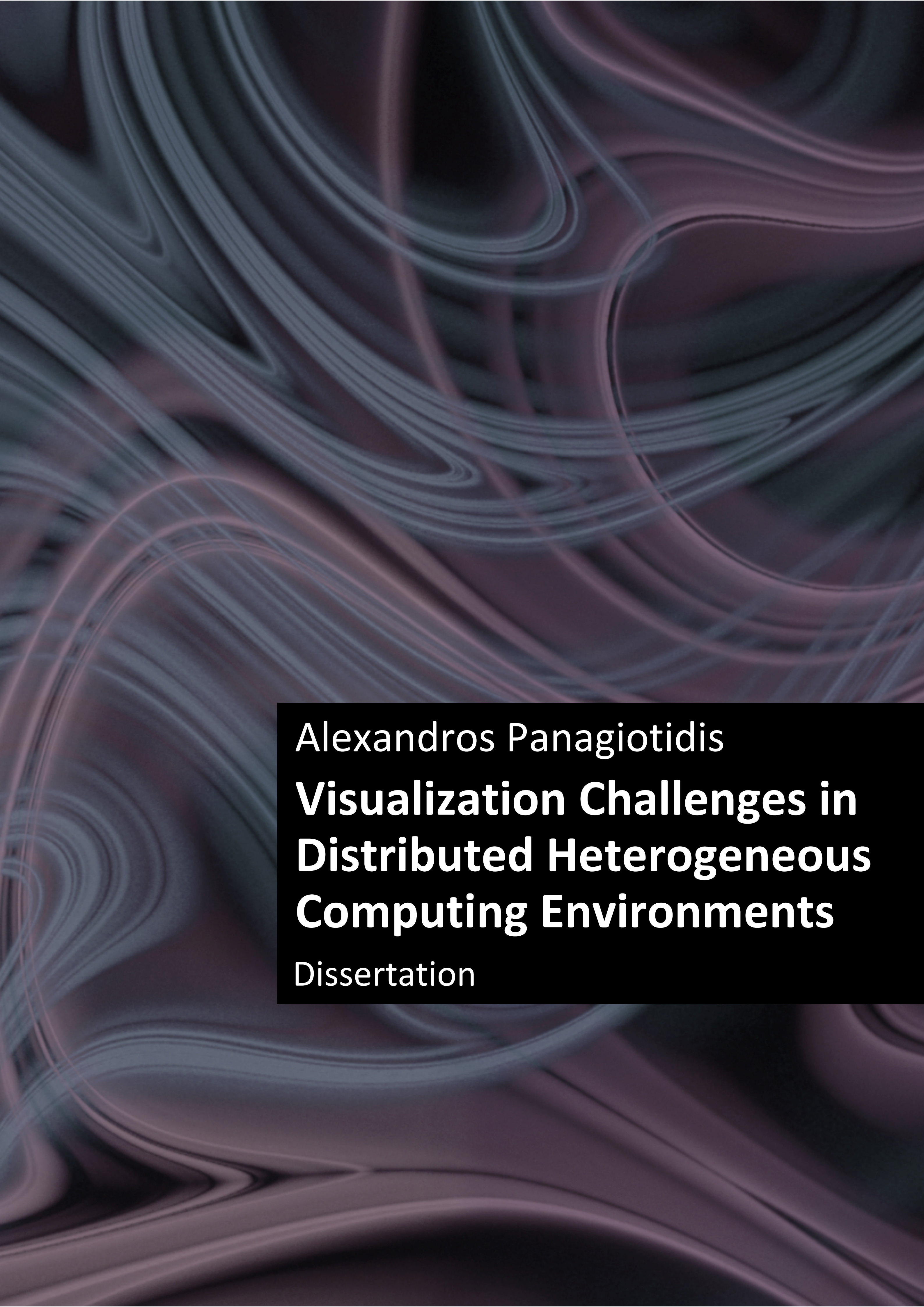


Alexandros Panagiotidis

Visualization Challenges in Distributed Heterogeneous Computing Environments

Dissertation

Visualization Challenges in Distributed Heterogeneous Computing Environments

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Alexandros Panagiotidis

aus Stuttgart

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Dominik Göddeke
Tag der mündlichen Prüfung:	07.12.2015

Visualisierungsinstitut
der Universität Stuttgart

2015

Acknowledgements

This dissertation was a wild ride from start to end and many people have supported me during this chaotic time in one way or another. In the following I want to express my gratitude towards these people in the hope that I did not forget anyone.

First and foremost, I am extremely thankful to Thomas Ertl who made this work possible with funding and by supporting me in word and deed. His influence on me can be best described by Lao Tzu: "A leader is best when people barely know he exists, when his work is done, his aim fulfilled, they will say: we did it ourselves." I believe if it were not for him, I would have never finished. I also want to thank Dominik Göddeke, who agreed to review this dissertation and still examined me at my defense after reading it.

My doctoral journey started with Steffen Koch and Harald Bosch who advised me during my diploma thesis. They inspired me to pursue a doctoral degree, co-authored my first paper, and shared their expertise with me.

I was fortunate enough to share my first office at VIS with Martin Falk. He jump-started my research on programming modern graphics processing units, had good ideas all the time, and created this LaTeX thesis template. Daniel Kauker later joined us and moved to VISUS with me. We worked together closely during the *MCSimVis* project and on several papers. Even though these were challenging times, we both made it, not only due to mutual support.

I am indebted to my co-authors for working with me and enabling me to present my research on many wonderful conferences. My research at VISUS definitely benefited from these bright and patient colleagues: Steffen Frey, Filip Sadlo, Michael Krone, Guido Reina, Michael Burch, and Daniel Weiskopf. I will always remember how we discussed about these crazy ideas that turned into great papers. All of you have given me invaluable advice and helped me immensely in understanding my research better.

Many people lend me their ears and listened to my (quite long-winded) rants, had some good advice, or helped me in some way: Marco Ament, Sven Bachthaler, Fabian Beck, Oliver Fernandes, Lena Gieseke, Sebastian Grottel, Julian Heinrich, Stefan Heßel, Sebastian Koch, Anette Müller, Thomas Müller, Michael Raschke, Katrin Scharnowski, Enrico Taras, Markus Üffinger, Alexander Wender, and Michael Wörner. I also had the opportunity to mentor HiWis and advise students, which always was a mutual learning experience, both technical and personal. My funding was provided by the Federal Ministry of Education and Research of Germany (BMBF) and Deutsche Forschungsgemeinschaft.

Besides working and teaching with the awesome bunch at VIS and VISUS, I also enjoyed hanging out with them at other occasions. Thanks to the *Mensa Boykott* I enjoyed some tasty food, which then also sparked my interest in baking. The *Demo Gang* made the public relation work more enjoyable and feel less like work. My C++ sparring partner Christoph Müller regularly amazed me with his knowledge and skill and how much can be done in a single day. Marcel Hlawatsch, Grzegorz Karch, and Hansjörg Schmauder are formidable pool opponents and good friends that I relied on, especially during the last months of my dissertation up to my defense. I hope we can keep up the random evenings with pool, board games, Whiskey tastings, and other fun activities.

Last, but definitely not least, I want to express my deepest gratitude to my family for always supporting me and giving me strength, hope, and perspective.

Alexandros Panagiotidis

Contents

List of Abbreviations

Acronyms

API	application programming interface
ASIC	application-specific integrated circuit
CPU	central processing unit
CUDA	Compute Unified Device Architecture, developed by NVIDIA
DGEMM	double precision general matrix multiplication
DMP	distributed memory processing
FPGA	field-programmable gate array
FTLE	finite-time Lyapunov exponent
GPGPU	general purpose computing on graphics processing units
GPU	graphics processing unit
HPC	high performance computing
LCS	Lagrangian coherent structure
MPI	message-passing interface, a standardized, portable application programming interface for distributed computing
MSSIM	multiscale structural similarity
NUMA	non-uniform memory access
OpenCL	Open Computing Language, a standard for heterogeneous computing of diverse processing units
OpenGL	Open Graphics Library, a specification for cross-language, cross-platform graphics programming
OpenMP	Open Multi-Processing, an application programming interface for high-level parallelism in C, C++, and Fortran
PB	Protocol Buffers, a platform-neutral framework for serializing structured data, developed by Google
PDB	Protein Data Bank
PPLL	per-pixel linked list
PU	processing unit
RPC	remote procedure call
SIMD	single instruction, multiple data
TDP	thermal design power, maximum amount of heat that a cooling system is required to dissipate

Units

FLOP/s	floating point operations per second
FPS	frames per second

Abstract

Large-scale computing environments are important for many aspects of modern life. They drive scientific research in biology and physics, facilitate industrial rapid prototyping, and provide information relevant to everyday life such as weather forecasts. Their computational power grows steadily to provide faster response times and to satisfy the demand for higher complexity in simulation models as well as more details and higher resolutions in visualizations.

For some years now, the prevailing trend for these large systems is the utilization of additional processors, like graphics processing units. These heterogeneous systems, that employ more than one kind of processor, are becoming increasingly widespread since they provide many benefits, like higher performance or increased energy efficiency. At the same time, they are more challenging and complex to use because the various processing units differ in their architecture and programming model. This heterogeneity is often addressed by abstraction but existing approaches often entail restrictions or are not universally applicable. As these systems also grow in size and complexity, they become more prone to errors and failures. Therefore, developers and users become more interested in resilience besides traditional aspects, like performance and usability. While fault tolerance is well researched in general, it is mostly dismissed in distributed visualization or not adapted to its special requirements. Finally, analysis and tuning of these systems and their software is required to assess their status and to improve their performance. The available tools and methods to capture and evaluate the necessary information are often isolated from the context or not designed for interactive use cases. These problems are amplified in heterogeneous computing environments, since more data is available and required for the analysis. Additionally, real-time feedback is required in distributed visualization to correlate user interactions to performance characteristics and to decide on the validity and correctness of the data and its visualization.

This thesis presents contributions to all of these aspects. Two approaches to abstraction are explored for general purpose computing on graphics processing units and visualization in heterogeneous computing environments. The first approach hides details of different processing units and allows using them in a unified manner. The second approach employs per-pixel linked lists as a generic framework for compositing and simplifying order-independent transparency for distributed visualization. Traditional methods for fault tolerance in high performance computing systems are discussed in the context of distributed visualization. On this basis, strategies for fault-tolerant distributed visualization are derived and organized in a taxonomy. Example implementations of these strategies, their trade-offs, and resulting implications are discussed. For analysis, local graph exploration and tuning of volume visualization are evaluated. Challenges in dense graphs like visual clutter, ambiguity, and inclusion of additional attributes are tackled in node-link diagrams using a lens metaphor as well as supplementary views. An exploratory approach for performance analysis and tuning of parallel volume visualization on a large, high-resolution display is evaluated.

This thesis takes a broader look at the issues of distributed visualization on large displays and heterogeneous computing environments for the first time. While the presented approaches all solve individual challenges and are successfully employed in this context, their joint utility form a solid basis for future research in this young field. In its entirety, this thesis presents building blocks for robust distributed visualization on current and future heterogeneous visualization environments.

German Abstract —Zusammenfassung—

Große Rechenumgebungen sind für viele Aspekte des modernen Lebens wichtig. Sie treiben wissenschaftliche Forschung in Biologie und Physik, ermöglichen die rasche Entwicklung von Prototypen in der Industrie und stellen wichtige Informationen für das tägliche Leben, beispielsweise Wettervorhersagen, bereit. Ihre Rechenleistung steigt stetig, um Resultate schneller zu berechnen und dem Wunsch nach komplexeren Simulationsmodellen sowie höheren Auflösungen in der Visualisierung nachzukommen.

Seit einigen Jahren ist die Nutzung von zusätzlichen Prozessoren, z.B. Grafikprozessoren, der vorherrschende Trend für diese Systeme. Diese heterogenen Systeme, welche mehr als eine Art von Prozessor verwenden, finden zunehmend mehr Verbreitung, da sie viele Vorzüge, wie höhere Leistung oder erhöhte Energieeffizienz, bieten. Gleichzeitig sind diese jedoch aufwendiger und komplexer in der Nutzung, da die verschiedenen Prozessoren sich in Architektur und Programmiermodel unterscheiden. Diese Heterogenität wird oft durch Abstraktion angegangen, aber bisherige Ansätze sind häufig nicht universal anwendbar oder bringen Einschränkungen mit sich. Diese Systeme werden zusätzlich anfälliger für Fehler und Ausfälle, da ihre Größe und Komplexität zunimmt. Entwickler sind daher neben traditionellen Aspekten, wie Leistung und Bedienbarkeit, zunehmend an Widerstandfähigkeit gegenüber Fehlern und Ausfällen interessiert. Obwohl Fehlertoleranz im Allgemeinen gut untersucht ist, wird diese in der verteilten Visualisierung oft ignoriert oder nicht auf die speziellen Umstände dieses Feldes angepasst. Analyse und Optimierung dieser Systeme und ihrer Software ist notwendig, um deren Zustand einzuschätzen und ihre Leistung zu verbessern. Die verfügbaren Werkzeuge und Methoden, um die erforderlichen Informationen zu sammeln und auszuwerten, sind oft vom Kontext entkoppelt oder nicht für interaktive Szenarien ausgelegt. Diese Probleme sind in heterogenen Rechenumgebungen verstärkt, da dort mehr Daten für die Analyse verfügbar und notwendig sind. Für verteilte Visualisierung ist zusätzlich Rückmeldung in Echtzeit notwendig, um Interaktionen der Benutzer mit Leistungscharakteristika zu korrelieren und um die Gültigkeit und Korrektheit der Daten und ihrer Visualisierung zu entscheiden.

