and the dimensions of the dataset (or number of points in a point mesh). Visualization software may need to produce a spatial hierarchy of the data, but again this can be done with relatively little metadata. By providing this information to the underlying I/O subsystem up front, application developers can cleanly solve one of the more difficult problems in defining distributed file systems: distributed lock management.

It should be an error to specify a byte range beyond the file length when opening a file for read-only access. When used for write access, this would be a viable method for extending the file's length. All offsets from a file opened in this manner are relative to the start of the byte range. Attempting to read beyond the end of the byte range results in end-of-file.

If the API is made to work with existing file descriptors, the standard `close` call is the only API needed. If this API returns a more opaque type, an API-specific `close` method will be required.

`readanyv`; a read that accepts a number of blocks and returns one of them. Many applications can identify what data it will need using a small amount of metadata. For the segmentation application used in this work, for example, we could compute that easily based only on the total amount of data and the number of

processes in the analysis job. A volume renderer could read just the world extents of each block and use that for a spatial subdivision. In short, it is common for an application to be able to make progress given some small subset of its input, as long as each subset is 'complete' in some sense. This interface allows an API implementer to do *intelligent* read-ahead; as demonstrated in our test program, this can provide compelling performance advantages.

The API should return pointers; it should not accept previously allocated buffers. The gives the implementer freedom to manage allocations, enabling flexibility in choices of underlying APIs. For example, memory-mapped files require page-aligned memory that is not provided by `malloc` or `new`, and is more difficult to use at fixed addresses, as opposed to letting the kernel choose the mapping.

`finished`; an asynchronous flush operation. This indicates that the given file (or byte range within the file, given `open_range`) will no longer be used. The method returns prior to performing any I/O operations. It is an error to read from or write to the given file after performing this operation. It is an error to open the given file within the same process without an intermediate `close` operation. An implementation may detect these errors. It is unspecified whether any other process sees any modifications to the open file before a future `close` operation completes.

The intent is to allow a system to better manage its cache and write throughput. Should the system experience memory pressure, these cache blocks are the best candidates to consider for flushing. If the network or host resources are currently busy, the system might delay making the write request until a better time. This would also allow an implementation to avoid a 'thundering herd' of disk write requests: mitigated in a system such as Lustre, but a difficult problem in an NFS-like environment.

It is important to note that, while this system interface was explicitly developed to deal with the problems of distributed systems, most calls could provide benefits for applications targeted to typical workstations. The issues are largely the same, though the stakes are higher in a distributed system. Furthermore, such a system would not obviate the need for current infrastructure; not all file access can be made to conform to this model, but the intent is that large scale applications would be able to effectively utilize these APIs for their primary I/O needs.

5.6 Conclusions

We have presented performance characteristics of modern disks. Utilizing that information, we evaluated a variety of APIs for file access with large scale data by

implementing the same program using multiple backends. Where APIs had options that may effect performance, we experimented with those options to identify the set that gave the best parallel performance on our chosen problem. We evaluated this program on multiple clusters, attempting to identify generalized practices that application developers should follow to obtain superior performance in the common case: where they have no control over where their users will run the released code.

Variability in I/O performance, such as that depicted in Figure 5.3, was considerably higher than we expected it to be. In some cases we observed a job taking twice as long to execute than it did at another point in time. This presents a difficult challenge for interactive visualization and analysis applications that should provide the illusion of interactive response yet are highly susceptible to such latency. The results encourage the use of progressive or multiresolution renderers that can be used to provide real-time responses in the plausible event that the supercomputer cannot respond quickly enough.

While the best performance was generally obtained by using operating system APIs directly, we do not advocate developers use these directly at this time. Higher level libraries such as NetCDF, HDF5, and ADIOS provide mechanisms for self-describing metadata and data attributes, and can achieve similar performance with the proper configuration, not to mention providing portability across a wider set of systems. Instead of having every application developer familiarize themselves with these to-the-metal APIs, our community should instead work towards the goal of incorporating these ideas into higher level libraries. However, some API changes, preferably to accomodate a model more like the one presented in Section 5.5, could go a long way towards getting users to write code that can be scaled much more easily.

For application developers, we present the following maxims for obtaining the best I/O performance possible:

- Stagger operations that read or write file metadata.
- Read or write in large chunks: 10 megabytes or more.
  - This frees the developer from the requirement of identifying and implementing intelligent data layout schemes.
- Use memory-mapped files whenever possible.
- If you can do more, unrelated work before `close`-ing some file resource, do so.

- *Always* check and report errors during `close`.

5.6.1 Limitations

Any study is subject to the limitations of that which can be tested, as well as the time available to perform tests *ad nauseum*. This study is no different, and suffers from at least the following barriers and limitations on its conclusions.

The most serious is our chosen test application. We have chosen to implement a program that essentially maintains two small buffers at any one time: a brick of the input file and an output brick. In a real-world application, it would desirable to load as much data as would fit in the current memory. Furthermore, many current applications are not intelligent enough to implement either method: they employ strictly in-core algorithms. Due to the memory struggle between application heap allocations and the operating system's filesystem caching, in-core applications clearly perform worse when the heap memory required grows close to the available memory on a node. Finally, our application performs very little work on each input voxel; this was done to emphasize I/O time, but is uncharacteristic of any useful analysis program. Therefore it is likely that the application presented here performs better than real-world visualization and analysis applications.

A second issue, particularly with respect to the proposed API, is the lack of thorough evaluation. No applications have been written to such an API. We have implemented the API in user-space, but no middleware or applications have as-yet been adapted to utilize the model it presents. Despite these shortcomings, we feel the approach is well-informed based on our experiences here and in prior literature.

5.7 Future Work

The ADIOS library is unique in that it is not a file format alone, but rather a middleware suite that interfaces to a variety of backend methods for reading and writing data. These methods include HDF5, NetCDF4 and ADIOS-only backends such as raw POSIX I/O and MPI-IO. Unfortunately the current release at the time of publication (ADIOS 1.2.1) does not yet support out-of-core data access. For large scale visualization and analysis applications that commonly run on just a subset of the nodes utilized to produce simulation data in the first place, we judged this to be an essential feature. We contacted the development team and they agreed to look into out-of-core APIs for a future release; we therefore hope to include ADIOS results in a future study.

We would like to extend the methodologies used in this work to a larger set of parallel algorithms. In particular, algorithms that must do considerably more per-voxel computation, and those that require information from neighboring voxels.

Chapter 6
Freeprocessing

In situ visualization has become a popular method for avoiding the slowest component of many visualization pipelines: reading data from disk. Most previous *in situ* work has focused on achieving visualization scalability on par with simulation codes, or on the data movement concerns that become prevalent at extreme scales. In this work, we consider *in situ* analysis with respect to ease of use and programmability. We describe an abstraction that opens up new applications for *in situ* visualization, and demonstrate that this abstraction and an expanded set of use cases can be realized without a performance cost.

6.1 Introduction and related work

The growing size of simulation data and the problems this poses for subsequent analysis pipelines has driven simulation authors to integrate visualization and analysis tasks into the simulation itself [106]. The primary advantage of this approach is to perform operations on data while they are still in memory, rather than forcing them through disk, thereby eliminating the most expensive component of the majority of visualization and analysis pipelines.

Scientists and engineers have developed many different approaches to *in situ*. DART uses RDMA to stage data from supercomputer to potentially separate analysis-focused resources [107], and a system performs computations on the data as they are in transit from one resource to another [108]. The dominant approach is to use the same supercomputer that is running the simulation for visualization, though potentially on just a subset of cores, in the manner of Damaris/Viz [109]. Damaris/Viz can provide a wealth of visualization and analysis opportunities due

to its ability to act as a front end to both VisIt's [13] `libsim` [110] as well as ParaView's Catalyst [111, 112]. Biddiscombe et al. proposed an HDF5-based driver that forwards the data from HDF5 calls to ParaView [113]; we give an example of our system implementing similar functionality in § 6.3.2. Abbasi et al. introduce DataStager, a system for streaming data to staging nodes and demonstrate a performance benefit by asynchronously streaming multiple buffers at one time [114]. *In situ* libraries can also be used to improve the performance of simulation code [115].

Most work focuses on extreme-scale performance with less regard for the effort required in integrating simulation and visualization software, whereas we focus on the latter concern. Notably, however, Abbasi et al. extend their previous work with a JIT compiler that allows users to customize data coming through ADIOS [116] using snippets of code written in a subset of C [117]. Zheng et al. modify OpenMP runtimes, an approach that shares our mentality of working within the constraints of existing infrastructure [118]. Others have tightly integrated simulation with visualization to allow steering, but these generally come at high integration costs [119, 120].

Existing solutions leave a potentially large segment of the user community behind. Most previous work has integrated or presupposed integration with particular libraries for performing I/O operations, and no such library has achieved universal adoption. Yu et al. note the tight collaboration required for a fruitful integration [121]. Reasons for not adopting I/O middleware are varied: the difficulty in integrating the library with local tools, perceived lack of benefit, lack of support for existing infrastructure with home-grown formats, or issues conforming to required interfaces, such as synchronous 'open' calls.

Moreover, the focus of modern I/O middleware specifically on simulations at the extreme scale leaves a long tail of potential *in situ* uses behind. The set of simulation authors focused on creating exascale-capable simulations is a small subset of all simulation authors. A large set does not even dream of petascale; and even larger are those who would barely know how to exploit a terascale-capable solver for their science. The distribution gets larger and more diverse as one moves out to lower scalability levels.

At the opposite end of 'extreme scalability' uses for *in situ*, one may find a number of heretofore ignored applications. There is no reason to limit the *in situ* idea to parallel code running on a supercomputer, for example. Analysis routines embedded into the fabric of network transfer operations would be a boon to distributed research groups (and the success of tools such as Globus [122] speaks to the multitudes of domains faced with this problem). Those writing simulations

in MATLAB® might also benefit from precanned visualization tasks that occur concurrently with their simulation, yet the closed source nature of the product makes the prospect of integrating I/O middleware improbable at best.

The currently-dominant middleware approach to *in situ* requires significant effort. It is reasonable for simulation authors to spend a week integrating and retooling their code to achieve thousand-way concurrent *in situ* visualization, but this level of investment is unreasonable to users who simply wants to compute a data range on their files as they move across the country. The cliff between 'nothing' and a '100%' solution for *in situ* visualization with existing middleware solutions is too high to appease such diverse use cases. Worse, the model is unworkable in some situations; it is doubtful that the OpenSSH maintainers would accept patches incorporating ParaView's Catalyst into `sftp`, for example.

*Free*processing is an abstraction of previous work. Using it, one can implement classical *in situ* visualization and analysis, computation or data reduction via staging nodes, unique instrumentation such as gathering power consumption information dynamically [123], or a number of novel 'processing while moving data' ideas. This processing can be synchronous or asynchronous depending on the needs and desires of the user. Developers of a *free*processor can connect it to existing visualization tools such as VisIt's `libsim` or ParaView's Catalyst, implement their own analysis routines, and even push data into another language such as Python, all without data copying—*or* with data copying, should those semantics be preferable. The general nature of *Free*processing not only allows one to implement the diverse domains of previous work, but also allows novel use cases. Specifically, we contribute:

- a new method for inserting data processing code into I/O operations;
- the generalization of *in situ* ideas to heretofore unexplored domains, such as visualization during network transfer;
- greatly increased programmability for *in situ* ideas, making them applicable with considerably less effort;
- a sample implementation that demonstrates all of these ideas in real-world cases.

The rest of this paper is organized as follows. First, we explain the technical underpinnings of how the program works. In § 6.3 we demonstrate *Free*processing in some classical environments and show that there is almost no overhead. We

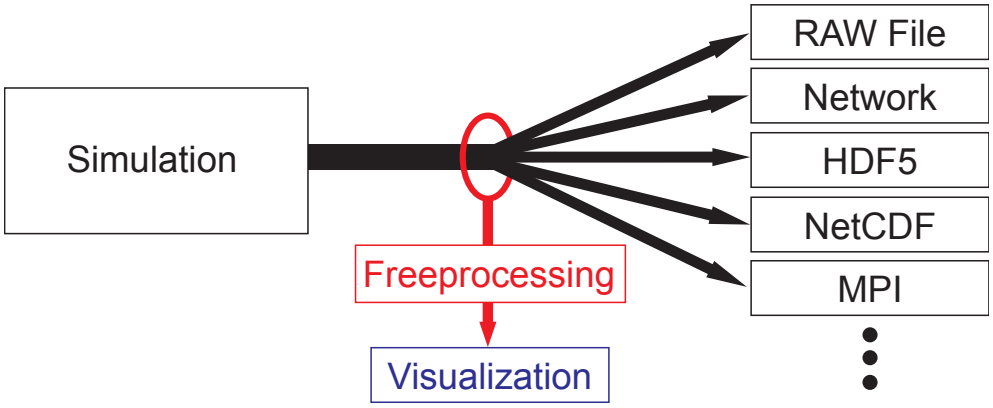

Figure 6.1: *Free*processing works like a vampire tap on the data coming out of a simulation. Without changes to a program's source code, we can intercept the data as it goes to the IO library and inject visualization and analysis tasks.

demonstrate some novel uses before we conclude and note limitations as well as future work in § 6.5.