Diese Dissertation präsentiert Beiträge für all diese Aspekte. Zunächst werden zwei Ansätze zur Abstraktion im Kontext von generischen Berechnungen auf Grafikprozessoren und Visualisierung in heterogenen Umgebungen untersucht. Der erste Ansatz verbirgt Details verschiedener Prozessoren und ermöglicht deren Nutzung über einheitliche Schnittstellen. Der zweite Ansatz verwendet pro-Pixel verkettete Listen (per-pixel linked lists) zur Kombination von Pixelfarben und zur Vereinfachung von ordnungsunabhängiger Transparenz in verteilter Visualisierung. Übliche Fehlertoleranz-Methoden im Hochleistungsrechnen werden im Kontext der verteilten Visualisierung diskutiert. Auf dieser Grundlage werden Strategien für fehlertolerante verteilte Visualisierung abgeleitet und in einer Taxonomie organisiert. Beispielhafte Umsetzungen dieser Strategien, ihre Kompromisse und Zugeständnisse, und die daraus resultierenden Implikationen werden diskutiert. Zur Analyse werden lokale Exploration von Graphen und die Optimierung von Volumenvisualisierung untersucht. Herausforderungen in dichten Graphen wie visuelle Überladung, Ambiguität und Einbindung zusätzlicher Attribute werden in Knoten-Kanten Diagrammen mit einer Linsenmetapher sowie ergänzenden Ansichten der Daten angegangen. Ein explorativer Ansatz zur Leistungsanalyse und Optimierung paralleler Volumenvisualisierung auf einer großen, hochaufgelösten Anzeige wird untersucht.

Diese Dissertation betrachtet zum ersten Mal Fragen der verteilten Visualisierung auf großen Anzeigen und heterogenen Rechenumgebungen in einem größeren Kontext. Während jeder vorgestellte Ansatz individuelle Herausforderungen löst und erfolgreich in diesem Zusammenhang eingesetzt wurde, bilden alle gemeinsam eine solide Basis für künftige Forschung in diesem jungen Feld. In ihrer Gesamtheit präsentiert diese Dissertation Bausteine für robuste verteilte Visualisierung auf aktuellen und künftigen heterogenen Visualisierungsumgebungen.

Chapter 1

Introduction

The *TOP500* [Strohmaier, 2006] is a listing of the 500 fastest supercomputers with statistics and information about *high performance computing* (HPC) systems. It evolved in roughly 30 years from a simple system count to a detailed ranking of system components, manufacturers, and their performance which is updated twice a year in June and November. Besides the theoretical peak performance, the listings include results of the *LINPACK* [Dongarra et al., 2003] benchmark as a practical measure of the system performance. Even though the *TOP500* is a comprehensive listing, it is not a complete one; some systems are not listed even though they could reach a high ranking, for example *Blue Water* [Bode et al., 2013].

In the last 9 years (see Table 1.2), the *TOP500* experienced an intriguing development. In November 2006, *Tsubame* increased its theoretical peak performance by over 60 % by employing accelerator cards, being the first system in the *TOP500* to do so. The ClearSpeed *X620*, a general-purpose coprocessor, featured 1 GB of ECC memory and was capable of 40 GFLOP/s. Two years later in 2008, *Roadrunner* topped the ranking with IBM's *PowerXCell 8i*, a variant of IBM's Cell broadband engine. In the same year, *Tsubame* was extended with NVIDIA GT 200 *graphics processing units* (GPUs), almost doubling its peak performance. In November 2009, *Tianhe-1* entered the ranking in 5th place with ATI GPUs. From then on, the use of coprocessors increased and they were employed in many more systems. The first NVIDIA Tesla C2050 GPUs were employed in June 2010 in *Nebuale* and *Mole-8.5*. *Tianhe-1A* followed and replaced its ATI

Date	Rank	System	Accelerator	TFLOP/s	kW
2006/11	9	TSUBAME	ClearSpeed CSX620	82.1	n/a
2008/06	1	Roadrunner	PowerXCell 8i	1375.8	2345
2008/11	30	TSUBAME	adds NVIDIA GT 200	161.8	n/a
2009/11	5	Tianhe-1	ATI Radeon HD 4870	1206.2	n/a
2010/06	2	Nebulae	NVIDIA C2050	2984.3	2580
	19	Mole-8.5	NVIDIA C2050	1138.4	n/a
2010/11	1	Tianhe-1A	NVIDIA C2050	4701.0	4040
	4	TSUBAME 2.0	NVIDIA M2050, S1070	2287.6	1398
	22	LOEWE-CSC	AMD Radeon HD 5800	469.7	n/a
2012/06	150	Discovery	Intel MIC	181.0	101
2012/11	1	Titan	NVIDIA K20	27,112.5	8209
	7	Stampede	Intel Xeon Phi	3959.0	n/a
	53	Discover	Intel Xeon Phi 5110P	628.8	216
	58	Endeavor	Intel Xeon Phi	502.1	300
	59	MVS-10P	Intel Xeon Phi	523.8	223
2013/06	1	Tianhe-2	Intel Xeon Phi 31S1P	54,902.4	17,808
2014/11	369	Suiren	PEZY-SC	373.0	n/a
2015/06	160	Shoubu	PEZY-SC	843.0	n/a
	366	Suiren	PEZY-SC	373.0	55
	392	Suiren Blue	PEZY-SC	384.8	n/a

Table 1.2 — Notable entries in the *TOP500* that employed PUs with their theoretical peak performance and energy consumption.

GPUs in November, while *Tsubame-2.0* entered in 4th place with a combination of NVIDIA M2050 and S1070 GPUs. *Tsubame-1* and the *CSX620* left the ranking in June 2012 while the first system using Intel's *Many Integrated Core* architecture (nowadays known as *Xeon Phi*) entered the rankings. More systems followed that year using various PUs from Intel, NVIDIA, and AMD. The *Suiren* promoted this development again in November 2014, entering in 369th place by employing yet another PU, the *PEZY-SC*.

As of June 2015 [TOP500 Supercomputer Sites, 2015], 90 systems (18 %) use accelerators, 52 of which are NVIDIA GPUs and 35 variants of Intel's Xeon Phi. In the same month, Intel acquired Altera, a manufacturer of *field-programmable gate arrays* (FPGAs) and *application-specific integrated circuits* (ASICs). A month later, in July, DARPA announced their investment in Rex Computing's *Neo* architecture, a chip with dramatically improved performance-to-energy ratios [Nicole Hemsoth, 2015]. Finally, there are further PUs like Adapteva's Epiphany or Kalray's MPPA that are not in wide use yet but become increasingly interesting due to their power efficiency, one of the major challenges on the road to exascale [Moreland, 2012].

While PUs were established in HPC, the number of cores increased steadily. In 2006, *BlueGene/L* held the 1st place with 131,072 cores, while *Tianhe-2*, currently 1st, employs a total of 3,120,000 cores. A major challenge in systems of these dimensions are failures of hardware and software. In particular, Schroeder and Gibson [2007] argue that the mean time between failures decreases with increasing system size and the failure rates will grow dramatically in the future. Cappello [2009] even states that "there is a strong risk that parallel applications using all CPUs will not progress any more in parallel computers". He argues that the mean time to interrupt might fall as low as to one hour while the prevalent recovery approach, *checkpoint-restart* methods, requires the same or more time. This is a threat to large-scale systems and applications, as they might never yield a useful result in the worst case.

Such large systems become increasingly complex, both w.r.t. hardware and software. Understanding and optimizing either is an elaborate task. While there are many profound options for both monitoring and analysis (e.g., Munin, Nagios, Vampir), they stem from times in which systems were considerably smaller. As such, they might scale w.r.t. the data they need to process, but fail to address the human, for example by providing responsive interfaces. In addition, their analysis is often decoupled, i.e., infrastructure and applications are analyzed separately, requiring manual steps to correlate them.

Interlude: Graphics APIs

Before computer graphics was supported with hardware acceleration, applications employed software rendering, used VESA BIOS extensions, and accessed the video memory directly. Nowadays, computer graphics relies on GPUs, streaming processors that are designed for data-parallel operations and the *single instruction, multiple data* (SIMD) paradigm, i.e., related graphical primitives are processed in bulk and in parallel. Programming them requires special *application programming interfaces* (APIs) to set up the data streams, transfer constant and dynamic data to the GPU programs, and initiate the data processing by the rendering pipeline.

SGI was leading the efforts for hardware-supported graphics with the *Iris GL* API in the early 1990s due to its immediate mode rendering which simplified programming [Wikipedia, 2015d]. As more 3D graphics hardware vendors entered the market, SGI tried to strengthen their position by turning their API into an open standard — the *Open Graphics Library* (OpenGL). In 1992, the OpenGL *Architecture Review Board* was created to lead and maintain the OpenGL specification and the first version of OpenGL was released. While OpenGL slowly developed, Microsoft created a competing API — Direct3D, part of DirectX — and released it in 1995. Over time, both DirectX and OpenGL evolved into large and complex APIs that provide access to a wide range of GPUs with different capabilities. In particular, the complexity of the implementations (i.e., the driver) became so troublesome, that NVIDIA began investigating how to approach zero driver overhead [Everitt et al., 2014]. Shortly before that, AMD announced *Mantle*, a low-level graphics API to alleviate bottlenecks by transferring complex tasks (e.g., validation, synchronization) from the driver to the application and by allowing greater control over the GPU. Both *Mantle* and NVIDIA's initiative prompted action by the Khronos Group (which maintains OpenGL since 2006) to announce their plans for a next-generation graphics API in 2014. The Khronos Group revealed this new API at SIGGRAPH 2014 under the name *Vulkan*, but did not present any specific information at that time. In March 2015, it was revealed that the foundation of *Vulkan* was *Mantle* [Andersson et al., 2015], only one day after AMD announced its end [Koduri, 2015]. During all of this, Microsoft worked on DirectX 12 with the same goals: reduced overhead, fine-grained control, and scaling across *central processing unit* (CPU) threads. While the merits of these new APIs have been unclear for some time, Imagination Technologies just recently presented a demo at SIGGRAPH 2015 with very promising results [Smith, 2015].

1.1 Motivation and Research Questions

The paradigm shift in HPC towards *general purpose computing on graphics processing units* (GPGPU) and heterogeneous systems in general happened in less than 10 years. During that period, many architectures and APIs emerged and evolved but some have also vanished. Great resources and efforts have been and are continually invested into using PUs and maximizing their performance and efficiency. However, what will happen when the next paradigm shift occurs? Will developers have to refactor their simulation codes and post-processing tools again? How can developers even determine whether a new technology is worthwhile? What if it changes rapidly during initial revisions or requires big changes to applications, potentially dismissing many months and years of development effort? While waiting until it stabilized and prevailed is feasible, this relinquishes any lead to competitors. These inherent risks and trade-offs of emerging technologies can have many consequences in science and industry, even more so when no universal solution is foreseeable [Moreland, 2012].

All of these problems and questions apply to distributed visualization but with additional challenges. For one, visualization is most useful when interactive which requires fast techniques but also steering by humans. Resolving failures in these systems has different semantics than in non-interactive systems. For example, it might even be favorable to interact with partial or even erroneous results (which are traditionally discarded) in order to interpret or validate data sources like simulations. Analysis of large data sets often results in large images for unbiased and clear views on the visual mappings. This necessitates large displays — combinations of multiple monitors and projectors — even with increasing resolutions on commodity monitors. These aspects add several dimensions of complexity due to technical aspects and the 'human in the loop'.

This thesis discusses the aforementioned challenges in the context of distributed visualization in heterogeneous environments, in particular:

1. Which approaches can visualization systems employ to support evaluating and switching between programming models and visualization techniques?
2. How must resilience in visualization systems improve to support challenges of large displays and increasing system sizes?
3. How can systems for interactive visualization on large displays be analyzed and optimized?

1.2 Contribution and Outline

The major findings and contributions of this thesis comprise methods and approaches to simplify and generalize development of visualization systems for large displays and heterogeneous environments. Novel approaches and techniques are presented which employ, combine, and extend existing concepts and paradigms in new ways. Challenges of interactive visualization on large displays in combination with a heterogeneous computing environment — beyond mapping and rendering techniques — are discussed and investigated for the first time.

This thesis consists of four main parts. Part I promotes abstraction as an important concept for distributed visualization and development in heterogeneous environments. Part II presents methods for fault tolerance in the context of distributed visualization. Part III advances analysis in cluttered visualizations as well as tuning of interactive visualizations on high-resolution displays. Part IV concludes this thesis with answers to the research questions of Section 1.1 and an outlook for future work.

This work is based on several publications and close work with many colleagues during several projects in the last six years:

A. Panagiotidis, H. Bosch, S. Koch, and T. Ertl. EdgeAnalyzer: Exploratory Analysis through Advanced Edge Interaction. In *44th Hawaii International Conference on System Sciences*, pages 1–10. IEEE Computer Society, 2011a.

A. Panagiotidis, D. Kauker, S. Frey, and T. Ertl. DIANA: A Device Abstraction Framework for Parallel Computations. In P. Iványi and B. H. V. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press, 2011b, paper 20.

A. Panagiotidis, D. Kauker, F. Sadlo, and T. Ertl. Distributed Computation and Large-Scale Visualization in Heterogeneous Compute Environments. In *11th International Symposium on Parallel and Distributed Computing*, pages 87–94, 2012.

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Rendering Molecular Surfaces using Order-Independent Transparency. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 33–40. The Eurographics Association, 2013b.

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Evaluation of per-pixel linked lists for distributed rendering and comparative analysis. *Computing and Visualization in Science*, 15(3):111–121, 2013a.

A. Panagiotidis, G. Reina, and T. Ertl. Strategies for Fault-Tolerant Distributed Visualization. In *2014 IEEE Pacific Visualization Symposium*, pages 286–290, 2014b.

A. Panagiotidis, M. Burch, O. Deussen, D. Weiskopf, and T. Ertl. Graph Exploration by Multiple Linked Metric Views. In *Proceedings of the 18th International Conference on Information Visualisation (IV)*, pages 19–26. IEEE, 2014a.

A. Panagiotidis, S. Frey, and T. Ertl. Exploratory Performance Analysis and Tuning of Parallel Interactive Volume Visualization on Large Displays. In *Eurographics Conference on Visualization - Short Papers*, pages 13–17. The Eurographics Association, 2015a.

I have also worked on partial link drawings to further reduce visual clutter in node-link diagrams [Burch et al., 2014] and on consistently accelerating computation of node-link diagram layouts and their visualization on modern GPUs employing features such as tessellation and compute shaders [Panagiotidis et al., 2015b].

The following describes the structure of this thesis, summarizes novel contributions, and delineates them in joint research.

Chapter 2 introduces the foundations and terminology for both visualization and heterogeneous systems. This chapter also introduces the target platform for the concepts in this thesis — the VVand— the VISUS visualization cluster and powerwall. It is based on parts of all of my aforementioned publications.

Chapter 3 presents DIANA, an abstraction layer for PUs, their memory, and operations [Panagiotidis et al., 2011b]. DIANA was developed together with Daniel Kauker during the *MCSimVis* project as middleware to access GPUs in an industrial FEM simulation in a portable and extendable manner. I developed the basic concepts — lookup and access of PUs through a database and unified invocation of all PUs operations —, integration of CUDA and OpenCL, and most of the plugins to provide functionality like linear algebra or compression, while Daniel worked on the application integration and remote invocation. Together we created a distributed flow visualization [Panagiotidis et al., 2012] that utilized DIANA. After *MCSimVis* finished, I redesigned and reimplemented DIANA to include lessons learned during the project and to incorporate new concepts, such as using *Protocol Buffers* (PBs) as an abstraction for commands. This thesis refers to this new version of DIANA as paradigm for developing for current and future computing environments and presents new and improved results.

Chapter 4 introduces *per-pixel linked lists* (PPLLs) and their implementation on modern GPUs. Daniel Kauker and I worked on this technique during the *MCSimVis* project for order-independent transparency and distributed rendering. I worked on optimizations and different memory layouts for this technique, while Daniel further refined and extended it for distributed rendering and comparative analysis [Kauker et al., 2013a] and rendering of semi-transparent molecular surfaces [Kauker et al., 2013b], ultimately extending it to per-voxel linked lists and further rendering techniques [Kauker, 2015]. This thesis presents the basic technique as method for generic compositing on the VVand, which was implemented independently by myself.

Chapter 5 presents extensions to an existing taxonomy of fault tolerance methods in HPC for distributed visualization [Panagiotidis et al., 2014b]. The traditional approaches of this taxonomy are explained and their application to visualization in general and distributed visualization on large, high-resolution displays in particular, is discussed. Based on this, new methods and strategies are proposed to increase resilience of such visualization systems and applications. This work was created during the *FeToL* project based on guidance and discussion with Guido Reina. This thesis goes beyond the publication and discusses example implementations of the proposed strategies.

Chapter 6 presents two approaches to local exploration of dense and cluttered graphs. *EdgeAnalyzer* (see Section 6.1) utilizes a lens metaphor for exploration of node-link diagrams with bundled edges. It is based on an idea of Steffen Koch [Koch et al., 2009] and Harald Bosch which I investigated in my diploma thesis [Panagiotidis, 2009]. My addition of multiple, hierarchical lenses and further improvements were later published [Panagiotidis et al., 2011a] and are discussed in this thesis. *Graph Metric Views* (see Section 6.2) show additional information besides a node-link diagram in the form of histograms. Michael Burch suggested this idea to me, which I then implemented and evaluated until it was refined enough for publication [Panagiotidis et al., 2014a].

Chapter 7 presents an approach to exploratory performance analysis and tuning of an interactive volume visualization. I got the idea for this work during the *FeToL* project when I wondered how to debug and analyze visualization systems on the VVand. After *FeToL* finished, I integrated my GPU-accelerated parallel coordinates plot into Steffen Frey's volume renderer. Steffen provided metrics from the ray casting kernel, while I collected metrics from *NVAPI* (NVIDIA's management API) and added metric gathering and aggregation. Together we created the case studies and published the results [Panagiotidis et al., 2015a].

Chapter 8 discusses the joint utility of the presented techniques and how they answer the research questions (see Section 1.1). An outlook to open challenges and future work concludes this thesis.

Chapter 2

Fundamentals

This chapter introduces the foundations of this thesis. Section 2.1 describes basics of visualization, rendering on *graphics processing units* (GPUs), distributed visualization, and large displays. Section 2.2 defines heterogeneous systems, *processing units* (PUs), *general purpose computing on graphics processing units* (GPGPU), and offloading. Section 2.3 presents the VISUS powerwall and cluster. Section 2.4 briefly describes box plots, which are used several times in this thesis to present results.

2.1 Visualization

Haber and McNabb [1990] define visualization as "a series of transformations that convert raw simulation data into a displayable image". This definition alludes to the notion that gaining insight from data requires images and human perception due to the (increasing) size and complexity of data sets. It is a data-centric and inter-disciplinary science comprised of technical and psychological aspects.

2.1.1 The Visualization Pipeline

While Haber and McNabb [1990] explicitly stated simulations in their definition, it was since extended to account for the multitude and diversity of today's data sources. This process of turning data into images has been coined as *visualization pipeline* (see Figure 2.1), for example as described by Weiskopf [2007].

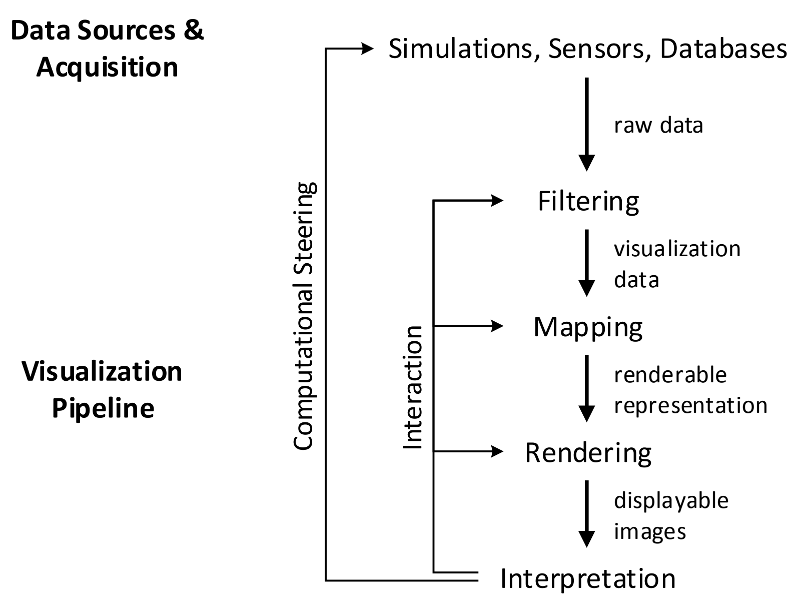


Figure 2.1 — The visualization pipeline is a process to transform data from various sources into images for analysis and interpretation by humans.

The input to the visualization pipeline is *raw data* which is acquired from sources like numerical simulations, sensor measurements, and databases. This data is then transformed during *filtering* into *visualization data* by operations like resampling, smoothing, denoising, segmentation, classification, and selection. *Mapping* then creates a *renderable representation* consisting of graphical primitives with attributes like geometry, color, opacity, and surface texture. Finally, *rendering* creates *displayable images* used for analysis by humans.

It is often unclear what the important and relevant aspects and parts of data sets are and as such visualization has to be an interactive process. Consequently, all stages of the visualization pipeline are subject to many parameters, for example selecting which parts of the data to show or modifying how it is shown. Typical actions include panning, zooming, and picking/selecting in the visualization itself to explore the data set.

Visualization is mostly used in post-processing for analysis of data that is stored persistently after it was collected or generated, i.e., the data acquisition is decoupled from the visualization. In contrast, *computational steering* [Mulder et al., 1999] denotes the tight coupling of visualization with data generation. Parameters from both visualization and simulation can be modified interactively to incorporate insights gained from the visualization in the data generation.

In visualization, the raw input data is often referred to as *object space* or physical space, as it describes the objects or physical extents of the data. The visualization pipeline transforms data from the object space into the *image space* or screen space. This denotes the resulting images (or parts thereof) that are shown on displays.

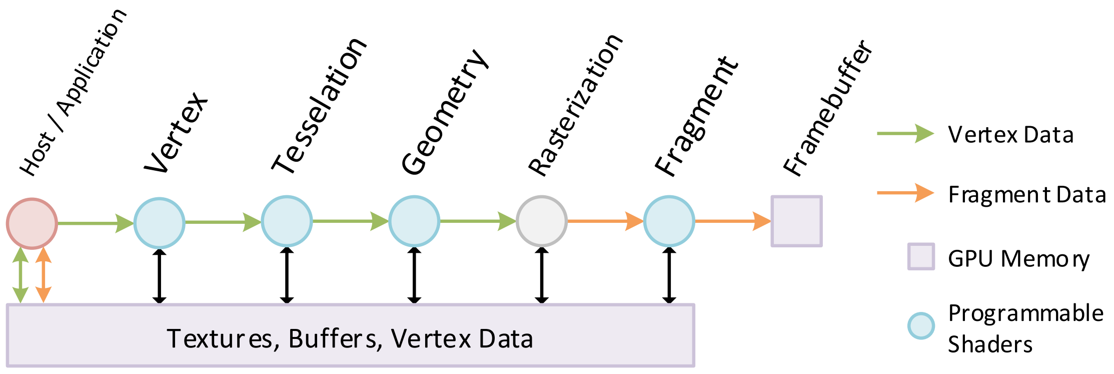


Figure 2.2 — Simplified OpenGL rendering pipeline.

2.1.2 The Rendering Pipeline

An essential part of visualization is rendering, i.e., the generation of images from graphical primitives like triangle meshes. The *rendering pipeline* (see Figure 2.2) describes this process of projecting a 3D scene onto a 2D image. An optimized implementation of the pipeline for GPUs is provided by OpenGL [Sellers et al., 2013], described in the following. The OpenGL pipeline was only configurable to some extent until programmable shaders were introduced in 2004 with OpenGL 2.0. DirectX 8.0 defined programmable shader stages already in 2000 and GPUs supported them since NVIDIA's NV20 architecture, first implemented in their GeForce 3, in 2001. Nowadays almost all parts[1] are fully programmable due to the unified shader model.

The elements of a 3D scene are composed of primitives such as points, lines, or triangles. Vertices with attributes (e.g., colors, normals, texture coordinates, etc.) and their connectivity define these primitives. The vertex data is transferred to the GPU along with textures — buffers that can be accessed with spatial coordinates in 1D, 2D, and 3D. The texture units in modern GPUs further allow for hardware-accelerated interpolation between texels and voxels (for 1D/2D and 3D textures, respectively). The host initiates the rendering with a draw call which triggers the processing of the scene by the pipeline.

Vertices are processed independently without any connectivity information in the *vertex shader*. There, they are transformed from their local coordinate system (the object space) into world space using affine transformations (e.g., the model and view transformation) and homogeneous coordinates. The attributes of the vertices can also be modified and passed to subsequent stages. The output of the vertex shader can be optionally passed to the *tesselation shaders*,

[1]Rasterization is the most prominent part of the rendering pipeline that is not yet programmable or even accessible via public APIs.

to subdivide existing geometry, and the *geometry shader*, to create, modify, or discard geometry. At this point, perspective projection transforms the vertices into clip space, mapping them into the unit cube $[-1,1]^3$. The parts of the geometry outside of the cube are clipped while the remaining parts are transformed into normalized device coordinates by perspective division. *Rasterization* breaks the remaining parts of each primitive into discrete elements — *fragments* — and interpolates vertex attributes and depth w.r.t. the fragment position. The viewport transformation maps each fragments' position into window coordinates. Then, each fragment is passed to the *fragment shader* which can perform final calculations like sampling textures, per-pixel lighting, or discarding individual fragments based on their depth or opacity. Fragment shaders can write multiple outputs and are not limited to writing only color or depth. Fragments that pass the fragment shader are subject to several final tests such as pixel ownership, scissor, stencil, or depth test. Typically, *z-buffering* is used, i.e., the depth test discards fragments whose depth is greater than the one already in the framebuffer. Finally, *blending* combines the color of the fragment with the value stored in the framebuffer at the fragment position. If it is disabled, the previous value is overwritten; otherwise, a configurable predefined blending equation is employed. For this, the *over-operator* [Porter and Duff, 1984] is commonly used for alpha blending of semi-transparent objects 1 and 2 front-to-back:

$$
\begin{aligned}
C_{out} &= \alpha_1 C_1 + \alpha_2 C_2 (1 - \alpha_1) \\
\alpha_{out} &= \alpha_1 + \alpha_2 (1 - \alpha_1)
\end{aligned}
$$

or back-to-front:

$$C_{out} = \alpha_1 C_1 + (1 - \alpha_1) C_2$$

where C_i and α_i denote the color and the opacity, respectively. The over-operator is not commutative, i.e., the order matters in which the fragments are blended.

2.1.3 Distributed Visualization

Section 2.1.1 and 2.1.2 described visualization as an abstract process which can be implemented on specialized hardware, respectively. If not specified, it is assumed that one process on a single node is used, with all relevant data available locally and images shown on its attached displays. On the one hand, this allows for simplifications, such as synchronization only across threads (if any), low latency, or exclusive use of resources. On the other hand, there are many restrictions with regard to frame rates, data sizes, and display resolutions.

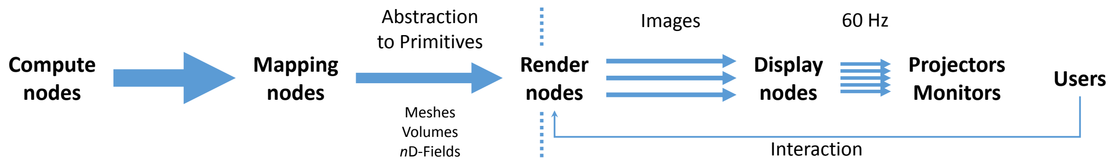


Figure 2.3 — Conceptual flow of data in distributed simulation and visualization. The size of the arrows correspond to the amount of data that is passed, while the number of arrows indicates the frequency of the transfer. The actual data sizes for the left of the render nodes depend on simulation parameters and those to the right depend on the resolution of the viewport.

These are inherent because there are limits to how much data a single node can store, how many FLOP/s are possible, how many displays (each with finite resolution) can be attached, etc. Even though advances in technology relax these limitations, they need to be overcome fundamentally for further scalability and resilience. Consequently, the visualization pipeline needs to be distributed to multiple processes, nodes, and displays.

Distributed visualization is a large and complex field, so only a brief introduction to the terms and concepts can be given here; Bartz and Silva [2001], Meligy [2008], and Bethel et al. [2012] provide thorough overviews and discuss the field in greater detail.

Figure 2.3 shows the generic flow of data between the different types of nodes for distributed visualization. *Compute nodes* provide data by various means, for example calculated by simulations or loaded from persistent storage. Generally, these nodes are considered black boxes for the purpose of visualization, since their operation and details should be independent from the analysis of the data they produce. For computational steering, the visualization may pass parameters to the compute nodes, but is not further concerned with their internals. The raw data is filtered and then passed to *mapping nodes* which create the graphical primitives, for example triangle meshes or volumes. *Render nodes* create images which are composited by *display nodes* and shown on their attached output devices, for example projectors or monitors. Interaction with the system is possible through dedicated *head nodes*, tracking, or gestures.