6.2 Instrumentation

Previous *in situ* solutions have relied on the simulation author explicitly invoking the visualization tool, or the simulation using a custom library for I/O that is then repurposed for analysis. In this work we demonstrate that there is little need for either; every simulation produces output already, an *in situ* tool just needs to tap into that output.

Our symbiont uses binary instrumentation to realize that tap. We take unmodified simulation binaries and imbue them with the ability to perform visualization and analysis tasks. In doing so, we remove a potentially complicated component of *in situ*: modifying the program to work with the visualization or analysis tool. Notably, this approach enables simulation software to produce *in situ* visualizations even when the source code of the simulation is unavailable. Furthermore, as the symbiont interposes these functions during load time, a user need only change the invocation of the program to enable or disable these features.

The method we use is to redefine some of the standard I/O functions, in a similar manner to the way the GLuRay or Chromium systems operate [12, 79]. These methods rely on features available in runtime dynamic linkers to replace any

function implemented within a library at load time. The overridden entry points form what we call the 'symbiont', the core of *Free*processing. The symbiont's purpose is to conditionally forward data to a *free*processor—a loadable module that implements the desired *in situ* computation—in addition to fulfilling the function's original duties. Separating the instrumentation itself and the *free*processor allows users to develop processing elements without knowledge of binary instrumentation.

The set of intercepted functions is different depending on the I/O interface that the simulation uses, as shown in Figure 6.1. For the C language, these functions are those of the POSIX IO layer, such as `open(2)` and `write(2)`. In Fortran these calls are implementation-specific, and C++ implements I/O differently, but on POSIX-compliant systems all such implementations are ultimately layered on top of the POSIX I/O interface. We also introduce interposition for higher-level functions, such as those that comprise MPI File I/O, and a subset of calls from the HDF5 family. Using this interposition, what the simulation believes is a standard 'write' operation actually calls in to our symbiont.

6.2.1 Data semantics

Function interposition for higher-level functions from libraries such as HDF5 and NetCDF provide an important benefit: data semantics. As these formats are self-describing, there is enough information in just the stream of function calls to identify data properties—in contrast to raw POSIX I/O functions that provide little more than an abstract buffer. The symbiont forwards any available data semantics from the interposed library functions to the *free*processor.

However, in contrast to previous work, *Free*processing will also willingly forward data without knowledge of any underlying semantics. A *free*processor can also ignore metadata simply by not implementing the methods that interpret those messages. This distinction is important, as it both enables *Free*processing to function in a larger set of scenarios, as well as increases the flexibility of the system. Presumably a *free*processor would then obtain this information from some external source. We view allowing semantic-less data transfer similar to using 'dangerous' constructs in a programming language, such as casts in C. While these constructs are generally frowned upon, with restrained application they can be a powerful and thereby useful tool.

6.2.2 Data semantics

Meta-information concerning data semantics are required, and are only available through *Free*processing in limited cases. While we consider such concerns beyond the scope of this work, they need to be provided for the demonstration of the technique. The general nature of *Free*processing allows any number of solutions: the problem is no different than understanding arbitrary binary data read from a file. One of the solutions we have found works well is a simple text file in the style of Damaris/Viz or ADIOS [109, 116]. An example of one such configuration is given in Listing 6.1. However, it is important to note that this configuration is external to *Free*processing itself. The symbiont does not contain this parsing and metadata acquisition code; the 'user code'—*free*processors—implements this only if they desire.

Listing 6.1: JSON configuration file used for a Silo conversion *free*processor. Variants that do not require the repeated `"i"`s are possible, but lack the desirable property of strict adherence to the JSON specification.

```
{ "dims": [ {"x":4}, {"y":2}, {"z":3} ],
"coords" : [
 { "x": [ {"i":0.0}, {"i":1.0}, {"i":2.0},
          {"i":3.0} ] },
 { "y": [ {"i":0.0}, {"i":4.5} ] },
 { "z": [ {"i":0.0}, {"i":5.0},
          {"i":10.0} ] } ],
"type": "uint8" }
```

*Free*processing itself does not endorse any specific method for obtaining data semantics, in the same way that the C file I/O routines do not endorse a specific encoding for metadata on binary streams.

6.2.3 Defining *free*processors

The module interface for a *free*processor is simple. The system exposes a stream processing model. Data are input to the processor, utilized (or ignored), and thereafter unavailable. This interface is in principle the same model as GLSL, OpenCL, and CUDA expose, though we do not currently impose the same restrictions. A *free*processor is free to implement a cache and process data in a more traditional manner, for example.

Listing 6.2 shows the *free*processor interface. The symbiont calls `Init` when

a file is first accessed; some of our *free*processors initialize internal resources here. The `filename` parameter allows the processor to provide different behavior should the simulation output multiple file formats. The `buffer` and `n` parameters are the data and its size in bytes. If the required information is available, the symbiont will call `Metadata` immediately before a write, communicating the characteristics for the impending data. Likewise, `finish` cleans up any per-file resources. Finally, the `create` function implements a 'virtual constructor' to create the processor. All functions sans `create` are optional; if a *free*processor has no need for metadata, for example, it simply does not implement the corresponding function.

Listing 6.2: Base class for a *free*processor.

```
class Freeprocessor {
  virtual void Init(const std::string&);
  virtual ~Freeprocessor();

  enum DType { FP_FLOAT, FP_INT8, ... };
  virtual void Metadata(const size_t[3],
                        enum DType);
  virtual void Stream(const void* buffer,
                      size_t n);
};
extern "C" Freeprocessor* create();
```

Configuration

The symbiont reads a configuration file that describes which *free*processor to execute. Any library that satisfies the interface given in Table 6.2 is a valid *free*processor. It is important to note that the operations share the semantics of the simulation code. For example, if a parallel simulation performs only collective writes for a given file, then it is appropriate to perform collective operations in the *free*processor's `Stream` call.

It is common for a simulation to produce a large set of output files. Furthermore, MPI runtimes frequently open a number of files to configure their environment, and all these files are 'seen' by the symbiont. It is therefore necessary to provide a number of filtering options. Some of these are built in, such as ignoring files that are opened for read-only access. Others the user specifies in the configuration file for the symbiont. The specification uses a match expression for the filenames,

so the user can further limit where instrumentation will occur. These match expressions provide a more convenient mechanism to uniquely connect processing elements to streams, but the assignment could also be done by the *free*processor implementation.

Python

Developers may also implement *free*processors in Python. We provide a simple *free*processor that embeds the Python interpreter and exports data and needed metadata. Most notably, it creates the '`stream`' variable: a NumPy array for the data currently being written. Exposing the array to Python does not require a copy; the simulation data shares the memory with the Python runtime. Should the Python script attempt any write operation on the data, a copy is transparently made inside the Python runtime that is then managed via Python's garbage collector. We allow only one of the simulation or the Python tool to run at any given time.

The Python script is otherwise indistinguishable from standard Python code; the symbiont imposes no restrictions beyond the unique source of data. Communication via, e.g., MPI4Py is even possible, provided the simulation utilizes synchronous writes. In § 6.3.2 we demonstrate this method by connecting *Free*processing with the `yt` visualization tool [124].

6.3 Classical in situ

*Free*processing can implement a number of *in situ* ideas, including the traditional use case of *in situ*: visualization and analysis during a simulation run. In this section, we detail how the corresponding *free*processors for a few simulation codes operate, and demonstrate that the overhead of the method is negligible.

6.3.1 PsiPhi

PsiPhi is a Fortan95/2003-based CFD-solver that focuses on Large Eddy Simulation (LES) of flows that include combustion and other types of chemical reactions. The simulation discretizes the governing equations of mass, momentum, and species concentration on a cartesian grid via the finite volume method. Second-order schemes discretize the domain, and an explicit third-order low storage Runge-Kutta scheme advances the solution. The immersed boundary (IB) technique handles diverse geometries in a computationally efficient manner. Besides the solution

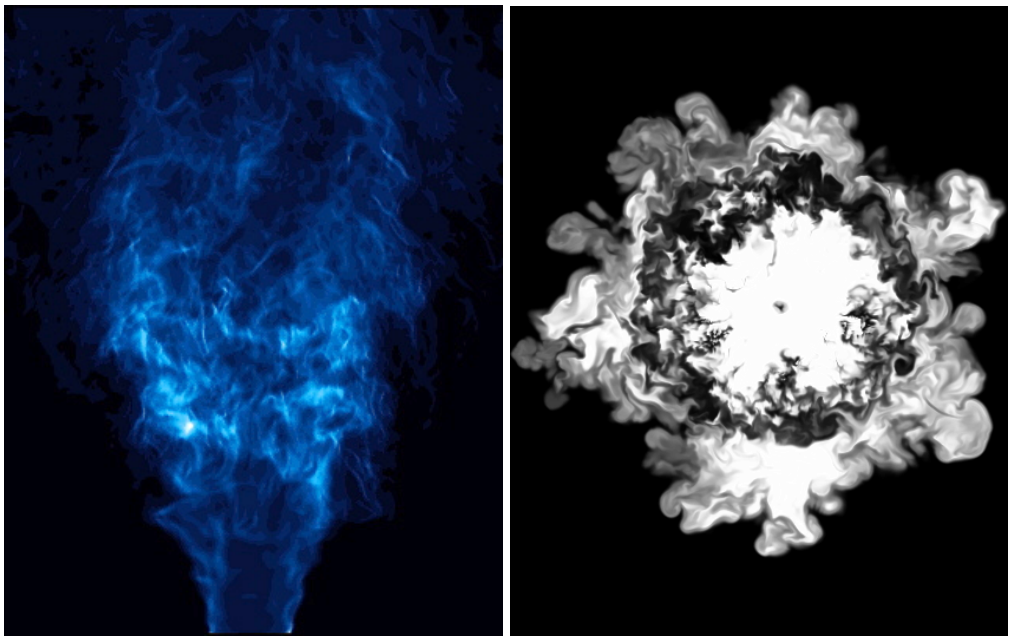

Figure 6.2: Sample *in situ* visualizations of the Cambridge stratified flame produced by the PsiPhi code.

of the mentioned transport equations in an Eulerian formulation, the code is able to solve the equations of motion for Lagrangian particles. A combination of Lagrangian particles and immersed boundaries describes moving objects. The code is modular, easy to extend and maintain, and highly portable to different machines. PsiPhi parallelizes via the distributed-memory paradigm, using MPI.

PsiPhi simulates highly-resolved simulations of reactive flows, e.g., premixed, non-premixed and stratified combustion, coal and biomass combustion, liquid spray combustion, and nanoparticle synthesis [125, 126, 127]. The software has scaled to thousands of cores on Top500 machines such as SuperMUC and JUQUEEN. Recent tests with the program have shown that the output of the computational results becomes a performance bottleneck when moving up to an even higher number of cores.

There are three types of intermediate outputs in the PsiPhi simulation. The first are actually custom-developed *in situ* visualizations: slice outputs and volume renderings. The simulation writes out these visualizations in custom ASCII-based formats every n time steps, with typical values of n in between 100 and 1000 [128]; Figure 6.2 shows example visualizations. The second type of output is a simulation-specific binary format used for restart files that is organized in a 'one file per process' manner. Synchronous Fortran 'unformatted' `WRITE` operations create these outputs. The third kind of output is an ASCII-based metadata file that describes the layout of the binary restart files.

The PsiPhi authors are interested in extracting arbitrary 2D slices as well as 3D visualizations with more flexibility than their custom-developed routines allow. Therefore, we developed a custom *free*processor for the PsiPhi simulation. PsiPhi periodically dumps its state to disk in the form of restart files, at approximately the same cadence as 'normal' output files. We utilized the aforementioned restart files as the basis for our *free*processor, in addition to parsing the ASCII-based metadata to interpret these restart files.