Partitioning

For distributed visualization, the object space and the image space can be partitioned for rendering by multiple processes and nodes. Molnar et al. [1994] discuss this as sorting problem and define three classes:

Sort-first (send-data) is used to distribute parts of the object space for visualization algorithms, i.e., each node or process handles a part of the physical domain. This scales well with data or model size but might require communication between nodes to exchange boundary information. Compositing becomes also more complex since many nodes might contribute to a pixel in the final image.

Sort-middle (send-geometry) processes the input data and distributes the transformed graphical primitives for rasterization to the respective nodes. It is not well supported in the current architecture of GPUs since vertex processing and rasterization is tightly coupled and reading back data from a GPU is usually avoided whenever possible for performance reasons.

Sort-last (send-images) is used to partition the image space into so-called *tiles*. Different nodes then process each tile and send the results to display nodes. While this scales well with image resolution, it requires that each node has access to all data and is thus only reasonable for small data sizes.

The partitioning is often dictated by the data source, for example when simulations already decompose their physical domain to different nodes, i.e., the data is already partitioned in object space. In these cases it is still possible to partition the image space, i.e., let multiple render nodes create parts of the projected image of that object space.

Compositing

Section 2.1.2 briefly mentioned blending, the combination of a fragments' color with the one already stored in the framebuffer. For distributed visualization, this operation is more challenging because multiple nodes might contribute to the final color of a pixel. In a completely opaque scene, *depth compositing* is sufficient, i.e., only the closest fragment is used. It is not feasible to composite a correct image this way with semi-transparent objects in the scene. Maule et al. [2011] discuss this as a sorting problem:

Geometry-sorting methods sort whole objects (i.e., parts of a scene) or primitives w.r.t. their depth before rendering. This is the standard approach for rendering semi-transparent objects and is often realized using the Painter's algorithm [Foley et al., 1990] but provides incorrect results with interpenetrating geometry.

Fragment-sorting collects all fragments of the scene and sorts them by depth before blending. These methods store the order of fragments either explicitly in buffers or implicitly in depth layers.

Hybrid-sorting approaches combine geometry-sorting with fragment-sorting. They sort the geometry approximately and correct the fragment order during rendering or employ occlusion queries to obtain a correct back-to-front ordering.

Depth-sorting-independent methods ignore depth and blend fragments in incoming order (i.e., unsorted). They are based on weighted averages and sums and are not suitable for fragments with distinct colors and transparencies.

Probabilistic techniques employ screen-door transparency [Mulder et al., 1998] by using stochastic and probability to determine the fragment visibility. Consequently, they introduce noise, which can be attenuated through the number of samples.

Maule et al. [2011] conclude that A-buffer [Carpenter, 1984] and linked-list [Yang et al., 2010] approaches are the fastest methods for rendering with correct handling of transparency but have high memory requirements because they store all fragments.

2.1.4 Large Displays

Visualizing large data sets often results in large images which exceed the resolution of traditional monitors. Even with the advent of 4K monitors, the same reasoning as for distributed visualization (see Section 2.1.3) applies, namely that scaling beyond technical limitations requires distribution. In this context, this means to show the image on a bigger physical space, i.e., on multiple monitors or projectors.

Large displays range from megapixels to gigapixels [Papadopoulos et al., 2015]. They consist of multiple monitors or projectors [Ni et al., 2006] that show parts of the full image which can be computed on the attached display nodes or by utilizing a render cluster (see Figure 2.3).

Common issues of large displays include color calibration and frame synchronization. For monitor arrays, the bezels are problematic as they disrupt both perception and interaction. To prevent hard edges, projector arrays often employ blending areas which require geometric calibration as well as intensity correction in these areas.

2.1.5 Frameworks

There are several frameworks for distributed visualization. *WireGL* [Humphreys et al., 2001] implements a sort-first rendering pipeline by streaming OpenGL commands to multiple 'pipeservers' that render parts of the geometry. *Chromium* [Humphreys et al., 2002] extends that approach by transforming the *application programming interface* (API) stream in other ways and thus enables sort-last rendering. *Equalizer* [Eilemann et al., 2009] provides an API for scalable parallel rendering. *ParaView* [Ahrens et al., 2005] is a full-fledged application to configure the visualization pipeline during run time, distribute it to multiple nodes, and execute it in parallel, which utilizes *IceT* [Moreland et al., 2011], a "high-performance sort-last parallel rendering library".

MegaMol [Grottel et al., 2015] is a framework for rapid prototyping of visualization applications that supports sort-last rendering and large displays. It was initially designed for the visualization of large number of molecules in the context of the SFB 716[2]. *MegaMol* is a modular approach to realize the visualization pipeline by combining several modules that load, filter, map, and render data sets. It is used several times as display front-end for the approaches in this thesis.

2.2 Heterogeneous Systems

Heterogeneous systems use more than one kind of PU, "allowing each to perform the tasks to which it is best suited" [Shan, 2006]. This is not a new trend, as coprocessors have been used in computing for some time, for example floating point units, AGEIA's PhysX cards, or GPUs.

In general, hardware vendors supply their own, distinct APIs that are required to utilize their PUs. Even though there are standardized interfaces for some PUs, there are also vendor- and hardware-specific features that can only be accessed using the vendor API. This is complex and time-consuming for a variety of reasons, like detecting available hardware, differing data structures and calling conventions, incompatibilities between hardware, and different toolchains. On the one hand, even when developers choose to specialize their applications by supporting only one API, they face problems like differing hardware capabilities (e.g., double precision support) and evolving APIs. On the other hand, it is sensible to support different APIs and PUs, because of changes in hardware, for example better performance, higher energy efficiency, or lower costs. Finally, software that is already in long-term production is not easily adapted to new

[2]Sonderforschungsbereich 716 — "Dynamic simulation of systems with large particle numbers", see http://www.sfb716.uni-stuttgart.de/.

hardware and the new programming models. Production code needs to be changed to use new APIs, computations need to be adapted for the specific architectures, interfaces are needed that support these additional code paths, and everything needs to be re-tested. The following is a brief introduction to this field; a general overview and survey about heterogeneous computing is provided by Brodtkorb et al. [2010].

2.2.1 Processing Units

While there are many different PUs, *central processing units* (CPUs) and GPUs are the most relevant since their combined use offers high flexibility and efficiency. CPUs have multiple cores (in the order of dozens), several layers of big caches, good branch prediction, and direct access to the main memory (up to hundreds of GB). GPUs have many cores (in the order of several thousands) arranged in blocks of multiprocessors and separate memory (up to 32 GB), and smaller caches than CPUs. They operate using the *single instruction, multiple data* (SIMD) paradigm and in lock-step, i.e., the multiprocessors execute the same instructions on different data. Even with consumer-grade hardware it is possible to achieve non-negligible speedups. Nowadays, Intel and AMD integrate GPUs with their CPUs, while NVIDIA offers the *Tegra*, a system-on-a-chip solution that combines their GPUs with an ARM CPU.

Intel worked for some time on a many core architecture named *Larrabee* which was eventually canceled and lead to the development of the *Xeon Phi*, now used in some supercomputers (see Chapter 1). In recent years, *field-programmable gate arrays* (FPGAs) gained some popularity in the *high performance computing* (HPC) field as they can offer better energy efficiency for some computations and because they become easier to program, for example using OpenCL on Adapteva's Parallella boards. Another interesting trend is the use of *application-specific integrated circuits* (ASICs) for very specific computations, for example computation of hashes for Bitcoin mining, where they outperform GPUs by several orders of magnitude (see Bitcoin Wiki [2015a] and Bitcoin Wiki [2015b]). As discussed in Chapter 1, new PUs are already in use or in development, for example the *PEZY-SC* or the *Rex Neo*.

2.2.2 Parallelization and Communication

Using multiple PUs requires some form of concurrency so the host, that is coordinating them, is not blocked and might even be used for computations itself. Traditionally, parallel programming employs threads, for example using POSIX, the Windows API, or OpenMP [Chapman et al., 2007]. This leads to some performance gains on individual nodes since modern CPUs feature multiple cores that can execute several threads in parallel. Parallelization beyond node boundaries requires some sort of communication between nodes, for example using the *message-passing interface* (MPI) or sockets.

Non-trivial data structures cannot be transferred directly between processes, for example due to pointers or byte order. Furthermore, the semantics of raw data may not be clear and changes in the data order might lead to incompatibilities. These issues are overcome with serialization [Wikipedia, 2015e], which converts data structures into byte streams by means of deep copies (i.e., following pointers) and encoding. The receiver of the data can then reverse the serialization, i.e., deserialize the byte stream into corresponding data structures.

Since serialization and communication are both necessary for *distributed memory processing* (DMP), there are many approaches that incorporate both, for example *remote procedure calls* (RPC) [Birrell and Nelson, 1984]. RPC enables clients to execute methods in different address spaces without explicit knowledge of the details. Method calls and their data are automatically serialized (also called marshalling in this context) and then transferred to a peer node which deserializes (unmarshalling) the information and executes the method. Upon completion, the results are transferred back to the caller, for whom all of this happened transparently.

2.2.3 Protocol Buffers

Serialization frameworks often require code annotations to encode data structures and their members properly which increases the development effort, especially for programming languages without reflection capabilities (e.g., C++). The information about the serialization format also has to be duplicated in all applications that will receive the serialized data, for example when applications written in different programming languages interact with each other. This can lead to errors when one side of the data transfer changes the format, for example by adding new items or changing data types.

Google *Protocol Buffers* (PBs)[3] is a framework to address these problems. It is a "language- and platform-neutral mechanism for serializing structured data" that offers several advantages: (a) a well-defined description for extensible data structures, (b) an efficient binary serialization format, (c) message passing as inherent paradigm, and (d) introspection and reflection. Data structures are defined as *messages* in a domain-specific language which is compiled into host code for a programming language (see Appendix B.4 for an example). Google's official compiler *protoc* creates C++, Java, and Python code, but there are many

[3]https://developers.google.com/protocol-buffers/

third party compilers for various languages like C#, Lua, or JavaScript. Each compiler creates serialization code that follows the specification of Google's binary format for PBs. Developers can instantiate these messages, fill them with data (see Appendix B.2), and serialize and deserialize them. For C++, *protoc* also creates code to introspect and reflect messages for dynamic traversal and access of their members during run time. As such, PBs are a powerful tool to define data structures which can be exchanged between any process through the PB binary format.

2.2.4 General Purpose Computing on GPUs

The following is a short introduction, history, and overview of *general purpose computing on graphics processing units* (GPGPU), i.e., using GPUs for arbitrary computations. Further details and in-depth overviews of GPGPU and the programming of GPUs are discussed by Mark J. Harris [2003], Owens et al. [2008], Blythe [2008], Göddeke [2010], and Brodtkorb et al. [2013].

GPUs were always used as coprocessors for computer graphics, traditionally a compute-intensive field that greatly benefits from parallelization. They were already used for non-graphical computations (e.g., planing of robot movement [Lengyel et al., 1990]) even before they became programmable through shaders, which required creative use of the configurable rendering pipeline (e.g., blending operations, stencil and accumulation buffers). As shaders became available, general computations could be performed on GPUs by rendering geometry (e.g., a quad) with the same extents as the computation domain and exploiting the fragment shader for the parallel processing of each element. At that time, only textures and the framebuffer could be used for input and output.

As GPGPU became more popular, vendors invested in this paradigm and provided APIs to access their hardware directly without graphics APIs. *CTM* (close-to-metal) [Peercy et al., 2006] was a very low-level API to program ATI GPUs. After AMD acquired ATI, CTM evolved into the *Stream SDK* until they finally switched to OpenCL in 2011. Meanwhile, NVIDIA developed the *Compute Unified Device Architecture* (CUDA) which was heavily marketed and rapidly adopted in both research and industry. Consequently, even though CUDA is only available for NVIDIA GPUs, it gained a wide reach and is used in many systems.

CUDA and OpenCL work in the same way, i.e., they expose methods for explicit memory management of various buffer types and a convenient way to start computations. Both APIs realize the SIMD paradigm using *kernels* that are defined in CUDA C or OpenCL C which are subsets of C and C++. For each element in the compute domain, a kernel is run in a separate thread on the PU, similar to how a fragment shader is run for each generated fragment during rasterization (see Section 2.1.2). The compute domain is defined as 3D grid which can be further refined, i.e., each grid cell consists of 3D thread blocks. Groups of these threads are executed in unspecified order on available multiprocessors of the PU.

CUDA and OpenCL are designed to work on a wide range of PUs. Naturally, these differ in their capabilities as technology advances, for example early GPUs did not support double precision computations and only few concurrent threads (i.e., small compute grids). This diversity across PUs complicates their use since application developers need to consider these restrictions, at least by checking for them and supplying proper error messages. Otherwise, an application may behave in erratic ways, which was especially true in earlier versions of the APIs when their error messages were misleading[4]. In general, their portability only applies to basic use, i.e., they are not performance portable across comparable architectures even within the same API.

2.2.5 Offloading

Programming different PUs is elaborate: computations have to be formulated accordingly (e.g., as shaders or kernels) which might necessitate special toolchains (e.g., 'nvcc' for CUDA), data has to be transferred from the host, and computations have to be invoked. The application also needs to coordinate the host with the PU, for example with barriers in order to further process results from that PU on other PUs. Over time, this process became easier and many tools and frameworks emerged to help with this.

Similar to OpenMP, there are some approaches to access PUs through compiler directives (e.g., OpenACC, AMP++) by injecting necessary API calls and generating kernel code at compile-time. There are also many high-level frameworks (e.g., GPULib [Messmer et al., 2008], CuPP [Breitbart, 2009]) that provide convenient access or simplified memory management [Sundaram et al., 2009]. Furthermore, there are many works that specialize on certain domains (e.g., FEM [Göddeke et al., 2005], sorting [Satish et al., 2009], linear algebra [Dongarra et al., 2014]). Frameworks like GPU++ [Jansen, 2007] and HONEI [van Dyk et al., 2009] are more on a middle ground, as they allow for convenient use of PUs as well as low-level access. For distributed computations there are several frameworks that either relay calls (e.g., Duato et al. [2010], Barak et al. [2010], Grasso et al. [2013]) or distribute tasks to remote nodes automatically (e.g., Strengert et al. [2008], Müller et al. [2009], Frey and Ertl [2010], Kofler et al. [2013]).

[4]Early versions of CUDA were notorious for declaring most errors as 'unspecified launch error'.

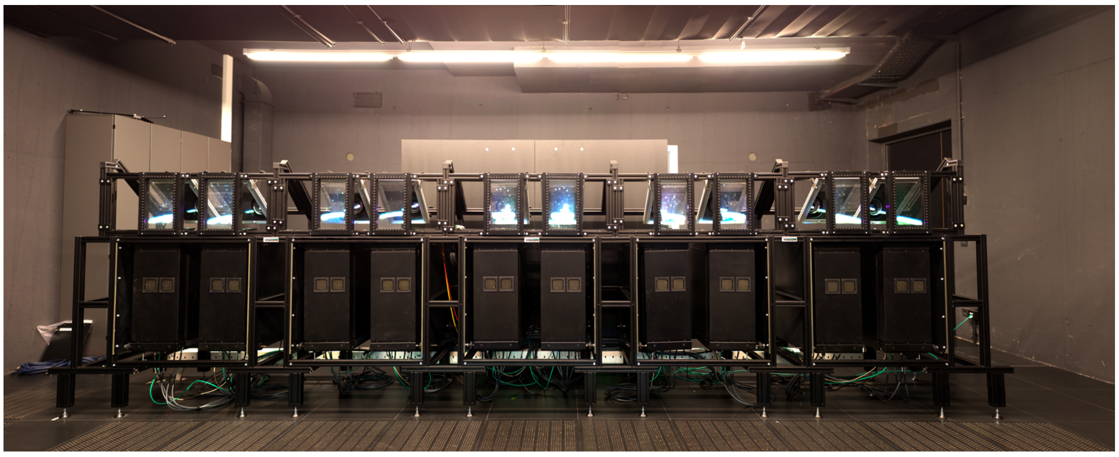


Figure 2.4 — The powerwall of the VVand consists of two groups with five 4K projectors each, forming a display with an effective resolution of 10,800 × 4096 pixels on a 6 m × 2.2 m glass wall.

2.3 VVand—The VISUS Powerwall and Cluster

Most of the following approaches presented in this thesis are designed for and implemented on the VVand, the VISUS powerwall and cluster [Müller et al., 2013]. It is a two-tier system specifically designed for interactive, large-scale visualization.

Powerwall

The display of the VVand is a rear projection system consisting of 10 JVC 4K LCoS projectors with a resolution of 4096 × 2400 pixels each, organized in two groups of five projectors (see Figure 2.4). Each group fills the 6 m × 2.2 m physical display space of a glass wall entirely and together they are used for stereoscopic visualization, i.e., each group displays the images for one eye.

The projectors are arranged vertically with a 300 pixel wide blending area for smooth transitions and to counter hard lines between them. This setup yields an effective resolution of 10,800 × 4096 pixels (≈44 megapixel per eye) with a pixel size of 0.56 mm. Each projector is attached to one display node (see Appendix A.2 for their specifications).

Compute and Graphics Cluster

Small data sets and simple scenes can be rendered directly on the display nodes. For bigger and more complex data sets, a GPU cluster of 64 nodes is available (see Appendix A.3 for their specifications).

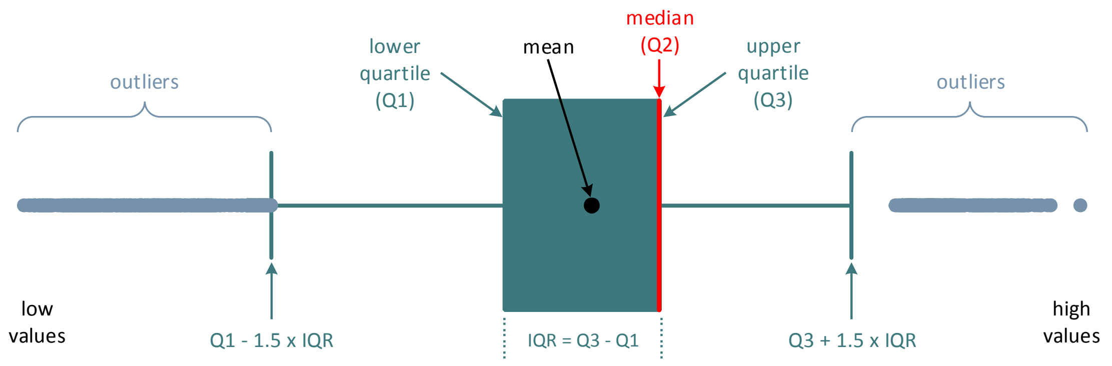


Figure 2.5 — A box plot is a compact representation of various statistical values of groups of data.

Network Interconnect

All nodes are connected via Ethernet and InfiniBand. The Gigabit Ethernet network (1 Gbit/s) is used for administrative tasks and computations with low bandwidth requirements. The InfiniBand network has 4X double data rate (16 Gbit/s) with full bisectional bandwidth, i.e., the full bandwidth is available for communication between any two nodes.

2.4 Box Plots

Box plots [Tukey, 1977] are a compact visualization for the distribution of groups of data points and "one of the most frequently used statistical graphics" [Wickham and Stryjewski, 2012]. They are used in this thesis to show and compare the distribution of a series of values. The visual elements of the box plot (see Figure 2.5) correspond to statistical values of a data set and its quartiles[5]:

The box shows the first (lower) and third (upper) quartile at the bottom and top, respectively. The first quartile ($Q1$) splits the data into the lower 25 % and upper 75 %, while the third quartile ($Q3$) splits it into the lower 75 % and upper 25 %.

The median splits the data set in half, thus also called the second quartile ($Q2$).

Whiskers show the values of the data that are within $1.5 \times IQR$.

[5]A quartile divides a data set into four equal groups, each containing 25 % of the data.

Outliers are below and above the whiskers. They are considered anomalies in the data set and shown as individual points so that their distribution and quantity can be estimated.

The interquartile range, IQR, is a measure for the dispersion of the values. The mean of the data set — in this thesis its arithmetic average — is not a robust measure, i.e., it is affected by outliers, and might not actually occur in the data set.

Part I

Abstraction

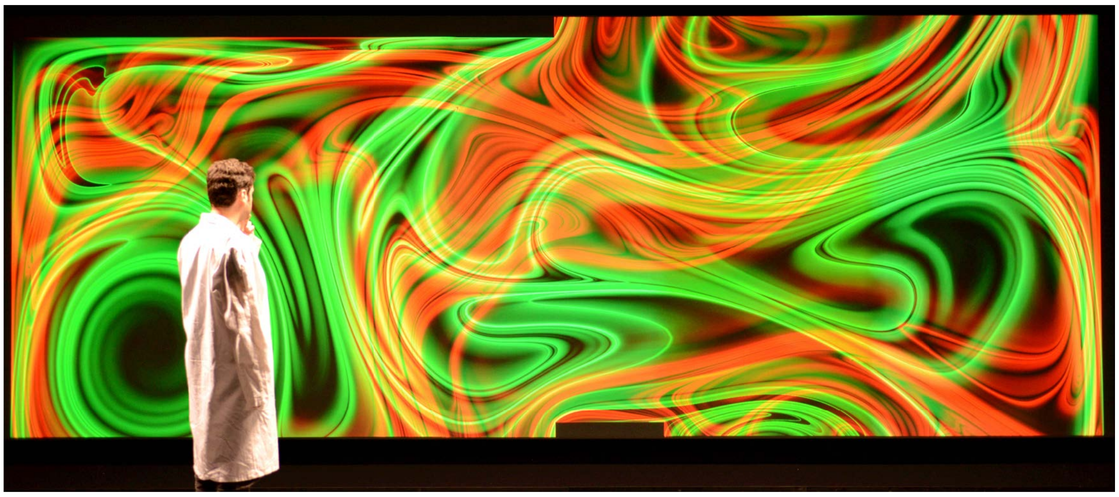


Figure I — Distributed finite-time Lyapunov exponent visualization of a time-dependent buoyant flow on the VVand (see Section 3.2.3).

Abstraction is an important aspect of many areas of life, like art, mathematics, or linguistics, but even more so in computer science. The visualization pipeline presented in Section 2.1.1 itself is an abstract process of transforming data into an image for interpretation by humans. While the use *graphics processing units* (GPUs) is almost mandatory for interactive visualization, it is not so clear which *application programming interface* (API) to use, even more so with the advent of new APIs like DirectX 12 or Vulkan (see Chapter 1). Furthermore, distributed rendering was always elaborate but is even more complex with semi-transparent objects. As such, utilizing large-scale, heterogeneous visualization systems like the VVand has high requirements in addition to development of appropriate rendering techniques.

The following chapters discuss abstraction methods to address both challenges. Chapter 3 introduces DIANA, an abstraction layer for unified access to different *processing units* (PUs), and discusses its utility as well as overhead in comparison to the supported APIs. Chapter 4 discusses *per-pixel linked lists* (PPLLs) as abstraction for simplified compositing in the context of distributed rendering of semi-transparent scenes.

Chapter 3

An Abstraction Layer for Heterogeneous Environments

Programming *graphics processing units* (GPUs) is different than *central processing units* (CPUs) in several ways (see Section 2.2.4). For one, they have discrete memory (with some exceptions like AMD's APUs), i.e., data needs to be transferred from main memory before it can be read or modified on the GPU. They are also designed for computer graphics tasks with different memory access patterns, for example when using texture filtering. In contrast, CPUs can access memory randomly with a smaller performance penalty than GPUs. GPU cores are combined in multiprocessors that operate in lock-step making branching costly. Even though challenging, GPUs and other *processing units* (PUs) are increasingly used, since they have proven to speed-up many computations by several orders of magnitude.

This chapter presents DIANA, the *distributed memory processing in a node abstraction layer*. It provides unified access to PUs and is designed to facilitate seamless switching between and simultaneous use of different PUs. This chapter is partially based on previous publications [Panagiotidis et al., 2011b, 2012].

3.1 Concepts and Architecture

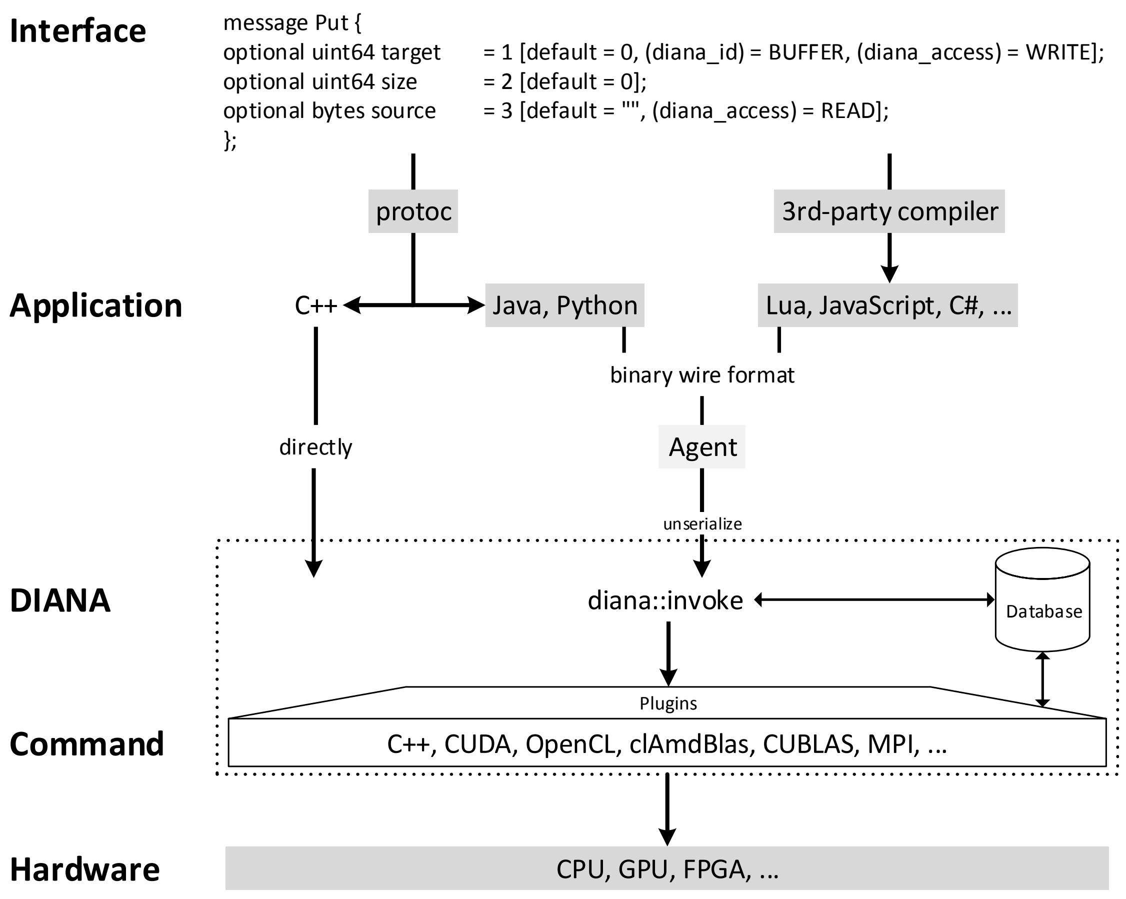
Existing libraries and frameworks simplify programming of PUs by focusing on ease of use or on specific use cases (see Section 2.2.5). In order to use them, they need to be embedded into applications at compile-time and their specific data structures have to be used. This makes switching between different frameworks and PUs challenging, if not impossible, for example due to the required time- and effort-overhead. Techniques like encapsulation and dependency injection remedy this to some extent but introduce additional overhead both during development and run time.

Ideally, switching between PUs, frameworks, or even different variants of an operation should be possible without rebuilding or changing an application. Applications should not be developed with a certain framework or *application programming interface* (API) in mind but instead, developers should focus on the data and the operations they wish to execute. Furthermore, using local or remote PUs should be seamless because the *non-uniform memory access* (NUMA) paradigm is similar to *distributed memory processing* (DMP), i.e., data needs to be transferred to a PU's memory where it is read and modified. DIANA is an approach to address these and the following challenges:

1. Provide unified access to current and future PUs and their APIs as well as frameworks and libraries.
2. Hide specific details, boilerplate code, and quirks from higher-level code.
3. Switch seamlessly between different PUs and operation implementations during run time with minimal source modification.
4. Provide this functionality across languages and platforms.