The simulation authors were enthusiastic about the *free*processor. All the outputs the simulation previously created were redundant with the restart files. Furthermore, PsiPhi users hardcoded postprocessing parameters such as slice numbers into the simulation source, necessitating a recompile to modify the parameters. In light of the visualization options presented by the *free*processor, the PsiPhi authors elected to remove all custom-developed *in situ* outputs and create only the restart files.

We therefore reimplemented their outputs in a *free*processor and measured the performance of the system under both the old and new configurations. As shown in Figure 6.3, not only was the overhead miniscule, but the simulation actually ran

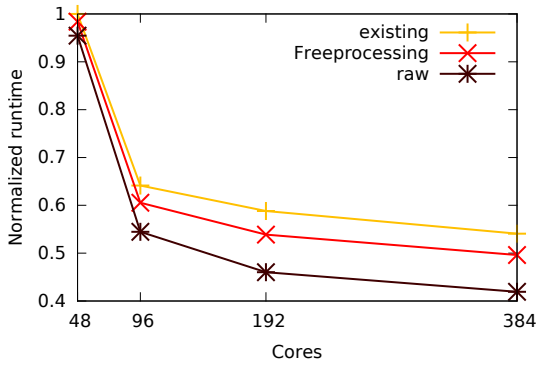

Figure 6.3: Scalability of the PsiPhi simulation. 'existing' and 'Freeprocessing' produce the same outputs via different mechanisms, while 'raw' produces only restart files. Freeprocessing's overhead is negligible; new output methodologies can even increase performance.

faster with the *free*processor. The performance difference arose from the difference in how PsiPhi and the *free*processor organize their writes. In the *free*processor, we calculate the appropriate file offsets on each rank and output to a shared file directly; the original PsiPhi approach was to gather the data on the root processor and then do all writing from there.

6.3.2 Enzo

Enzo is a simulation code designed for rich, multi-physics hydrodynamic astrophysical calculations [129]. It is of special interest in the visualization community due to its use of adaptively-refined (i.e., AMR) grids. Enzo runs in parallel via MPI and CUDA on some of the world's Top 500 supercomputers, with OpenMP hybrid parallelism under investigation. For I/O, Enzo relies on the HDF5 library.

As Enzo is HDF5-based and HDF5 provides all the data semantics required, the selection of which fields are of interest is the only required work. For HDF5 outputs, the symbiont configuration file specifies the 'Datasets' (in the HDF5 sense) of interest as opposed to a filename; the symbiont assumes that all HDF5 files opened for write access are a simulation output.

When Enzo was first investigated, HDF5 support was not available in our symbiont. Generic HDF5 support in the symbiont required only a day of effort. Configuring it to work with Enzo takes seconds. Users must edit a text file to indicate which field[s] they wish to see. To work with Enzo's `yt` tool, we

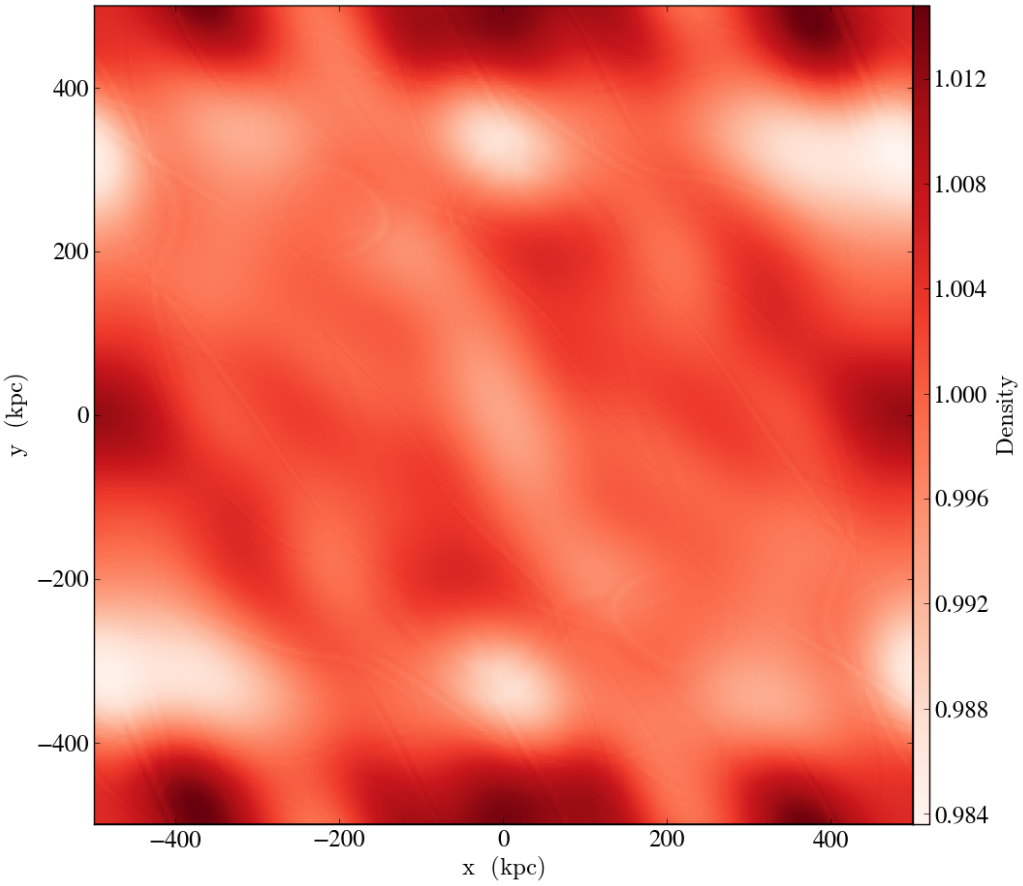

Figure 6.4: 'Density' field generated in situ by the Python visualization tool 'yt' applied to an Enzo hydrodynamics simulation. A *free*processor exposed the data into Python and a standard yt script created the visualization.

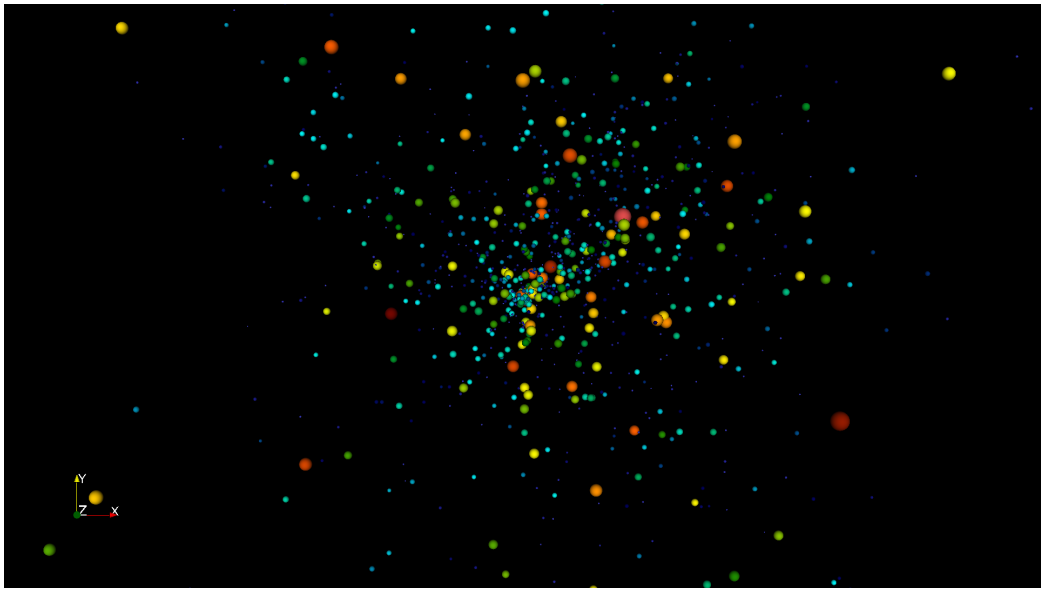

Figure 6.5: Sample frame from an animation produced from a student's simulation using our tool. The ease of use allowed the student to quickly get the tool running, allowing fast and simple visual debugging.

utilize the aforementioned *free*processor that exposes data into Python and runs a script (§ 6.2.3); the script we utilized is a standard yt script, except that it pulls its data from the special '`freeprocessing`' import, instead of a file. Figure 6.4 demonstrates this. The 100-line *free*processor is applicable for any *in situ* application; the 20-line Python script is specific to yt.

6.3.3 N-Body simulation coursework

We taught a course in High-Performance Computing during the preparation of this manuscript. Among the work given in the course was an MPI+OpenMP hybrid-parallel N-Body simulation. We provided our symbiont to the students along with a simple ParaView script that would produce a visualization given one of their timestep outputs. A sample visualization is shown in Figure 6.5.

The flexibility of the system was a boon in this environment. Visualizing the data in-memory would be difficult. The data were distributed, and the writes were in ASCII; parsing the data from the given stream was daunting for undergraduates. Therefore they elected to delay launching ParaView until after a timestep

completed. The system must write and then read particle information from disk, but visualization was still concurrent with simulation and faster than serializing the two tasks. Most importantly, the simplicity allowed application of the technique in *tens of minutes*.

6.4 Alternative use cases

The ability to hook into *any* data movement operation of a process enables *Free*processing to create novel applications of *in situ* ideas. In this section, we highlight a couple uses that make *Free*processing unique among *in situ* tools.

6.4.1 Transfer-based visualization

A heretofore lost opportunity has been in applying visualization methods to data *during transport from site to site*. This use case shares the primary motivation behind prior *in situ* visualization work: that we should do operations on data while they are *already* in memory, instead of writing the data to disk and then reading them back. While most if not all HPC experts agree that—at the largest scale—moving data will no longer be viable, a large userbase still exists for which simulation on a powerful remote supercomputer and analysis on local resources is the norm.

To downplay this drawback, we propose preprocessing during this transit time. As an example of *Free*processing for this novel case, we use it to instrument the transfer of a dataset using the popular secure copy (`scp`) tool. The system works by intercepting data as it goes out to or comes in from a socket. The source of the secure shell program itself needs no modification; the system could work with any network service, such as an FTP client or a web browser.

One use case is the computation of an isosurface; Figure 6.6 shows an example. A *free*processor computed this isosurface of a Richtmyer-Meshkov instability during network transfer. This example demonstrates one of the issues with our system: we needed to modify a marching cubes implementation to work in a slice-by-slice manner, as opposed to assuming all data were in-core. Additionally, our marching cubes implementation required at least two slices to operate, which necessitated a cache in the *free*processor to make up for the small writes utilized by `scp`. This buffering and our unoptimized marching cubes implementation slows down a gigabit-link transfer by 4x. Although this still proved faster than transferring the dataset and computing the isosurface in series, it highlights the pain

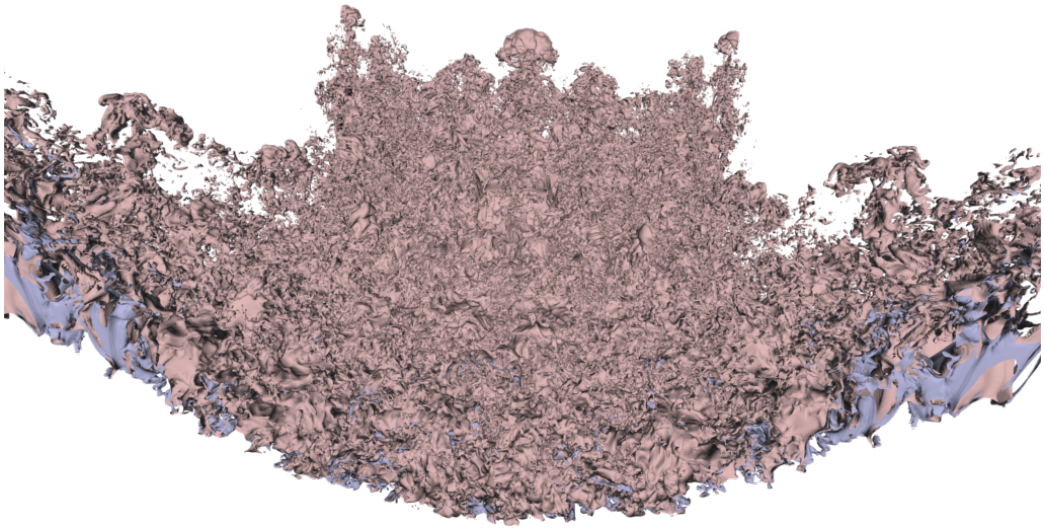

Figure 6.6: Richtmyer-Meshkov instability isosurface computed by a *free*processor. Whereas the *free*processor could be applied to any process that moves data, this particular isosurface was computed during network transfer via `scp`.

associated with the need to rewrite code in a stream processing fashion. On the other hand, with the rise of data parallel architectures and the decreasing memory per core ratio, one might argue that a transition to a stream processing model is inevitable.