The main purpose of DIANA is to allow for rapid evaluation of new technologies, algorithms, and methods in heterogeneous, diverse, and evolving environments without code refactoring or even recompilation. It is a small C++ library utilizing several C++11 features (e.g., lambdas, std::function) which focuses on a semantic level where abstract interfaces define a contract for operations which are fulfilled by multiple implementations.

Since DIANA was initially published [Panagiotidis et al., 2011b], it has been extended and refined, most notably: devices and commands are now identified as unique unsigned integers, command interfaces are now realized using *Protocol Buffers* (PBs) (see Section 2.2.3), and all command implementations are



```
message Put {
optional uint64 target    = 1 [default = 0, (diana_id) = BUFFER, (diana_access) = WRITE];
optional uint64 size      = 2 [default = 0];
optional bytes source     = 3 [default = "", (diana_access) = READ];
};
```

Figure 3.1 — Architecture of DIANA. *Protocol Buffers* (PBs) are used to define interfaces to operations on *processing units* (PUs). These are compiled into host code that can be used to pass data either directly or via serialization to *diana::invoke*. There, the corresponding command is called which in turn calls APIs or low-level PU code. Agents can act as a bridge that mediates between DIANA and external parts (gray elements).

called via *diana::invoke*. The following describes the principal concepts that still apply as well as the changes, and presents new results based on the most recent version.

Figure 3.1 shows the architecture and main concepts of DIANA. Applications access PUs through commands and their interfaces. These commands are executed through *diana::invoke*, which in turn calls functions that were associated with the interface. The specific function to call can be queried from the internal database using SQL. Any functionality is isolated in plug-ins which are loaded dynamically during run time and then register operations to interfaces.

3.1.1 Data Types, Identifiers, and the Database

There are three object types in DIANA: devices, buffers, and commands. Each instance of such an object is represented by an identifier unique to that type. These opaque handles are mapped internally to API-specific data types and structures (see Table 3.1).

Devices and commands are registered in an SQLite database along with meta data. For example, device plugins store the available memory or their compute capability (e.g., double precision support, number of cores, clock frequency) so applications can query for devices with a specific amount of memory or minimum performance. Command implementations provide hints about their details, for example the library used. The queried identifiers are then supplied when invoking commands to find the associated function pointer and then passed to that particular function. Buffers are specified as parameters to commands, which resolve them to PU-specific pointers or data structures and pass them to kernels or library calls.

Arbitrary additional information can be stored in the database, for example log messages and exceptions for post-mortem inspection without interfering with an application's output. DIANA records each command invocation along with the device and command as well as the execution duration on the host and the device. Additionally, commands can insert similar profiling information (e.g., the duration of a specific API call) for fine-grained tracing.

Mapping database entries to C++ objects

The database is used primarily as flexible and convenient search mechanism. When an application queries for a device or command, the corresponding ID needs to be mapped to a C++ object or an API handle. These mappings are created and removed by plugins (see Section 3.1.4) when they are loaded and unloaded, respectively. For both devices and commands, an entry is created in the database first to obtain the ID which is then used as key in a hash map. Each device plugin manages its own hash map internally and provides methods to resolve an ID to API-specific handles (see Table 3.1). There is one global hash map for commands in DIANA to resolve *command IDs* to function pointers of command implementations. It is not exposed; instead, *command IDs* are passed to *diana::invoke* which looks up the implementation and calls it. The *command ID* is also passed to the command implementation to allow for multiple code paths in one function.

	CPU	CUDA	OpenCL
diana::DeviceID	—	int/cudaSetDevice	cl_command_queue
diana::BufferID	void*	void*	cl_mem/cl::Buffer
device.Alloc	std::malloc	cudaMalloc	cl::Buffer()
device.Free	std::free	cudaFree	~cl::Buffer()
device.Put	std::memcpy	cudaMemcpy	clEnqueueWriteBuffer
device.Get	std::memcpy	cudaMemcpy	clEnqueueReadBuffer

Table 3.1 — DIANA hides native pointers and memory operations behind unique identifiers and commands. Commands can ensure type-safety and perform boundary checking before passing the native handles to the respective API.

3.1.2 Commands

The basic unit of functionality in DIANA is the *Command*, which consists of two parts: an interface and implementations. The command interface contains the necessary data for an operation, similar to a function signature, while the implementation realizes the actual operation, for example calls to a library or PU API. Each command interface is a PB *message* which is compiled into C++ using *protoc* for DIANA itself. Applications, even developed in other languages than C++, can use any PB compiler to create the corresponding host code, as long as the same binary representation is written during serialization (see Figure 3.1). For computations, PBs are instantiated, filled with the relevant data, and then passed to the command implementation through *diana::invoke*.

Command implementations are functions with four parameters: a *device ID*, a *command ID*, an instance of a PB, and a callback that is run after the command is finished. These implementations resolve identifiers to data structures and pointers, may perform plausibility and constraint checks on parameters, and then pass those to the actual kernels and library calls.

diana::invoke

Commands can only be executed through *diana::invoke*. There, the function pointers are looked up, a unique *command status ID* for this execution is generated, and the starting time of this execution is recorded in the database. The *command status ID* can be used to refer to a specific computation, for example to query its duration.

By convention, users of DIANA should assume that the actual command implementation is asynchronous, i.e., the computation may not have completed or even started after *diana::invoke* returned. Commands signal their completion by calling *diana::commandFinished* when the corresponding API signaled that the PU finished the execution (see Figure 3.2). In *diana::commandFinished*, DIANA stores the tracing information in the database and calls a per-invocation user-supplied callback to allow applications to react, for example to start data transfers or new computations. This mechanism is also used to realize barriers by waiting until a flag has been set in the callback.

3.1.3 Devices and Memory

DIANA supports various PUs by encapsulating low-level methods for memory management and kernel execution. These *devices* are exposed to applications as a unique identifier which is internally resolved to specific handles for the respective PU API. Table 3.1 shows the mapping of the respective DIANA identifiers to PU-specific data structures.

Buffers

Buffers are created by calling *diana::createBuffer* with the desired size in bytes. This only creates a unique identifier while the actual memory allocation on a PU has to be done explicitly by calling the corresponding command (e.g., device.Alloc). There, the *buffer ID* is mapped to a native handle (see Table 3.1). While buffers only have a single ID, they can be allocated on multiple devices concurrently, each with distinct native handles. *Buffer IDs* are resolved to native data types by commands using a PU-specific mapping. There are also convenience methods to perform range and boundary checks on buffers to detect buffer under- and overflows.

Buffers are deliberately immutable to prevent repurposing, i.e., they cannot be resized once they are created. In other words, DIANA tries to enforce that a *buffer ID* always represents only one semantic object.

Remote Invocation

Since an operation is encapsulated by a command, it is trivial to relay it to a remote peer (see Figure 3.2). First, a server (i.e., the peer that will execute the actual command) announces its local devices and commands to clients, for example during initialization of the remote plugin. Then, each client registers a meta-command (the client stub) locally for each remote command. Consequently, when the application queries for a command, remote commands will also be returned as if they were local.

The client stub serializes the PB into the built-in wire format and sends it to the corresponding server along with the *device ID* and *command ID*. On receiving such a message, the server will deserialize the PB, query for the requested

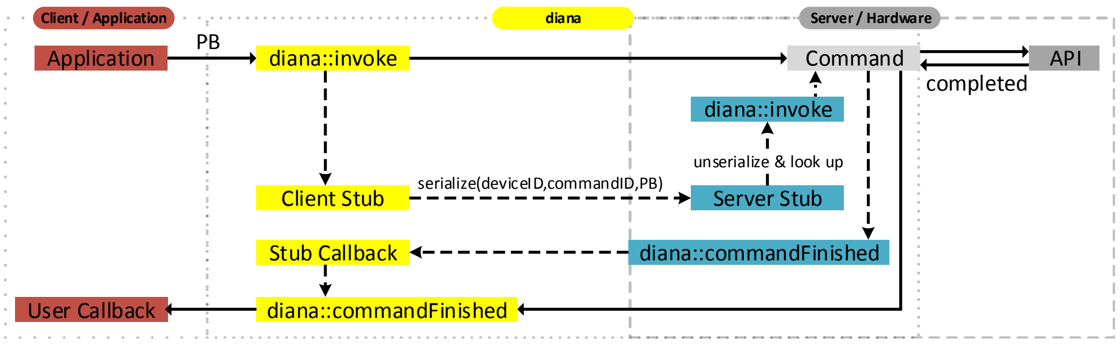


Figure 3.2 — For local computations (continuous lines), applications (red) pass a PB to *diana::invoke* which calls the corresponding command to execute some API calls. Remote invocations (dashed lines) are possible by routing the serialized information from the client (yellow) to a server (blue) which then looks up and executes the requested command.

device ID and *command ID*, and execute the command locally. The callback on the server then signals the completion to the client, which in turn calls the user-supplied callback locally.

3.1.4 Plugins

DIANA itself only consists of the basic functionality surrounding the database and *diana::invoke*. Commands, and thus operations on PUs, are encapsulated in plugins that are loaded dynamically during run time. Conceptually, each plugin should be responsible for a specific piece of functionality. For example, the default device plugins contain only the API- or PU-specific initialization and the respective low-level memory access (see Table 3.1). Through this plugin system, applications using DIANA only need the interface (i.e., the PB specification to create the corresponding host code) to execute an operation. The actual implementation is loaded during run time and, as described in Section 3.1.1, mapped to the interface, enabling switching between implementations without changing other parts of an application.

3.2 Evaluation

While DIANA provides several advantages, it also introduces some overhead both during development and during run time. The following discusses why DIANA should be used, how big the overhead is, and what it is composed of. In addition, an application using DIANA is discussed in the context of distributed flow visualization on the VVand.

3.2.1 Matrix-Matrix Multiplication

The *double precision general matrix multiplication* (DGEMM) is an important operation used in many domains and applications such as phylogenetics, fluid dynamics, computer vision, and finite-element modelling. It is also often used as benchmark for PUs as well as APIs due to its mix of memory transfers and computations, for example by *LINPACK*. Consequently, there are many implementations and variants available (Netlib's CBLAS, Intel's MKL, AMD's ACML and clAmdBlas, MAGMA, and ViennaCL to name a few) and thus it might be desirable to switch between these variants in order to provide multiple choices to users (e.g., the best implementation for their use case or environment), allow for verification of results, and to support new PUs and improved implementations.

Most DGEMM implementations use Netlib's definition in *BLAS* [Blackford et al., 2002] but differ in several details. Some have specific types for the data, others can restrict the computation to a subwindow of the matrices, and some expect their parameters in a different order. As such, multiple variants of this operation cannot be supported with a simple search-and-replace approach or even encapsulation of the function because the data structures need to be used throughout an application. Memory transfers also need to be considered for variants that are executed on a PU. The interface for DGEMM in DIANA (see Listing B.2) also follows *BLAS* but models the window parameters as PB *extensions* (Lines 31 to 39 in Listing B.2) because not all variants support them.

Figure 3.3 shows the behavior of several DGEMM implementations on different GPUs for increasing sizes of square matrices. This figure is not intended as an extensive analysis of the implementations or GPUs, but to illustrate why DIANA should be used.

The first observation to notice is that some of these variants are suboptimal, as they do not reach the theoretical or expected performance of the GPUs and only few reach it for some matrix sizes. To determine this, one would either need to use a benchmark for each variant or to adapt existing benchmarks each time a new variant emerges. Only one benchmark is necessary using DIANA by encapsulating variants in plugins and swapping them in and out without any other major changes.

Another observation is that both variants on the AMD HD 5870 outperform all CUDA variants. This is noteworthy because it was released in September 2009 [Wikipedia, 2015a], while all tested NVIDIA GPUs were released at least

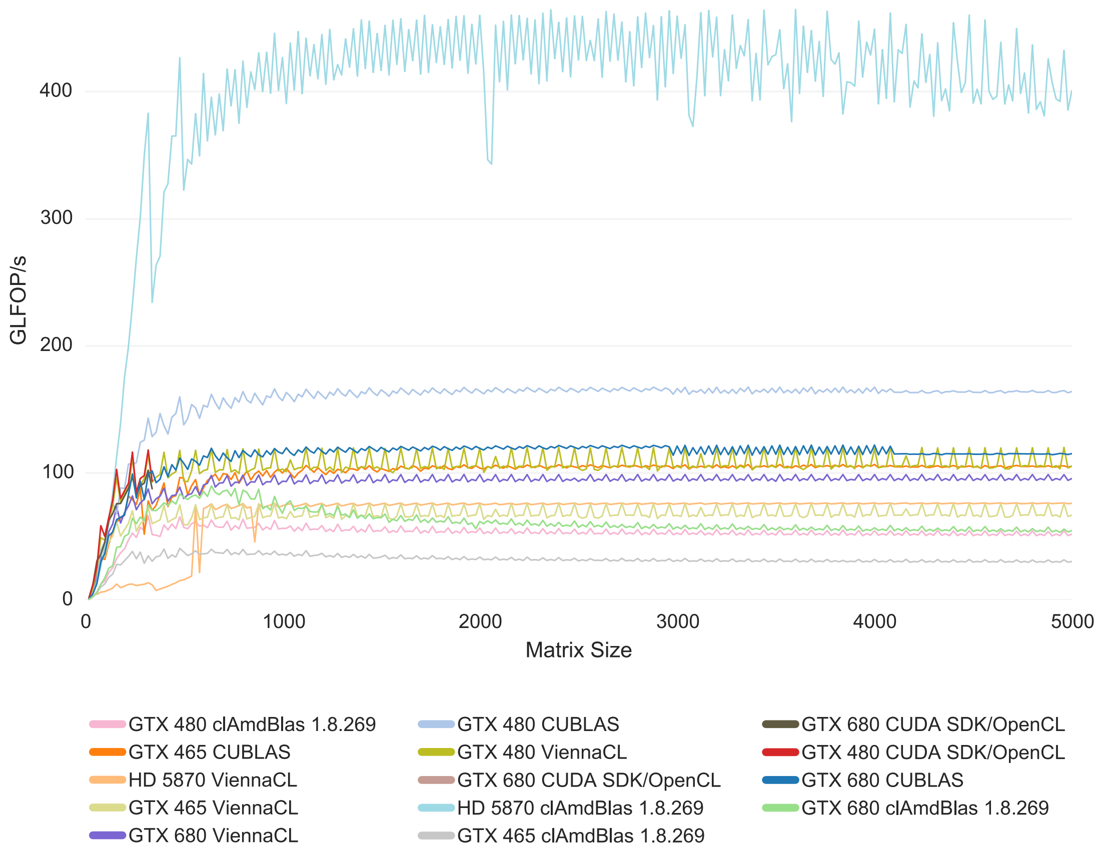


Figure 3.3 — Comparison of various implementations of the double precision general matrix multiplication on different GPUs.

six months later [Wikipedia, 2015b]. Compared to NVIDIA's GTX 480 and GTX 680, it was cheaper at release date and has a lower *thermal design power* (TDP), i.e., it consumes less energy and emits less heat. Considering this, using the AMD HD 5870 could result in a real competitive advantage, especially in an industrial context. However, at that time, OpenCL was just released and many developers already invested into NVIDIA and CUDA since there were only few other options (see Section 2.2.4). Now that OpenCL is available for some years and many libraries are available, it would make sense to switch between CUDA and OpenCL, to reach more users and gain access to the specific benefits (e.g., earlier release, less power consumption). This is time-consuming for applications that use CUDA directly or rely on CUDA-specific features but easily possible using DIANA and the plugin system.

3.2.2 Quantifying Overhead

Like every form of abstraction and indirection, DIANA also imposes a penalty during run time, since all calls have to be routed through *diana::invoke* and several entities are looked up from hash maps. The following test is performed to analyze this overhead: allocate memory on a PU, transfer data from the host, modify it on the PU, transfer it back to the host, and release the memory. These are the essential operations that applications have to use for any meaningful computation. The computation for this test, *sscal*, scales a large vector (25,000,000 single precision floating point numbers, 100 MB), though the operation itself and its duration is irrelevant for this test. Each individual operation is blocking, i.e., the test is waiting until the operation was finished on the PU using a barrier. The initialization and shutdown of the tested API (i.e., CUDA, OpenCL, DIANA) is not measured since they are executed once at the beginning and ending of an application, respectively, and are thus not critical to the run time. The test is repeated 10,000 times on the system *enka* (see Appendix A.1).

This test is heavily skewed against DIANA, which performs additional operations to increase safety, portability, and traceability. For one, the tested commands check for buffer overflows, i.e., they check whether the used buffers are large enough for the requested operations. Furthermore, when necessary, the memory transfer operations will allocate or resize buffers, again to prevent reading or writing out of bounds. During the execution, the start and end time points of subtasks are recorded in the database as well as the execution duration on the PU as reported by the API. Finally, since every operation in DIANA should be assumed asynchronous, completion is signaled using a callback. In contrast, for testing the other APIs, only the necessary, minimal calls are performed with the synchronization primitives of that API to ensure an operation has finished. No boundary checks are performed, no additional memory is allocated (i.e., all required memory is allocated once by the application), no callbacks are called, and no trace information or PU duration is neither queried nor recorded.

Figure 3.4 shows the distribution of the run times as box plots (see Section 2.4) for all tested operations from the applications' point of view, i.e., what users of the corresponding API would perceive as the time until an operation has finished and its results are available. As can be seen, the times for calling CUDA or OpenCL directly do not fluctuate much, i.e., they are very similar over all runs. For DIANA, the results fluctuate heavily and are at least $2-3$ times higher than the native API.

In general, the overhead can be attributed to the way DIANA operates internally, i.e., various look-ups, database queries, range checking of buffers, and strict error checking of API calls. Furthermore, completion of API calls are reported

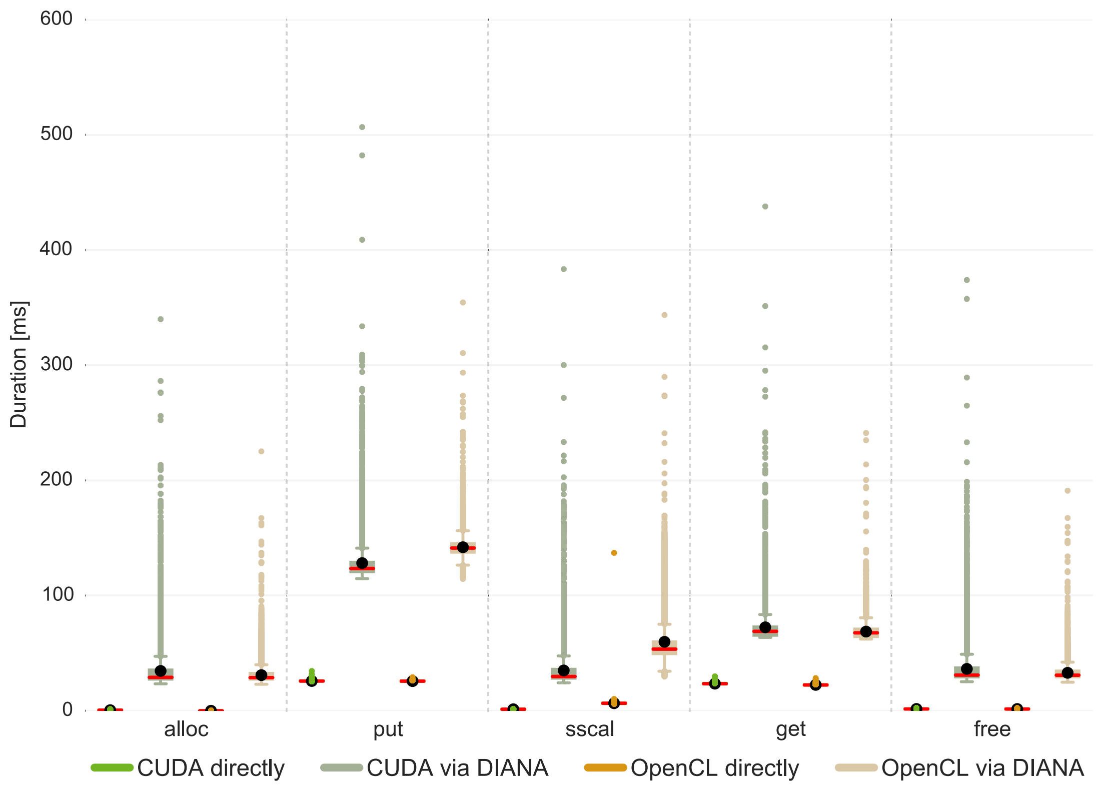


Figure 3.4 — Box plots (see Section 2.4) of the run times for PU memory operations and, as example, for scaling a large vector (*sscal*).

by the driver from their own threads, which induces some latency (e.g., due to context switching of threads). In particular, both CUDA and OpenCL support callbacks after certain events (e.g., kernel completion) but there are several restrictions for these callbacks, for example calling any API methods is prohibited, and the callback should not block or consume too much time. To escape these restrictions, the call to *diana::commandFinished* is queued into DIANA's own thread pool when the driver calls the completion callback, further increasing the latency.

Figure 3.5 depicts the various subtasks for copying data from host to the GPU (see Appendix B.3 for all other operations). The two top bars show the durations when calling OpenCL directly and via DIANA, while the three bottom bars show a break-down of this operation as recorded by DIANA in the database. The exact start and end time on the PU is unclear, since only the duration is reported and thus the bar is centered w.r.t. the API call. The red bars depict the delay until a subtask was started and how much time passed until the caller finished (i.e., the latency and the overhead).

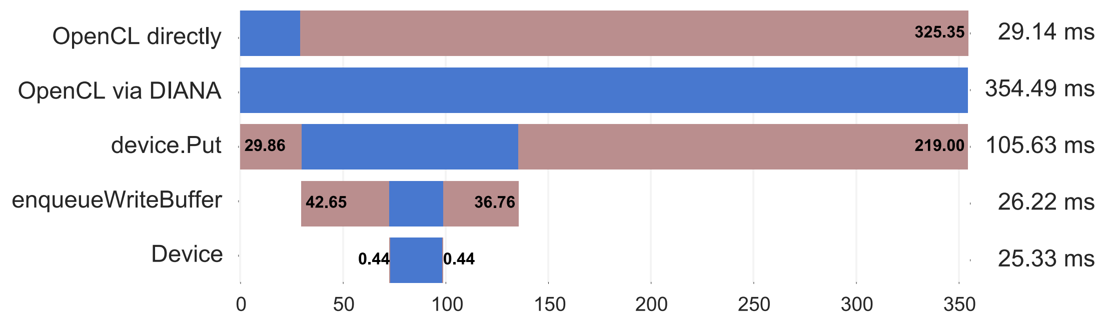


Figure 3.5 — Worst case for the memory transfer from host to GPU in OpenCL directly and via DIANA (top two bars). The bottom three bars depict the subtasks within DIANA for this command.

As can be seen, the actual OpenCL calls are similar when called directly or by DIANA (29.14 ms vs. 26.22 ms, 1st and 4th bar). Before the command is called, 29.86 ms are spent on copying the input data into the PB and inserting the trace information in the database (red bar at the beginning of *device.Put*). Then 42.65 ms are spent to verify buffer ranges, resolve the *device ID* to the *clCommandQueue* and the *buffer ID* to the *cl::Buffer* instance (red bar at the beginning of *enqueueWriteBuffer*). Then *enqueueWriteBuffer* is called and the command queue flushed, in the same way as when calling OpenCL directly. Finally, it takes another 219 ms until the driver notifies DIANA, which updates the tracing information with the completion time, thus ending the *device.Put* command. At that time point, the barrier at the call site is signaled and the test resumes with the next iteration.

Most of the overhead can be accounted to lookups and tracing of commands and buffers. Figure 3.6 shows the results for the same benchmark when the tracing is disabled, i.e., no entries are created in the database for: creation, modification, and destruction of buffers; beginning and completion of commands; duration of API calls on the host and PU. Overall, there are less fluctuations and outliers and the range of values is smaller. The means and averages decreased by up to 40 ms. Memory allocation, kernel execution, and memory deallocation in DIANA is now close or equal to their direct counterparts. The memory transfers are still slower due to copying the data into the PB and allocating memory in the command.

In conclusion, the gap between direct API calls and their equivalent in DIANA can be closed at the cost of losing tracing information. This is similar to compiling applications with debug information and assertions during development and removing that for release. The overhead for memory transfers remains for further investigation, since the actual API call and the duration on the GPU already matches the direct calls.

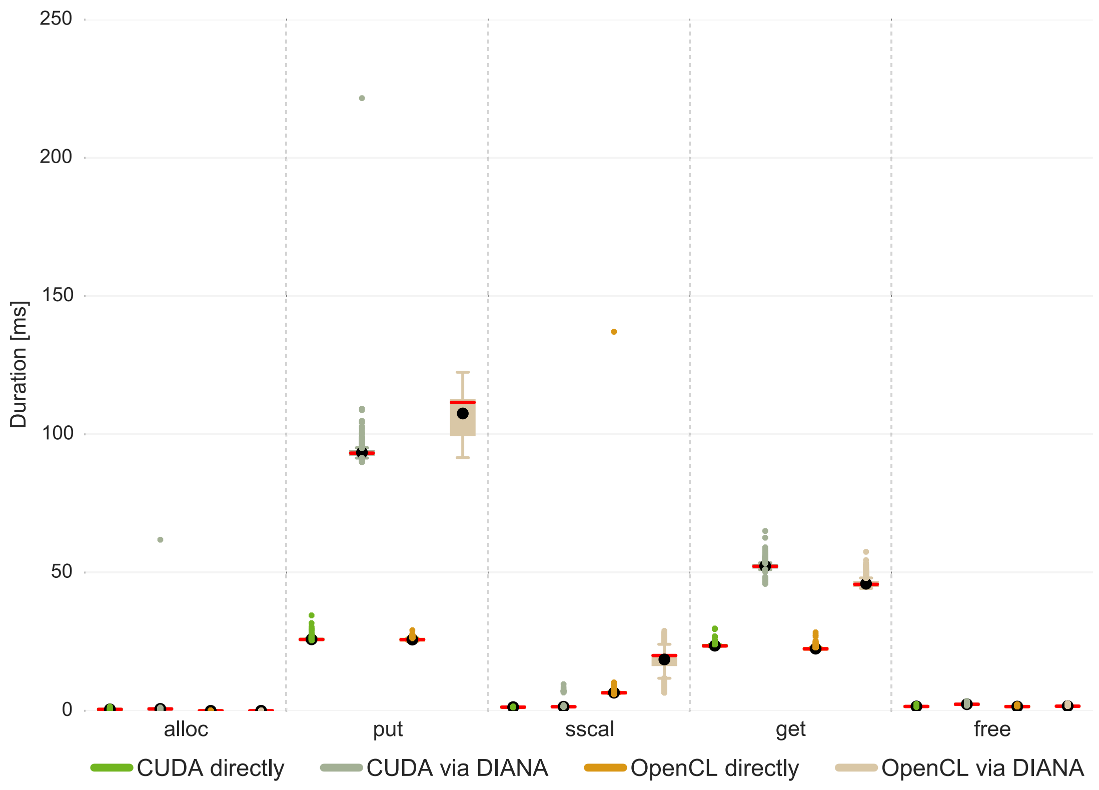


Figure 3.6 — Box plots of the run times for PU memory operations and *sscal* without buffer and command tracing in DIANA.

3.2.3 Distributed Finite-Time Lyapunov Exponent Visualization

Flow visualization is used to show direction and movement over time in vector fields and is applied in many fields such as fluid dynamics. While the local behavior of the vector field is often of interest, its global behavior, i.e., its topology, also provides valuable insight.