6.4.2 MATLAB

Users often request methods to read outputs of binary-only commercial software in tools like VisIt.[1]. We implemented a *free*processor that accepts raw data, reads a metadata description from a configuration file for semantics, and exports these data into a Silo file that VisIt can easily import. Applying this *free*processor incurs an additional overhead of 3–10% on a simple Julia set calculation in MATLAB, due to the additional data that it writes.

The alternative of an 'export to Silo' MATLAB extension has notable drawbacks. First, one must compile using the '`mex`' compiler frontend, and every major MATLAB update will require a recompilation or even rewrite. Second, divorcing the code from MATLAB and its interface may require significant effort. In con-

[1] c.f. "Using MATLAB to write Silo files to bring data into VisIt", `visit-users` mailing list, February 2014.

trast, our *free*processor is indendendent of the MATLAB version it instruments, with neither source changes nor a recompilation required. Furthermore, the same *free*processor is applicable in other manners, such as creating Silo files during a network transfer.

6.5 Conclusions

In this paper we have introduced *Freeprocessing*: an *in situ* visualization and analysis tool based on binary instrumentation. The method imbues an existing simulation with *in situ* powers, with little or—in some cases—no effort on the part of the simulation author. The method's generality enables novel applications, such as visualization during network transfer or instrumenting software for which source is unavailable.

The system is, however, not without its drawbacks. The symbiont is stable, but customizing the system via new *free*processors can require per-simulation effort. Furthermore, the unidirectional communication model precludes simulation steering applications. The ability of *Free*processing to insert small, *ad hoc* bits of code in myriad new places uncovers perhaps its greatest limitation: increased programmability requires increased programming.

The work presented here lowers the barrier of entry for a simulation to indulge in *in situ* processing. Previous work on *in situ* has largely focused on achieving highly scalable results, with less regard to the amount of integration effort required. The most significant contribution of this work may be that fruitful capabilities can arise from a modicum of effort.

Chapter 7

Metadata inference for *in situ* visualization

Coupling visualization and analysis software with simulation code is a resource-intensive task. As the usage of simulation-based science grows, we asked ourselves: what would it take to enable *in situ* visualization for *every* simulation in existence? This paper presents an alternative view focusing on the **approachability** of *in situ* visualization. Utilizing a number of techniques from the program analysis community and taking advantage of commonalities in scientific software, we find that we can vastly reduce the time investment required to achieve visualization-enabled simulations.

7.1 Introduction

In situ visualization has proven to be useful for simulation-based sciences. The majority of *in situ* visualization literature is focused on the performance story: the growing size of outputs from simulations makes the commonplace post-processing regime less attractive [109, 111, 110]. While these efforts push us in the right direction, the impetus is flawed. The post-processing approach is not inferior because it scales poorly—though it does indeed scale poorly—it is *intrinsically* inferior. The ability to visualize and understand a simulation's data as it is generated is *inherently useful*. The *in situ* approach has not been ignored until recent years because it was not useful. A more likely explanation is that the difficulty was prohibitive.

Let us redefine '*in situ* visualization' as 'interactive simulation'. Interactive simulation is simulation that can be controlled: sped up or slowed down, reinitialized with new parameters, visualized, selectively refined, or even have its underlying physics live-edited. This model of simulation is clearly superior to our present batch-oriented model. The cycle time from hypothesis to verification would be greatly reduced.

Hall et al. note [130] the importance of "program-analysis strategies to improve software construction, maintenance, and evolution." We introduce a methodology for "0 day" coupling of simulation and visualization code. We remove the need to link in *any* external code to the simulation. The simulation software often does not even need to be recompiled. The bulk of our contribution is in the form of program understanding: we demonstrate how to infer those data that are interesting *as well as* the parameterization of those data that enables visualization. This obviates the need for the simulation author to conform to or even learn external APIs.

7.2 Trivial example

Consider the task of modifying an existing simulation program to interactively visualize its in-progress results. The developer must identify the primary loop that advances the state of the simulation. That loop modifies some memory of interest that the developer typically has some external knowledge about. The external knowledge generally revolves around data type and format: 'a point cloud of 64-bit floating point values', for example. This in turn helps the developer search for memory matching that organization. Once the location where the simulation advances its state is discovered, metadata is uncovered via a similar process, and a call to the *in situ* visualization library is inserted.

The code fragment in Listing 7.1 is exemplary of this task. The developer adding *in situ* visualization to a 2D simulation would be pleased to find these loops: the code is accessing and updating a 2-dimensional structure, 'data'. Next tasks would be to identify the type and source of the memory in 'data'. Should it align with the developer's ideas of the simulation's data model, visualization will be inserted after the loops and testing would be done.

The work presented here automates this exploratory search-and-insert-visualization process.

```
for(size_t j=0; j < dims[1]; ++j) {
  const size_t row = j*dims[0];
  for(size_t i=0; i < dims[0]; ++i) {
    data[row+i] = (S(x-1,y-1) + S(x-0,y-1) + S(x+1,y-1) +
                   S(x-1,y-0) + S(x-0,y-0) + S(x+1,y-0) +
                   S(x-1,y+1) + S(x-0,y+1) + S(x+1,y+1)) / 9.0
  }
}
```

Listing 7.1: A code fragment representative of simulation software. A large array is smoothed using a set of nested loops. `S` is presumed to be a macro that samples `data` while properly accounting for edge cases.

7.3 Program model and simulation analysis

In this section we develop an abstract model of an executing simulation program. We will then use this general model to describe a collection of properties that code such as that in Listing 7.1 follows. This set of properties codifies the aforementioned developer process.

$$
\begin{aligned}
BaseType &:= Booleans \cup Integers \cup FP \cup Strings \\
Type &:= BaseType \cup Array \cup Pointer \\
Memory &:= Heap \cup Static \cup Local \cup Arguments \cup Text \\
IPtr &\in Text \\
T &:= Memory \mapsto Type \\
BT &:= Memory \mapsto BaseType \\
Fqn &:= [begin \in Text, end \in Text] \mid\ begin < end \\
Where &:= Text \mapsto Fqn \\
Wr &:= (m \in Text) \mapsto (n \in (Memory \setminus Text)) \mid m \neq n \\
\textbf{class} &\quad\ CFG_{Node}\ address\ edges \\
CFG &:= \{n \mid n = CFG_{Node}\} \\
BB &:= Text \mapsto CFG \\
K &:= Text \mapsto CFG_{Node} \\
Hdr &:= CFG_{Node} \mapsto Boolean \\
Depth &:= CFG_{Node} \mapsto Integer
\end{aligned}
$$

Listing 7.2: Definitions for an abstract machine and analysis based on control flow properties.

We use the formalisms given in Listing 7.2. We consider an abstract machine described by an *instruction pointer* and the current state of *memory*. The instruction pointer advances automatically, and memory operations consist of reads and writes that map an address to a mutable memory location. Note that our definition denies self-modifying code. Memory is assumed to be *typed*, with a small set of available types. The T and BT mappings define mappings from memory locations to type information.

The running process is assumed to consist of a series of *functions* (Fqns), that are defined as the functions' upper and lower addresses. We make use of an inverse mapping $Where$ that allows us to identify a function from the current instruction pointer. We build local *control flow graph*s (CFGs) that describe the potential execution paths. These graphs are represented as a set of *nodes* that contain an entry *address* as well as a set of *edges*. We build these CFGs based on the function address range. We define a mapping K that allows us to identify a node in the control flow graph from an instruction address. We define two final mappings from a node in the control flow graph: 1) a predicate identifying *loop headers*

(Hdr), and 2) a mapping for the calculated *loop depth* ($Depth$). In Listing 7.1, the basic blocks containing `j < dims[1]` and `i < dims[0]` would be the loop headers. Loop depth is the nesting level of the provided basic block. In Listing 7.1, the assignment to `row` has a depth of 1, whereas assignment to the element in `data` has a loop depth of 2.

Using this model of program execution, we consider the problem of automatically identifying memory regions that house data that a user would want to visualize. We model these as a set of constraints on type classes. An instance of the type class allows one to visualize data within a simulation.

We currently support searching for N-dimensional ("ND") data arrays. This type is parameterized by a `base` address, a `length` (in bytes), the number of dimensions `ndims`, an array of dimensions `dims`, and finally the type of the data.

$$
\begin{aligned}
\textbf{class}\quad & ND \quad base\ length\ ndims\ dims\ type \\
& \wedge \quad \exists m \in Heap : base \rightarrow m && (7.1)\\
& \wedge \quad BT(base) = type && (7.2)\\
& \wedge \quad T(dims) \in Array \cup Pointer && (7.3)\\
& \wedge \quad BT(dims) \in Integers && (7.4)\\
& \wedge \quad T(ndims) \in Integers && (7.5)\\
& \wedge \quad ndims > 0 && (7.6)\\
& \wedge \quad Wr(IPtr) \in [base, base + length] && (7.7)\\
& \wedge \quad \exists b \in BB(Where(IPtr)) : \\
& \qquad \wedge\ IPtr \notin b \\
& \qquad \wedge\ Hdr(b) \\
& \qquad \wedge\ Depth(K(IPtr)) > Depth(b) && (7.8)
\end{aligned}
$$

We use the $\rightarrow$ notation to mean "points to"; the first constraint simply states that the data of interest live on the heap. As simulation data is large, it rarely fits on the stack or even in statically initialized memory. The second constraint conveys that the base type matches a parameter of our model, such as FP (floating point). The third and fourth constraints dictate that the dimensions are stored in a linear list of integers, and the fifth and sixth say that that the length of that dimension list is a positive integer.

The 7th and 8th constraints are complex and intertwined. First, the application must write into the relevant memory block. Secondly, the basic block where the

data are written must be deeper than another basic block that contains a loop header. That is to say that the data write occurs within a loop.

The formulation gives rise to a pattern matching problem. The **`class`**es of interest are the patterns, and the space to match within is the running process' `Memory`. In Listing 7.1, the parameter bindings are: `data` for `base`, the size of the allocation (not shown, but assumed to be `dims[0]` × `dims[1]` × `sizeof(float)`) for `length`, `dims` for `dims`, and 2 for `ndims`.

We do not claim our set of properties is perfect, though they have proven remarkably effective for our uses thus far. There are a number of promising approaches for discovering new invariants automatically [131, 132, 133], as well as low-hanging fruit (e.g. 'the memory is written to a file'). In the future, we wish to allow simulation authors to specify these constraints at runtime. The penalty for a lax specification would be too many visualization windows popping up; the penalty for too strict a specification would be too few windows popping up. The author would see either error almost immediately.

7.4 Implementation

We seek to realize the aforementioned 'search' for a given simulation process. At first glance static analysis is the best tool for this task, however it runs into difficulties proving some of the properties. The pernicious problem is aliasing in C-based languages. A statement as trivial as '`x = &y;`' creates two names for the same memory; thereafter, proving that a write to '`*x`' changes or does not change '`y`' can be anywhere from unambiguous to undecidable.

A lesser reason to shy away from static analysis is the computational expense, of which the largest for us is building control flow graphs. Especially for languages that enjoy methods for exponential code expansion (e.g. C++ templates), building CFGs for the entire program would be prohibitively expensive. A more targeted mechanism is desirable.

For this and other reasons we utilize static analysis techniques judiciously sourced by dynamic analysis. To receive a notification when a particular memory location is written, we use page protection to detect writes to it. Address `0xdeadbeef` is address `0xdeadbeef`, regardless of the variable name used to access it, and thus there is no need to solve the aliasing problem. Control flow graph construction and analysis are still expensive, but we isolate their construction to the functions of interest. By targetting unstripped binaries that include debug information, we can obtain robust type information. Of particular note and perhaps

surprise to many in the scientific visualization community, binary analysis need not be lossy as compared to source-based analysis [134]. An advantage of targeting x86-64 machine code is that it is language-agnostic: our framework works for C++ and Fortran as easily as it works for C, at a fraction of the implementation costs.

Not all of the 8 aforementioned properties are worthy of exposition; variable type information, for example, is straightforwardly sourced from the binary's debug information. In the next subsections, we focus on three of the larger issues: efficiently tracking memory, control flow graph analysis, and teasing out the dimensions of an array from the instruction stream of the code accessing it.

7.4.1 Memory tracking

Any heap-allocated memory might potentially be of interest to us. We utilize a `ptrace(2)`-based supervisor on the target program, and model each allocation using the finite state machine in Figure 7.1[1]. Memory regions begin in the 'null' state and change state based on events observed in the simulation process. This event tracking induces overhead, but in the absence of events simulation execution proceeds at native speeds.

Allocations cause us to begin tracking a memory region. We implement this event notification using a breakpoint on `malloc` calls. By examining the stack and return value, we create a map of the heap-allocated memory in the process. Overhead for this operation is predominantly context switching between the simulation process and our supervisor.

The `allow` and `deny` states solve the access detection problem. As mentioned earlier, solving the aliasing problem would be prohibitively expensive. Inserting checks at every instruction that modifies memory is another alternative, but Antoniu and Hatcher previously demonstrated this to be too expensive for our needs [135]. Instead, we rewrite allocations of interest, enabling write protection on the returned memory. This causes the simulation to trap when altering the data of interest, notifying our supervisor. To avoid the performance issue of a notification on *every* access, we catch only the first per function.

Handling a signal on every dynamic memory access would be prohibitively expensive. Therefore we detect the first access within a function and then allow unfettered memory access until that function returns. During return, we re-enable protection and execute the defined visualization steps on the memory. This assumes

[1] `ptrace` was consciously chosen for portability. Previous work relied on `LD_PRELOAD` [6], and that created issues porting to some supercomputers.

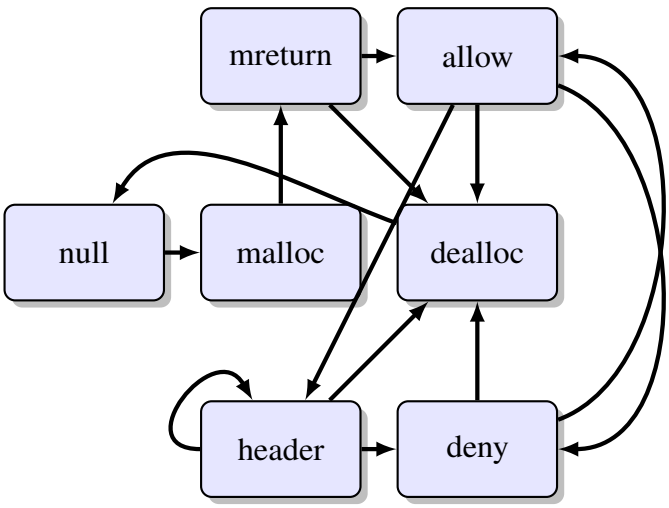

Figure 7.1: Finite state machine governing memory regions of interest. Regions transition between the states based on events observed in the observed simulation process. Basic information is obtained in the *malloc* and *mreturn* states. The *allow* state initializes parameters for visualization and enables unfettered access to the memory. *header* states build up the dimensions of the data. The *deny* state reenables access detection.

the simulation accesses memory within nested loops instead of accessing the memory via a functional indirection. In practice this has not been a problem: simulations authors are incentivized to access memory in this manner for performance reasons with or without our supervisor.

7.4.2 Control flow

The memory tracking described above enables our supervisor to track most of the events it needs. To pinpoint the remaining events we use analysis based on the local control flow. When a region is accessed, we build the local control flow graph for the currently-executing function. Our supervisor computes common compiler analysis information and uses the results to define per-node depth as well as perform loop header identification, as shown in Figure 7.2.

Our loop header identification relies on the common definitions of *reachability* and *dominance* [136]. Our current algorithm is known to be fallible in the presence of harmful `gotos`, but we have found it is reliable in practice. We deem a basic block to be a loop header when the basic block:

1. has exactly 2 in-edges,
2. has exactly 2 out-edges,
3. is reachable from one or both out-edges,
4. is dominated by exactly one in-edge, and
5. the dominating in-edge does not directly-dominate the other in-edge

In the future, we hope to simplify our flow graphs into loop trees [136]. This will resolve some of the possible ambiguities and modestly improve memory consumption.

We currently use Dyninst's ParseAPI [137] to compute the initial graph, and then perform the analysis with custom code. Other tools in this domain are DynamoRIO [138], Pin [139], and Valgrind [140]. All of these tools are capable of sophisticated binary transformations. However, our needs are modest and Dyninst presently represents the majority of our overhead. In the future we hope to replace this with custom graph construction code that can more effectively limit computation to the region of interest.

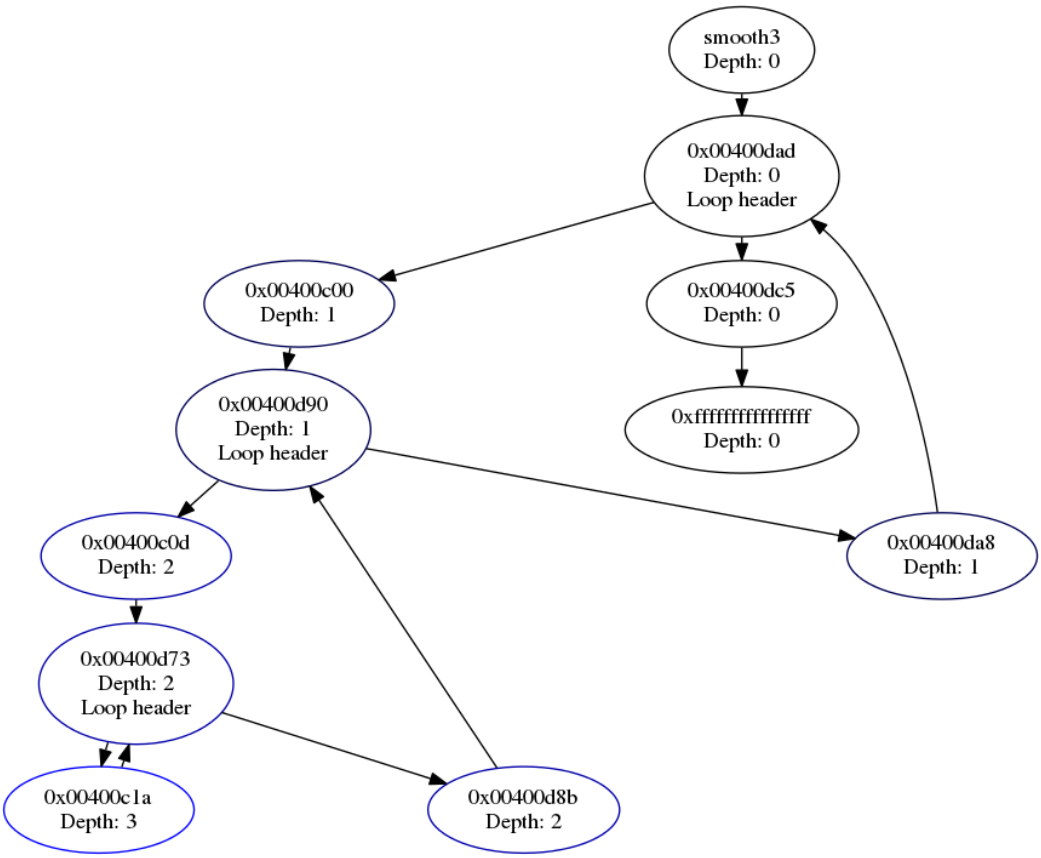

Figure 7.2: Simplified control flow graph for a small function that smooths a 3D array (the 3D analog of Listing 7.1). Analysis identifies loop headers and the nesting level ('Depth') of each basic block. On access, the loop tree is traversed to determine the dimensionality of the array.

7.4.3 Symbolic execution

As described in the **class** *ND* of Section 7.3, we assume a relation between loop headers and the basic blocks that are contained within those loops and accessing memory. The loop variable must be involved: if it were not, the access would be loop-invariant and hoisted out of the loop, either explicitly by the programmer or implicitly by the compiler. We assume a stronger relation, however: that the loop conditions imply the dimensionality of the memory regions accessed therein.

Loop conditionals do not definitively describe the format of the data. We have however found them to be remarkably accurate, and the loop nesting to be practically infallible. Still, we allow the user to override these discovered bounds, at which point our tool degrades to only identifying *where* visualization should be performed. More work is needed in this area.

Our general approach is to differentiate the loop bound from the induction variable in the loop conditional. Listing 7.3 gives the basic block for a real-world loop conditional (what might be implemented for `i < dims[0]`):

```
MOV %rdx, [%rip+0x20507]
MOV %rax, [%rpb-0x60]
CMP %rdx, %rax
JB -0x275
```

Listing 7.3: Instructions within a sample loop header. The induction variable and the loop bound appear as arguments to the `CMP` instruction.

We would like to know which of `%rdx` and `%rax` in Listing 7.3 is the loop bound. Unfortunately a myopic view of the `CMP` instruction is insufficient for operand classification: the *source* of the values is in the two `MOV` instructions. We use Algorithm 4 to track the source of operands by interpreting each instruction in the basic block. A heuristic that the induction variable is a local variable is used to differentiate the loop bound from the induction variable.

It is not strictly true that induction variables must be local variables. However, we have only seen this assumption violated in artifically-constructed test programs. Data dependency information and def/use sets [136] should make this more robust in the future.

Each iteration of this process gives a single loop bound. By following the state machine in Figure 7.1 and setting breakpoints up the chain of the loop tree, we derive the full set of bounds. At the function boundary, we enter the ‘deny’ state and re-enable memory protection for that region.

Algorithm 4 Tracking virtual register sets to identify the source of data. The algorithm begins at a loop header basic block and symbolically executes each instruction. The resultant data structure can be used to query the source of an instruction operand's value.

```
1:  register[*] := UNKNOWN
2:  instruction := bb_addr                          ▷ first instruction in loop header
3:  repeat                                          ▷ foreach instruction in the basic block
4:      if instruction.Opcode = MOV then
5:          mov := (MovInstruction)instruction
6:          if mov.source ∈ register then
7:              register[mov.target] := register[mov.source]
8:          else if mov.source ∈ Memory then
9:              register[mov.target] := mov.source +
10:                                     memdiff[mov.source]
11:         end if
12:     end if
13:     if instruction.Opcode = ADD then                ▷ track Δaddr
14:         add := (AddInstruction)instruction
15:         if add.dest ∈ register ∧ register[add.dest] ≠ UNKNOWN then
16:             register[add.dest] += add.source
17:         end if
18:         if add.dest ∈ Memory then
19:             memdiff[add.dest] += add.source
20:         end if
21:     end if
22:                                 ▷ ... SUB, MUL, etc. cases omitted for brevity
23:     instruction := next(instruction)
24: until instruction.Opcode = CMP
```

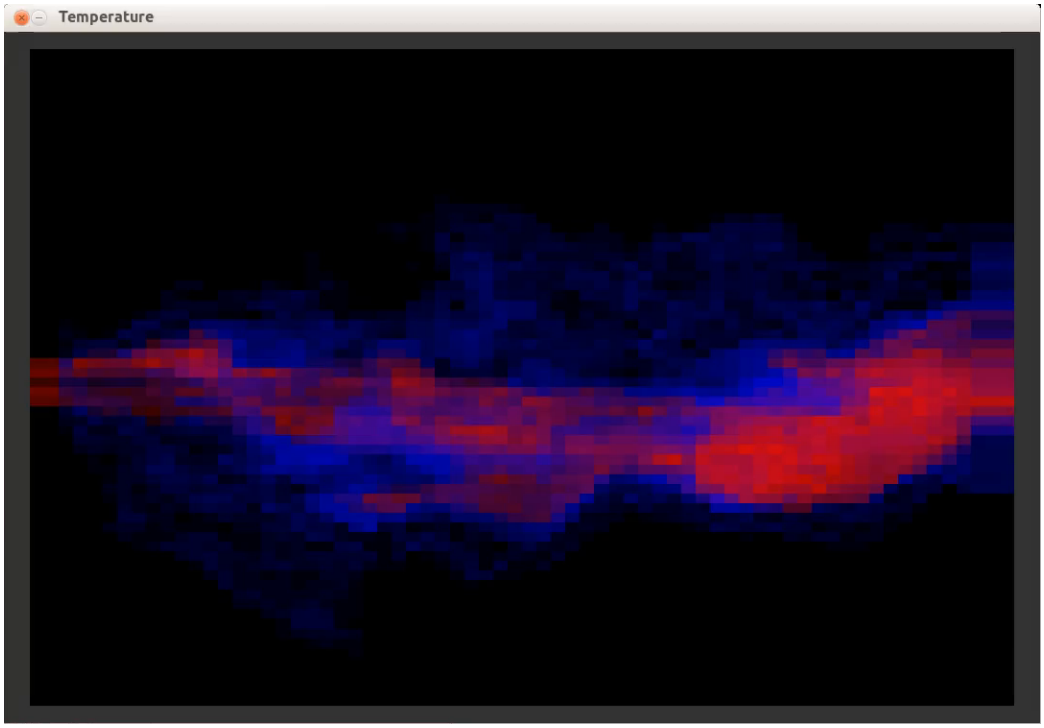

Figure 7.3: Volume rendering of the temperature field from a *PsiPhi* simulation [141]. Array shape information and data were pulled from the running simulation and given to an *ad hoc* volume renderer. Instrumentation and rendering time is on the order of milliseconds whereas a timestep can take seconds. User interaction was limited to transfer function design.

7.4.4 Visualization

Our contribution is in the approach to program understanding, though rendering is required to demonstrate these aspects. We have implemented a simple GLSL-based volume renderer and a python-based `yt` [124] backend thus far. Figure 7.3 shows the former with data sourced from a combustion simulation. In the future, we hope to incorporate backends using established visualization tools such as ImageVis3D, VisIt, and/or ParaView [2, 13, 42].

7.4.5 Performance

Performance is a cause for concern, as our instrumentation's theoretical upper bound is on par with `valgrind`-level instrumentation [140]. Fortunately, in practice we have found the slowdown to be approximately 4x for real-world programs. There is much work still to be done in this regard: the largest limitation is that we currently visualize *every* timestep's results.

Figure 7.4 looks at multiple aspects of performance across this set of programs. The red 'Uninstrumented' bar represents an upper bound on performance. 'Trace' inserts breakpoints at '`malloc`', '`free`', and their return addresses, measuring what it costs to start and stop the execution of the simulation program. Simulations that utilize more regions of dynamic memory will see higher overheads due to this aspect. However, the graph does not capture the phased nature of these processes: generally, our instrumentation is heavy for allocation-heavy startup routines and lightweight thereafter.

Figure 7.4's 'Relax' and 'allocs' are artificial programs constructed to illustrate overheads. The main component of 'Relax' is Listing 7.1. 'allocs' does nothing but allocate memory, the worst case for our instrumentation. We note that real-world programs experience considerably lower overheads, with the popular Linpack seeing a modest 15% slowdown.

Linpack is the matrix-vector multiplication benchmark of floating point performance that is used to rank supercomputers in the popular 'Top500' list. *Relax* is a program that identifies the steady state for the case of a plane connected to a constant heat source. The program's ratio between function calls and accessing the data to be visualized is at parity, stressing the memory access and analysis aspects of our supervisor. *PsiPhi* is a real-world computational fluid dynamics solver that focuses on Large Eddy Simulation (LES) of flows that include combustion and other types of chemical reactions [141]. *allocs* is a test program that simply allocs and frees memory without ever acessing it.

'Allocations' is a more expensive variant of the 'Trace' benchmark. In addition to allocation interception, this version reads relevant information from the client process so that it may generate a report of the [de]allocations made by the process. The traces this algorithm creates might be useful in producing and analyzing heap usage over time, in the same manner as Valgrind's 'Massif' [142]. We note that this adds little additional overhead to the instrumentation, demonstrating that reading memory from the process is cheap.

In practical terms, the performance scales with 1) the number of allocations the program performs, and 2) the number of allocations that are tracked and provide

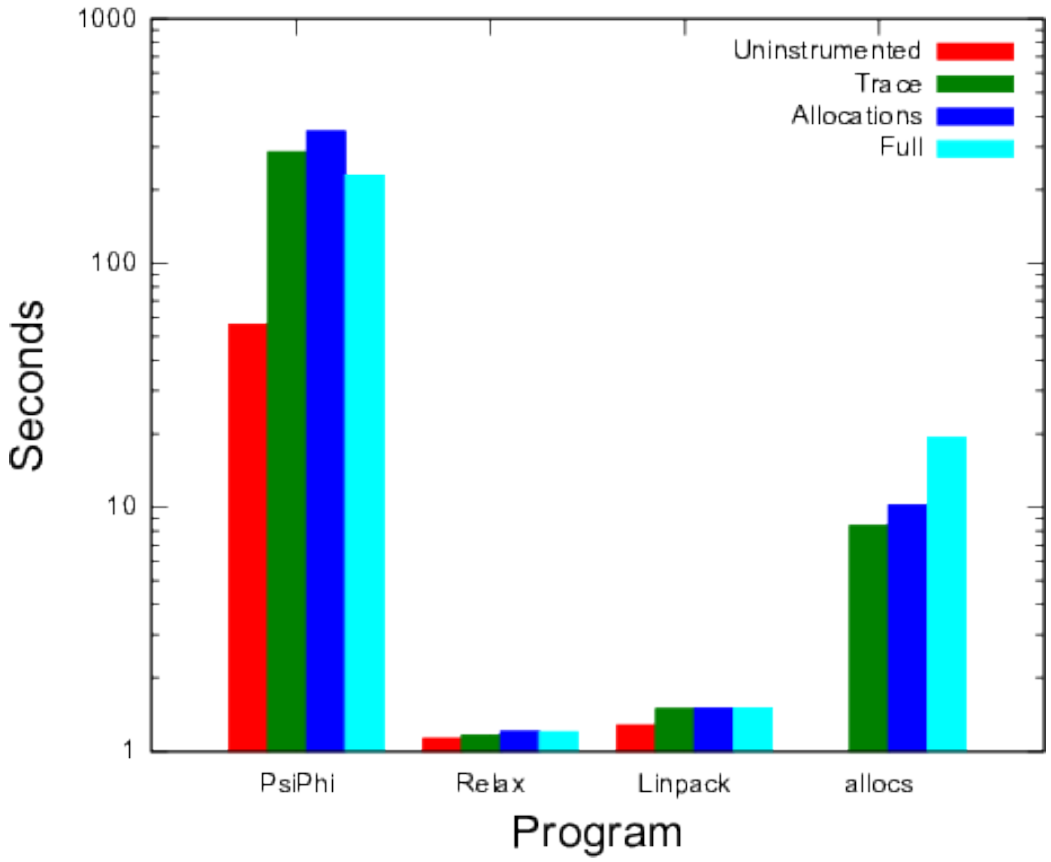

Figure 7.4: Performance of our evaluation programs under different instrumentation scenarios. Note logarithmic scale. 'Uninstrumented' is the runtime of the simulation without our interference. 'Trace' interrupts for allocations; 'Allocations' *reports* allocations as well, which requires reading more data from the instrumented program. 'Full' does allocation tracking, access detection, analysis, and visualization of the data.

source locations for analysis. Reducing the number of allocations requires changing the programs of interest, which is counter to our goal of a transparent solution. However, avoiding the tracking infrastructure for memory that the user is not interested in is a plausible practical mechanism by which the user can influence the performance of the instrumented system.

7.5 Conclusions

We have elucidated a method and demonstrated a prototype that eliminates the surface area between simulation code and visualization tool. By recovering the loop structure of a target binary and carefully instrumenting memory accesses, one can automatically insert visualization at appropriate places in a running simulation.

7.5.1 Future work

The most glaring present omission is the lack of support for data types beyond regular N-dimensional grids. An obvious next target is related data types such as adaptive mesh refinement data. Curvilinear grids may prove simple as well, and meshes or point clouds would certainly be of interest. An area of uncertainty is in data decomposition in distributed memory simulations.

We do not seek to replicate the full functionality of tools like VisIt or ParaView. We must therefore couple with one of these tools; doing so would immediately increase the utility of our prototype implementation.

We make a number of assumptions that are practically but not strictly true. Each requires more investigation, and aspects such as the specification used in our search require more user control than we presently have made available.

While some of these issues involve significant engineering efforts, the work presented here demonstrates that there is no need to modify simulation code to inject *in situ* visualization. We hope this encourages others to pursue 0-modification approaches to *in situ* visualization.

Chapter 8
Conclusions and future work

In this dissertation, we have:

- demonstrated a forward-looking architecture for volume visualization,
- supported the approach of ray-guided rendering with extensive benchmarks,
- established the multi-scale parallelism architecture that future visualization—and HPC—work is following,
- identified a number of best practices for the remaining largest challenge in performance-oriented visualization: IO, and
- outlined methods to profoundly simplify the practice of *in situ* visualization.

Here we offer some concluding remarks and projections based on the work developed here.

We began by outlining the volume rendering architecture, "*Tuvok*". The architecture strikes a balance between performance, practicality, and portability that few have done. Volume rendering modules in Voreen [43] have since followed a similar architecture. However, academic work tends to focus on particular hardware features whereas commercial work is often tied to particular platforms. Tuvok practically utilizes available, modern features of GPUs while addressing and falling back to less straightforward methods. More importantly, it has seen significant effort working around the nonintersecting shortcomings of modern OpenGL drivers, a critical component of its applicability. As such, Tuvok is the volume renderer that is most cited outside its field.

Chapter 3 described ray-guided rendering and provided a detailed performance analysis using a variety of real-world datasets. Comparisons with previous work, including even our own (Chapter 2), demonstrate that this is the best known method for large-scale volume visualization. The method is capable of providing subsecond response times with multi-terabyte datasets on workstation hardware, as in Figure 1.2, orders of magnitude faster than other volume rendering techniques. The majority of the performance increase comes from obviating the need to load large chunks of the data from disk, based on a fine-scale identification of early ray termination.

With the ray-guided rendering solution now known, the major impediment to high-performance volume rendering has become the data reorganization preprocess. Simply *reading* a terabyte from a commodity disk presently requires 2.9 hours, and the bricking needed in volume rendering consists of a great many scattered reads and writes. At the heart of this process is data movement: a well-known inscalable operation. Our dynamic rebricking technique introduced in Chapter 3 ameliorates this problem, but there is no true solution on visible on the horizon. More work is needed in this area.

We have established the scalability of desktop volume rendering solutions to be far beyond what was believed by the community. A single node solution is, however, unfit for extreme scale data sizes. While many of the same techniques developed on the desktop can help, Chapter 4 demonstrates that one must utilize multi-scale parallelism, both within and across nodes. This work also established in the community the superiority of the 'fat' node architecture, where large numbers of cores cooperate on a single node as opposed to a plethora of smaller nodes. The parallelization schemes at each scale must be considered distinctly.

Chapter 4 examined dynamic load balancing for volume rendering in detail. Unfortunately these tests were inconclusive. In many situations dynamic load balancing resulted in similar or even worse performance. Load balancers also include a number of tunable parameters that even experts have trouble setting. More work is needed in this area, both to ensure load balancing is beneficial and to establish auto-tuned parameters.

IO remains and will remain the major problem in large-scale visualization. Figure **??** shows that the rendering and compositing problems are dominated by IO at scale. While this image was from a specific strong scaling study, the story changes little if we replace the X axis by year: rendering and compositing times have plummeted as hardware capabilities have increased and the fruits of research efforts have been realized. Disk access has unfortunately not seen the same improvements.

We described a number of 'best practices' for IO-heavy visualization code in Chapter 5. One is to use large reads or writes to maximize throughput. If and only if this is impossible should one turn to space-filling curve methods to minimize access times. Despite conventional wisdom, the many files created by the '1 file per process' regime scales poorly. Most IO calls involve implicit synchronization, and staggering these operations can have a large impact on performance. Along the same line, delaying file closures until necessary can provide filesystems—especially distributed filesystems—the time needed to do effective buffering.

In situ visualization is playing an increasing role in simulation-based sciences. The widening disparity between memory and disk speed necessitates the approach at the largest scales. Furthermore, the increasingly multidisciplinary analyses performed on simulation runs encourages analysts at many different sites to participate, but the data are too large to be transferred between sites. Using knowledge of the sampling rate required by the analysis method may be the best way of performing otherwise computationally-infeasible analyses. All of these factors suggest a centralization of large-scale computing resources, in contrast to the decentralization that industry has experienced. It is not difficult to imagine a future where analysis *is* simulation: that all analysis is performed by (re)running simulations, as opposed to (re)processing simulation outputs.

In situ visualization is currently complex and difficult. Bespoke in situ visualization solutions abound, but there are presently only a couple visualization and analysis tools with *in situ* APIs that cater to a wide variety of data models. Stability is a problem with these large all-encompassing tools. This is in addition to the many difficult issues inherent to *in situ* visualization, such as balancing simulation time and visualization time.

Despite the growing importance of *in situ*, existing visualization tools couple with simulation software only with extreme difficulty. This heavy investment in tool coupling discourages *ad hoc* and exploratory solutions. Authors currently acquiesce this effort only under extreme pressure, which today means that data sizes and related IO performance exceed a high threshold. As the IO performance disparity widens, this threshold lowers. Eventually, the HPC community will need to consider *in situ* visualization for *all* simulation software.

Adding a new capability (even if it is the same capability) across such a large base of installed software is a daunting task. Much of the difficulty comes from the visualization software: large APIs to learn, relatively instable software, and complex methodologies for conveying metadata. These sharp edges can be filed down through (currently rather large) investments in learning the visualization tools of the day. However, other concerns are simulation-specific as opposed to

visualization-tool-specific. These include identifying *where* to insert calls into the visualization tool, and striking the correct balance between simulation time and visualization/analysis time.

Our work addresses these complexity issues head on. Chapters 6 and 7 outline methods that one can use to immediately solve the issues of injecting—at will—visualization and analysis capabilities into an entire corpus of simulation software. Chapter 7 takes the first steps at eliminating the complex metadata communication primitives. However, we must also ameloriate the untenable model of simulation authors becoming visualization experts. The approaches of both chapters put more onus on the efforts of visualization scientists, sweeping many visualization details under the rug from the simulation author's perspective. An important theme among these approaches is shifting responsibilities away from the simulation engineer and towards the visualization engineer.

Bibliography

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Thomas Fogal and Jens Krüger. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, November 2010.

[3] Thomas Fogal, Alexander Schiewe, and Jens Krüger. An Analysis of Scalable GPU-Based Ray-Guided Volume Rendering. In *Large Data Analysis and Visualization, IEEE Visualization*, 2013.

[4] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *Proceedings of High Performance Graphics 2010*, pages 57–66, June 2010.

[5] Thomas Fogal and J. Krüger. Efficient I/O for Parallel Visualization. In *Eurographics Symposium on Parallel Graphics and Visualization, Llandudno, Wales, UK*, pages 81–90, April 2011.

[6] Thomas Fogal, Fabian Proch, Alexander Schiewe, Olaf Hasemann, Andreas Kempf, and Jens Krüger. *Free*processing: Transparent in situ visualization via data interception. In *Proceedings of the 14th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV, Llandudno, Wales, 2014. Eurographics Association.

[7] Thomas Fogal and Jens Krüger. An approach to lowering the in situ visualization barrier. In *Proceedings of ISAV 2015: In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, Austin, TX / USA, November 2015.

[8] Thomas Fogal and Jens Krüger. Size Matters - Revealing Small Scale Structures in Large Datasets. Poster at the World Congress on Medical Physics and Biomedical Engineering, 2009.

[9] Thomas Fogal, Shanjie Jiao, Xue Yang, and Jens Krüger. Visualization as a bridge between chemical and nuclear engineering simulations. In *Proceedings of the 2010 American Institute of Chemical Engineering*, Salt Lake City, Utah, 2010.

[10] Tatjana Jevremovic, Thomas Fogal, Dong-OK Choe, Haori Yang, and Jens Krüger. The role of virtual engineering and emerging visualization tools in nuclear engineering education and training at the university of utah. In *Nuclear Engineering and Training Conference*, Prague, Czech Republic, May 2011.

[11] Jovana Knežević, Ralf-Peter Mundani, Ernst Rank, Hermilo Hernández, Tatjana Jevremović, and Thomas Fogal. Interactive computing in numerical modelling of particle transport methods. In *IADIS Conference on Theory and Practice in Modern Computing*, July 2012.

[12] Carson Brownlee, Thomas Fogal, and Charles D. Hansen. GLuRay: Enhanced ray tracing in existing scientific visualization applications using OpenGL interception. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50. The Eurographics Association, 2012.

[13] Hank Childs, Eric Brugger, B. Whitlock, Jeremy S. Meredith, Sean Ahern, Kathleen Bonnell, Mark Miller, Gunther Weber, Cyrus Harrison, Dave Pugmire, Thomas Fogal, Christoph Garth, Allen Sanderson, E Wes Bethel, Marc Durant, David Camp, Jean M. Favre, O. Ruebel, Paul Navratil, Matthew Wheeler, Paul Selby, and Fabien Vivodtzev. *VisIt: An End-User Tool for Visualizing AND Analyzing Very Large Data*, pages 357–372. CRC Press, October 2012.

[14] Hermilo Hernández, Jovana Knežević, Thomas Fogal, Todd Sherman, and Tatjana Jevremovic. Visual numerical steering in 3D AGENT code system

for advanced nuclear reactor modeling and design. *Annals of Nuclear Energy*, 55:248–257, 2013.

[15] Christopher Butson, Georg Tamm, Sanket Jain, Thomas Fogal, and Jens Krüger. Evaluation of Interactive Visualization on Mobile Computing Platforms for Selection of Deep Brain Stimulation Parameters. *IEEE Transactions on Visualization and Computer Graphics*, 19, January 2013.

[16] M.J. Ackerman, T.S. Yoo, and D. Jenkins. The visible human project: From data to knowledge. In *Proceedings of Computer Assisted Radiology and Surgery*, pages 11–16, 2000.

[17] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[18] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Vis*, pages 91–98, 1994.

[19] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[20] Joe Kniss, Gordon Kindlmann, and Charles D. Hansen. Multidimensional transfer functions for volume rendering. In Charles D. Hansen Chris R. Johnson, editor, *Visualization Handbook*, pages 189 – XVII. Butterworth-Heinemann, Burlington, 2005.

[21] Stefan Röttger, Michael Bauer, and Marc Stamminger. Spatialized Transfer Functions. In Ken Brodlie, David Duke, and Ken Joy, editors, *EuroVis 2005: Eurographics / IEEE VGTC Symposium on Visualization*. The Eurographics Association, 2005.

[22] C. Correa and Kwan-Liu Ma. Size-based transfer functions: A new volume exploration technique. *Visualization and Computer Graphics, IEEE Transactions on*, 14(6):1380–1387, Nov 2008.

[23] C. Correa and D. Silver. Dataset traversal with motion-controlled transfer functions. In *Visualization, 2005. VIS 05. IEEE*, pages 359–366, Oct 2005.

[24] Gunther Weber, S. Dillard, H. Carr, Valerio Pascucci, and B. Hamann. Topology-controlled volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):330–341, 2007.

[25] S. Bruckner and M. E. Gröller. Style transfer functions for illustrative volume rendering. pages 715–724, Sep 2007.

[26] Ivan Viola, Meister Eduard Gröller, Markus Hadwiger, Katja Bühler, Bernhard Preim, and David Ebert. Illustrative visualization. In Ming Lin and Celine Loscos, editors, *Tutorials of Eurographics*, pages 187–329. The Eurographics Association and The Image Synthesis Group, August 2005.

[27] Lujin Wang, Ye Zhao, Klaus Mueller, and Arie Kaufman. The magic volume lens: An interactive focus+context technique for volume rendering. *Visualization Conference, IEEE*, 0:47, 2005.

[28] Jens Krüger, Jens Schneider, and Rüdiger Westermann. ClearView: An interactive context preserving hotspot visualization technique. *IEEE Transactions on Visualization and Computer Graphics*, pages 941–948, 2006.

[29] Klaus Engel. Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware. In Dieter W. Fellner and Roberto Scopigno, editors, *State of the Art Reports*. Eurographics, 2002.

[30] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[31] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, and Daniel Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006.

[32] Compute graphics at stanford university: The volpack volume rendering library, 1995.

[33] Joe Kniss, Gordon Kindlmann, and Chuck Hansen. Simian, 2002.

[34] Inc. Colorado Research Associates Division of Northwest Research Associates. Ogle large-scale scientific data visualizer, 2002.

[35] C. Rezk-Salama, Klaus Engel, and F. V. Higuera. The openqvis project, 2002.

[36] Indiana University. Voxx: A Volume Rendering Program for 3D Microscopy, 2009. `http://www.indiana.edu/~voxx/`.

[37] Stefan Bruckner and Eduard Gröller. VolumeShop: An Interactive System for Direct Volume Illustration. In Cláudio T. Silva, Eduard Gröller, and Holly Rushmeier, editors, *Proceedings of IEEE Visualization (VIS 2005, October 23–28, 2005, Minneapolis, MN, USA)*, pages 671–678, 2005.

[38] J. Tian, J. Xue, Y. Dai, J. Chen, and J. Zheng. A novel software platform for medical image processing and analyzing. *IEEE Transactions on Information Technology in Biomedicine*, 6:800–812, 2008.

[39] A. Rosset. OsiriX, 2015. `http://www.osirix-viewer.com/`.

[40] Will Schroeder, Ken Martin, and Bill Lorensen. The visualization toolkit: an object-oriented approach to 3D graphics. January 2006.

[41] Terry S. Yoo, Michael J. Ackerman, William E. Lorensen, Will Schroeder, Vikram Chalana, Stephen Aylward, Dimitris Metaxas, and Ross Whitaker. Engineering and algorithm design for an image processing API: A technical report on ITK - the insight toolkit. In *Proceedings of Medicine meets Virtual Reality*, volume 85, pages 586–592, 2002.

[42] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The Visualization Handbook*, 717:731, 2005.

[43] Jennis Meyer-Spradow, Timo Ropinski, Jörg Mensmann, and Klaus Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13, 2009.

[44] R.S. Macleod, D.M. Weinstein, J.D. de St. Germain, D.H. Brooks, C.R. Johnson, and S.G. Parker. Scirun/biopse: Integrated problem solving environment for bioelectric field problems and visualization. In *Proceedings of the Int. Symp. on Biomed. Imag.*, pages 640–643, April 2004.

[45] Hank Childs, Eric S. Brugger, Kathleen S. Bonnell, Jeremy S. Meredith, Mark Miller, Brad J. Whitlock, and Nelson Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization 2005*, page 190–198, 2005.

[46] L. Bavoil, Stephen P. Callahan, P.J. Crossno, J. Freire, Carlos E. Scheidegger, Claudio T. Silva, and Huy T. Vo. Vistrails: enabling interactive multiple-view visualizations. In *IEEE Visualization*, pages 135–142, 2005.

[47] Aaron Knoll, Sebastian Thelen, Ingo Wald, Charles D. Hansen, Hans Hagen, and Michael E. Papka. Full-resolution interactive cpu volume rendering with coherent bvh traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium*, pages 3–10, 2011.

[48] Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, and Hanspeter Pfister. Interactive volume exploration of petascale microscopy data streams usign a visualization-driven virtual memory approach. In *Proceedings of IEEE Visualization 2012*, 2012.

[49] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press.

[50] F. Reichl, M. G. Chajdas, and R. Westermann. Hybrid sample-based surface rendering. In *Proceedings of the workshop on Vision, Modeling, and Visualization (VMV 2012)*, 2012.

[51] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[52] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[53] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[54] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, 2001.

[55] Eric C. LaMar, Mark A. Duchaineau, Bernd Hamann, and Ken Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Visual Data Exploration and Analysis VII*, volume 3960, pages 365–374, Bellingham, Washington, 2000. SPIE, The International Society for Optical Engineering.

[56] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmerman, and Thomas Ertl. Level-of-detail volume rendering via 3d textures. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, VVS '00, pages 7–13, New York, NY, USA, 2000. ACM.

[57] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[58] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, July 2008.

[59] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, May 2001.

[60] Hank Childs, Mark Duchaineau, and Kwan-Liu Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, May 2006.

[61] Mark Howison, E. Wes Bethel, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, May 2010. LBNL-3297E.

[62] J. Beyer, M. Hadwiger, J. Schneider, W.-K. Jeong, and H. Pfister. Distributed terascale volume visualization using distributed shared virtual memory. Poster at IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2012.

[63] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, 2002. ACM.

[64] Christopher Lux and Bernd Fröhlich. GPU-based ray casting of stacked out-of-core height fields. In *ISVC'11: Proceedings of the 7th international conference on Advances in visual computing*. Springer-Verlag, September 2011.

[65] Klaus Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard pcs. Poster at IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2012.

[66] Matthias Schott, Vincent Pegoraro, Charles Hansen, Kevin Boulanger, and Kadi Bouatouch. A Directional Occlusion Model for Interactive Direct Volume Rendering. In *EG Symposium on Visualization (IEEE-VGTC '09)*, volume 28, 2009.

[67] C. Müller, M. Strengert, and T. Ertl. Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, pages 59–67, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[68] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, June 1988.

[69] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, Jun 1995.

[70] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 169–177, New York, NY, USA, 1998. ACM.

[71] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. 14(4):23–32, July 1994.

[72] William M. Hsu. Segmented ray casting for data parallel volume rendering. In *Proceedings of the 1993 symposium on Parallel rendering*, pages 7–14, New York, NY, USA, Oct 1993. ACM.

[73] Kwan-Liu Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the*

1993 symposium on Parallel rendering, New York, NY, USA, Oct 1993. ACM.

[74] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 253–259, New York, NY, USA, 1984. ACM.

[75] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications*, 14:59–68, 1994.

[76] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. *Parallel volume rendering on the IBM Blue Gene/P.* Proc., pp. 73-80, Jan 2008.

[77] Tom Peterka, Hongfeng Yu, Rob Ross, Kwan-Liu Ma, and R. Latham. End-to-end study of parallel volume rendering on the IBM blue gene/p. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 566–573, Sept. 2009.

[78] Mark Howison, E. Wes Bethel, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, May 2010. LBNL-3297E.

[79] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, July 2002.

[80] S. Eilemann and Renato Pajarola. Direct send compositing for parallel sort-last rendering. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 29–36, 2007.

[81] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Parallel and Large-Data Visualization and Graphics, 2001*, pages 85–154, Oct 2001.

[82] Magnus Strengert, Marcelo Magallón, Daniel Weiskopf, Stefan Guthe, and Thomas Ertl. Hierarchical visualization and compression of large volume

datasets using gpu clusters. In *Proceedings of the 5th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '04, pages 41–48, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[83] S. Marchesin, C. Mongenet, and J.-M. Dischler. Multi-GPU Sort-Last Volume Visualization. In *EG Symposium on Parallel Graphics and Visualization (EGPGV'08), Eurographics*, April 2008.

[84] Stphane Marchesin and Jean-Michel Mongenet, Catherine an Dischler. Dynamic Load Balancing for Parallel Volume Rendering. In Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2006.

[85] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: Nfs version 3 protocol specification, June 1995. `http://www.ietf.org/rfc/rfc1813.txt`.

[86] Dean Hildebrand and Peter Honeyman. NFSv4 and high performance file systems: Positioning to scale. Technical Report citi-04-02, University of Michigan, 2004. `http://www.citi.umich.edu/NEPS/positions/hildebrand.pdf`.

[87] Inc. Sun Microsystems. Peta-scale I/O with the lustre file system. Oak Ridge National Laboratory/ Lustre Center of Excellence papers `http://wiki.lustre.org/images/9/90/Peta-Scale_wp.pdf`, February 2008.

[88] William J. Nitzberg. *Collective parallel I/O*. PhD thesis, University of Oregon, Eugene, OR, USA, 1995.

[89] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in panda. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[90] David Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Trans. Comput. Syst.*, 15:41–74, February 1997.

[91] Gokhan Memik, Mahmut Kandemir, and Alok Choudhary. Exploiting inter-file access patterns using multi-collective I/O. In *Proceedings of the 1st*

USENIX Conference on File and Storage Technologies, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[92] Wesley Kendall, Markus Glatter, Jian Huang, Tom Peterka, Robert Latham, and Robert Ross. Terascale data organization for discovering multivariate climatic trends. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 15:1–15:12, New York, NY, USA, 2009. ACM.

[93] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76 –82, jul. 1990.

[94] Jianwei Li, Wei keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, nov. 2003.

[95] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference, Volume 2 - The MPI-2 Extensions*, volume 2. The MIT Press, 1998.

[96] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[97] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[98] S. Hodson, S. Klasky, Q. Liu, J. Lofstead, N. Podhorszki, F. Zheng, M. Wolf, T. Kordenbrock, H. Abbasi, and N. Samatova. Adios 1.2.1 user's manual, August 2010. `http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.2.1.pdf`.

[99] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, John Shalf, and David Donofrio. Tuning hdf5 for lustre file systems. In *Proceedings*

of Workshop on Interfaces and Abstractions for Scientific Data Storage Heraklion, Crete, Greece, September 2010.

[100] Kwan-Liu Ma, Aleksander Stompel, Jacobo Bielak, Omar Ghattas, and Eui Joong Kim. Visualizing very large-scale earthquake simulations. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 48–, Washington, DC, USA, 2003. IEEE Computer Society.

[101] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 49–, Washington, DC, USA, 2004. IEEE Computer Society.

[102] Weikuan Yu, J.S. Vetter, and H.S. Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11, April 2008.

[103] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. I/o strategies for parallel rendering of large time-varying volume data. In *In Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Parallel Graphics and Visualization*, pages 31–40, June 2004.

[104] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/o performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA, 2009. ACM.

[105] Avery Ching, Wei keng Liao, Alok Choudhary, Robert Ross, and Lee Ward. Noncontiguous locking techniques for parallel file systems. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 26:1–26:12, New York, NY, USA, 2007. ACM.

[106] H. Childs, B. Geveci, W. Schroeder, J. Meredith, K. Moreland, C. Sewell, T. Kuhlen, and E.W. Bethel. Research challenges for visualization software. *Computer*, 46(5):34–42, May 2013.

[107] Ciprian Docan, Manish Parashar, and Scott Klasky. Enabling high-speed asynchronous data extraction and transfer using DART. *Concurr. Comput. : Pract. Exper.*, 22(9):1181–1204, June 2010.

[108] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, Michael E. Papka, and Scott Klasky. Examples of *In Transit* visualization. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, PDAC '11, pages 1–6, New York, NY, USA, 2011. ACM.

[109] Matthieu Dorier, Roberto R. Sisneros, Tom Peterka, Gabriel Antoniu, and Dave B Semeraro. Damaris/Viz: a nonintrusive, adaptable and user-friendly in situ visualization framework. In *Large Data Analysis and Visualization*, October 2013.

[110] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109. Eurographics Association, 2011.

[111] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jansen. The ParaView Coprocessing library: A scalable, general purpose *In Situ* visualization library. In *Large Data Analysis and Visualization*, pages 89–96. IEEE, 2011.

[112] Andrew C. Bauer, Berk Geveci, and Will Schroeder. *The ParaView Catalyst User's Guide*. Kitware, 2013.

[113] John Biddiscombe, Jerome Soumagne, Guillaume Oger, David Guibert, and Jean-Guillaume Piccinali. Parallel Computational Steering and Analysis for HPC Applications using a ParaView Interface and the HDF5 DSM Virtual File Driver. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *Eurographics Symposium on Parallel Graphics and Visualization*, pages 91–100, Llandudno, Wales, 2011. Eurographics Association.

[114] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM.

[115] V. Vishwanath, M. Hereld, and M.E. Papka. Toward simulation-time data analysis and I/O acceleration on leadership-class systems. In *Large Data Analysis and Visualization (LDAV)*, pages 9–14, October 2011.

[116] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[117] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in time: Adding value to the IO pipelines of high performance applications with JITStaging. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 27–36, New York, NY, USA, 2011. ACM.

[118] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. GoldRush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 78. ACM, 2013.

[119] Jean-Denis Lesage and Bruno Raffin. A hierarchical component model for large parallel interactive applications. *J. Supercomput.*, 60(3):389–409, June 2012.

[120] M. Ament, S. Frey, F. Sadlo, T. Ertl, and D. Weiskopf. GPU-based two-dimensional flow simulation steering using coherent structures. In P. Iványi and B. H. V. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom, 2011. Civil-Comp Press.

[121] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Comput. Graph. Appl.*, 30(3):45–57, May 2010.

[122] Ian T. Foster, J. Boverhof, A. L. Chervenak, Lisa Childers, A. DeSchoen, G. Garzoglio, D. Gunter, B. Holzman, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Livny, S. Martin, P. Mhashilkar, Z. Miller, T. Samak, M.-H. Su, S. Tuecke, V. Venkataswamy, C. Ward, and C. Weiss. Reliable high-performance data transfer via Globus Online. 06/2011 2011.

[123] Marc Gamell, Ivan Rodero, Manish Parashar, Janine C. Bennett, Hemanth Kolla, Jacqueline Chen, Peer-Timo Bremer, Aaditya G. Landge, Attila Gyulassy, Patrick McCormick, Scott Pakin, Valerio Pascucci, and Scott Klasky. Exploring power behaviors and trade-offs of in-situ data analytics. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 77:1–77:12, New York, NY, USA, 2013. ACM.

[124] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement*, 192:9, January 2011.

[125] M. Pettit, B. Coriton, A. Gomez, and Andreas M. Kempf. Large-eddy simulation and experiments on non-premixed highly turbulent Opposed Jet flows. *Proc. Combust.Inst.*, 33:1391–1399, 2011.

[126] T. Ma, O. Stein, N. Chakraborty, and Andreas M. Kempf. A-posteriori testing of algebraic flame surface density models for LES. *Combustion Theory and Modelling*, 2013.

[127] F. Cavallo Marincola, T. Ma, and Andreas M. Kempf. Large eddy simulations of the Darmstadt turbulent stratified flame series. *Proceedings of the Combustion Institute*, 34(1):1307 – 1315, 2013.

[128] Fabian Proch and Andreas M. Kempf. Numerical analysis of the Cambridge stratified flame series using artificial thickened flame LES with tabulated premixed chemistry. *submitted to Combustion and Flame*, 2013.

[129] The Enzo Collaboration, G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C So, F. Zhao, R. Cen, and Y. Li. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *Astrophysical Journal Supplement Series*, July 2013.

[130] Mary Hall, David Padua, and Keshav Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, February 2009.

[131] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 683–693, Piscataway, NJ, USA, 2012. IEEE Press.

[132] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 71–99, Berlin, Heidelberg, 2010. Springer-Verlag.

[133] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages #38; Applications*, OOPSLA '13, pages 391–406, New York, NY, USA, 2013. ACM.

[134] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 41–56, Berlin, Heidelberg, 2010. Springer-Verlag.

[135] Gabriel Antoniu and Philip Hatcher. Remote Object Detection in Cluster-Based Java. In *Proc. 3rd Int. Workshop on Java for Parallel and Distributed Computing (JavaPDC '01*, 2001.

[136] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[137] Barton P. Miller Giridhar Ravipati, Andrew Bernat and Jeffrey K. Hollingsworth. Towards the deconstruction of dyninst. Technical report, UW Madison, June 2007.

[138] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[139] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin:

Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[140] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[141] F. Proch and A. M. Kempf. Numerical analysis of the cambridge stratified flame series using artificial thickened flame les with tabulated premixed flame chemistry. In *Combustion and Flame*, volume 161, pages 2627–2646, 2014.

[142] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 2–2, Oct 2006.

[143] Ignacio Laguna, Todd Gamblin, Bronis R de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H Anh, Martin Schulz, and Barry Rountree. Large scale debugging of parallel tasks with automaded. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. ACM, 2011.

[144] Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R de Supinski, Dong H Ahn, and Martin Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 231–240. IEEE, 2010.

[145] Qi Gao, Feng Qin, and D.K. Panda. Dmtracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12, Nov 2007.

[146] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. Mpi-check: A tool for checking fortran 90 mpi programs. concurrency and computation: Practice and experience, 2003.

[147] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the*

16th European Conference on Research in Computer Security, ESORICS'11, pages 172–189, Berlin, Heidelberg, 2011. Springer-Verlag.

[148] Bernat and Miller. Structured binary editing with a cfg transformation algebra. In *Working Conference on Reverse Engineering (WCRE)*, Kingston, Ontario, Oct 2012.