The *finite-time Lyapunov exponent* (FTLE) can be used to separate vector fields into regions of different behavior, i.e., as measure of divergence of two infinitesimal close particles over time. It is determined as the spectral norm of the right Cauchy-Green deformation tensor to obtain the biggest gradient of the flow map for each point in the computation domain. In 2D, a particle is traced for each pixel of the resulting image for a certain advection time and its final position is stored in a flow map. The particle trajectories can be integrated forward and backward to find repelling and attracting ridges, respectively. This computation is expensive even for small data sets because the advection time needs to be long enough, so that the ridges are long and sharp enough to allow extraction of *Lagrangian coherent structures* (LCS) [Haller, 2001] which actually

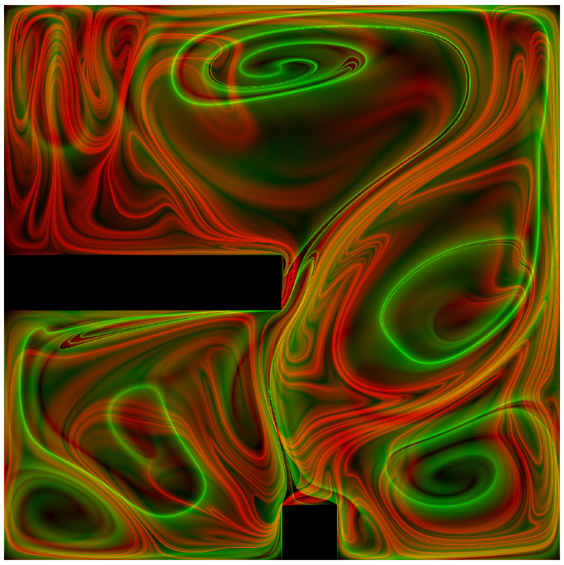
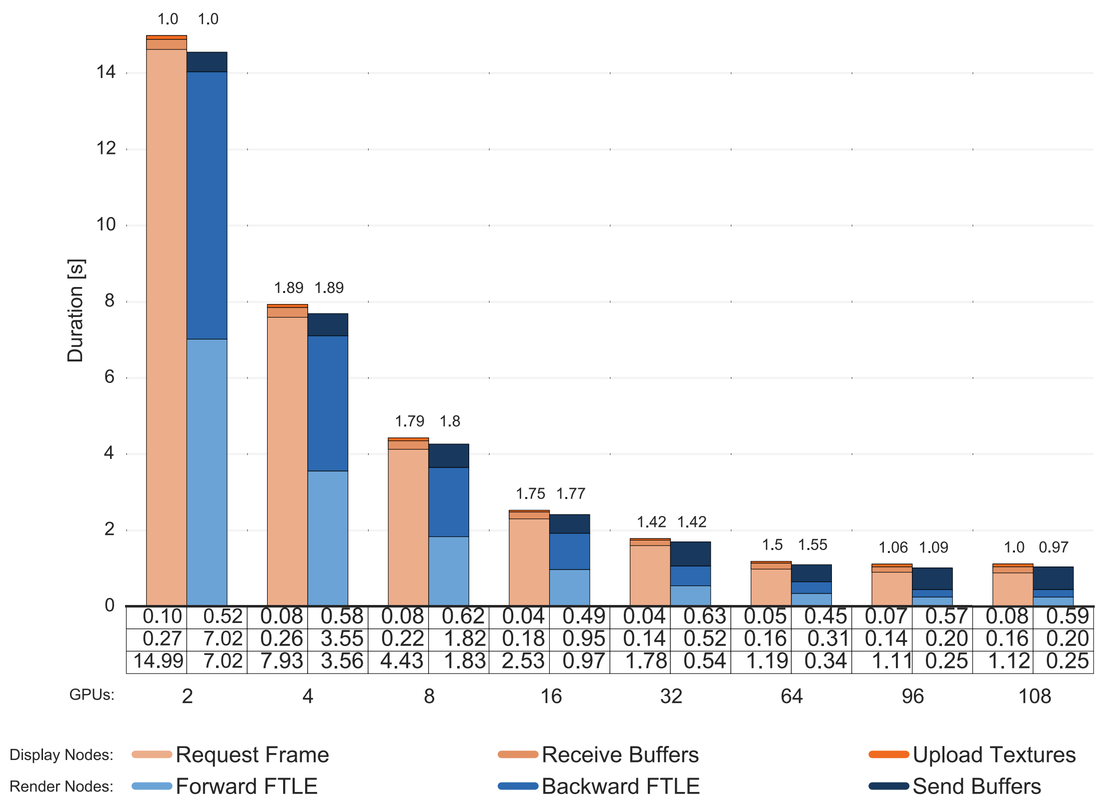


Figure 3.7 — FTLE visualization of a time-dependent buoyant flow with an obstacle. The forward and backward FTLE values are normalized and mapped to red and green, respectively, to show repelling and attracting LCS.

separate areas of the flow field. With longer advection time, a higher resolution is necessary to avoid aliasing and overdraw of the ridges. In certain cases, it is even necessary to create images with higher resolution than the display and then use supersampling [Üffinger et al., 2012]. As the particle tracing for each pixel can be performed independently, the computation domain can be partitioned among multiple nodes and GPUs and processed in parallel.

Both the flow map and FTLE computation are implemented as DIANA command in CUDA for parallelization on node level. These commands support tiled rendering, i.e., each can operate on a part of the final image. This allows for distribution of the computation to different nodes. The flow map is computed with a one-pixel wide ghost cell boundary so that the FTLE can be determined without communication while the FTLE itself is only written for the requested part of the image. The distributed visualization of the FTLE is realized using *MegaMol* (see Section 2.1.5) on five display nodes of the VVand and multiple render nodes running a specialized agent that uses DIANA without tracing of commands and buffers (see Section 3.2.2). Each time a frame is requested, the global parameters are broadcasted to the render nodes which compute the flow map and the FTLE. The render nodes manage memory themselves and stream the necessary time steps of the data set for the requested advection time to the GPU before computing both the forward and backward FTLE. The resulting FTLE fields are then sent to all display nodes over InfiniBand using MPI, so interactions in image space like panning and zooming can be achieved without recalculation.



	0.10	0.52	0.08	0.58	0.08	0.62	0.04	0.49	0.04	0.63	0.05	0.45	0.07	0.57	0.08	0.59
	0.27	7.02	0.26	3.55	0.22	1.82	0.18	0.95	0.14	0.52	0.16	0.31	0.14	0.20	0.16	0.20
	14.99	7.02	7.93	3.56	4.43	1.83	2.53	0.97	1.78	0.54	1.19	0.34	1.11	0.25	1.12	0.25
GPUs:	2		4		8		16		32		64		96		108	

Figure 3.8 — Average run times for distributed FTLE visualization. Note that the frame request duration on the display nodes includes the time to receive the buffers and upload them as textures while the render nodes calculate and send the FTLE fields sequentially. The speedup (above the bars) is determined w.r.t. the previous lower GPU count.

Figure 3.7 shows the FTLE visualization of one time step of a time-dependent buoyant flow with an obstacle simulated in a grid of 101×101 cells [Sadlo et al., 2012]. Figure I (see Page 27) shows a detail view of the same data set on the VVand. The forward and backward FTLE values are normalized and mapped to red and green, respectively, in both figures. Figure 3.8 shows the scalability behavior and relative speedup of this system w.r.t. the number of render nodes and GPUs, averaged over 100 subsequent time steps. The full image (4096×4096 pixels) is partitioned among all available GPUs of the render nodes, i.e., each of the n GPUs is rendering a partial image with a resolution of $4096 \times (4096/n)$ pixels.

The display nodes are idling most of the time until they start receiving a total of 128 MB for the FTLE buffers (the corresponding meta data is less than 100 B per partial image). This system behaves as expected when scaling from two to

16 GPUs, i.e., doubling the number of GPUs roughly halves the frame request duration with a speedup between 1.75 and 1.89. The render times are also very close for the forward and backward advection.

The speedup decreases with more GPUs. First, the computation duration is no longer halved when doubling the number of GPUs. This overhead of roughly 30 ms to 40 ms can be accounted to the overhead of starting and tracking the CUDA kernels. Second, even though the total amount of data is the same, more buffers are now transferred, and the send duration increases. This is accounted for in the send time of the render nodes, because they have to wait until the data is buffered by MPI on the display nodes. The actual receiving is then just a memory copy from MPI into an application buffer and thus not changing much w.r.t. the number of GPUs.

In conclusion, this system scales well until 64 GPUs when the FTLE can be visualized in ≈1.19 seconds per frame or ≈0.84 FPS. Adding more GPUs is only of marginal benefit and even counterproductive, for example when going from 96 to 108 GPUs there is no overall speedup and the communication is slower.

Since this system uses DIANA, it can be easily extended to nodes outside of the cluster and to architectures not supporting CUDA, for example by employing OpenCL [Panagiotidis et al., 2012]. This can be useful during development, when the target system might not be available (e.g., due to maintenance or other scheduled jobs) or to maximize utilization of all available resources (e.g., workstations that are idle during the night). In this case, it would be necessary to adapt the scheduling (i.e., the sizes of the partial images) to balance the overall load, since these nodes might have different compute capabilities.

3.3 Discussion

DIANA was initially designed as abstraction layer for various PUs with a strong focus on portability and ease of use while maintaining as much performance as possible and supporting the analysis of applications. Concepts like opaque identifiers, the use of PBs for interfaces, and the database as storage backend help but also introduce some disadvantages.

3.3.1 Portability and Usability

For current PUs, DIANA succeeded in facilitating the seamless switch between different PUs and multiple command implementations. With advances in technology, it is possible that future hardware requires yet another paradigm shift, similar to the one that happened with *general purpose computing on graphics processing units* (GPGPU) (see Chapter 1). In that case, it needs to be reevaluated whether that paradigm can be supported with DIANA.

Exposing the high degree of parallelism of various PUs in DIANA requires assuming that all command invocations are asynchronous. Notification of command completion is thus signaled using a callback. This event-driven model is a controversial way of handling concurrency and may lead to 'callback hell', the parallel programming counterpart of spaghetti code (e.g., due to misuse of the goto-statement). The tracing of buffers and commands in DIANA aims to help with this but its overhead is not negligible (see Section 3.2.2).

Even though DIANA simplifies many aspects when using PUs, it may be cumbersome to use DIANA itself. To call any PU operation, developers need to query for a *device ID* and *command ID*, and the data needs to be inserted into the corresponding PB. This is contrary to the traditional programming model where methods are called directly by name and the compiler or linker looks up the code or symbols, respectively, and parameters are given directly. Since buffers are opaque IDs, pointer arithmetic is not possible and thus important techniques for *high performance computing* (HPC) like windowing are not usable directly in DIANA but, for example, via PB extensions (see Section 3.2.1 and Appendix B.4). Even if it would be supported in some way, all operations on all PUs would need to implement it, which is not possible, for example with *cl_mem* in OpenCL.

The use of PBs as inherent way of passing data from an application to a PU facilitates their cross-platform and language-agnostic use. Any client can call any DIANA command by creating the binary format of a serialized PB and submit that to *diana::invoke*, for example using an agent that receives the binary PB through a socket, a file, or standard input. This is amplified by the availability of many PB implementations and compilers for various languages. As such, DIANA presents a very high degree of abstraction where callers do not know any details of an operation but the interface.

3.3.2 Querying for Devices and Commands

The mapping between IDs and API-specific entities is transparent to users. Selecting specific devices or even commands is thus only possible using meta information. Attributes like names and memory sizes are a good starting point for devices, but it is difficult to do the same for commands.

The DGEMM commands (see Section 3.2.1) supply a 'variant' field in the database. However, users need to be aware that such a field exists and even then it is unclear how to utilize it correctly. Using a fixed name during compilation would be counter-intuitive to the notion of not changing the source for switching or adding variants. Leaving this configurable during run time (e.g., as parameter in the applications' configuration) adds to the overall complexity but is the most effective way for now.

3.3.3 Database

Using a database as storage backend is challenging. On the one hand, the overhead is non-negligible for short operations (see Section 3.2.2) and data access requires an SQL query and traversing the results. On the other hand, there are no data races in the database due to transactions, the data is well-defined through schemas and constraints, and information can be retrieved very flexibly through SQL. Currently SQLite is embedded into DIANA[1], i.e., no additional services are necessary and there are many tools to inspect and manipulate the resulting files. The latency issues need to be investigated further and offloading all database accesses to a dedicated thread might improve this by not blocking the calling thread.

The benchmarks discussed in Section 3.2 allow for an interesting application. With the current command and buffer tracing, only the start and end times as well as the device durations are recorded. An extended tracing could also store the parameters of executed commands in the database, for example matrix sizes, which device and command variant was used, FTLE advection time, image sizes, etc. Storing this on a long-term basis can then be used to determine which device and command variant should be used for a given problem or to approximate the duration of a command before executing it. Additionally storing the result could even enable a form of memoization [Wikipedia, 2015c], a programming technique that caches results and skips recomputation of already known results. In that case, the actual command execution could be skipped by *diana::invoke* and the cached result returned instantly instead.

3.3.4 Protocol Buffers

Using PBs provides some advantages like a standard format to define interfaces and a well-defined binary representation (see Section 2.2.3). At the same time, they introduce some non-negligible overhead which could be reduced by improving their interface and internals but would require patching them which might conflict with Google's development plan.

Using PBs efficiently requires some planning throughout an application. In particular, memory re-/allocations need to be minimized, otherwise the overhead is too big w.r.t. the transfer operations (see Section 3.2.2). Furthermore, since

[1]SQLite As An Application File Format: https://www.sqlite.org/appfileformat.html.

buffers are realized as *std::string* in the C++ interface (when using Google's *protoc* compiler), their direct use requires casting to other types (e.g., float, double, int) for efficient use.

3.3.5 Availability of Commands

DIANA only provides the basic infrastructure for registering, querying, and invoking commands. The actual interfaces and their implementations are created manually as needed, which limits the a priori applicability of DIANA. Integrating larger APIs such as CUBLAS in advance is laborious but can be automated using special parsers and generators, though edge cases can arise that require manual adjustments.

Interfaces and implementations are added to DIANA mostly on demand of individual users. As such, these interfaces often reflect what those users or their implementations require. Adding further implementations reveal conflicts (e.g., float vs. double, dimensionality) or additional requirements (e.g., offsets and windows for buffers) that might not be realizable in all implementations. Even then, all other implementations might have to be updated when modifying the interface. Consequently, having a large library of different implementations is not only an asset but can also be a huge burden w.r.t. maintenance.

3.4 Summary

This chapter presented DIANA, a generic abstraction layer for accessing local and remote processing units. Interfaces of computations are defined as Protocol Buffers which are passed to user-provided implementations of the actual operation. While DIANA provides many benefits such as portability and traceability, the introduced overhead can be non-negligible but could be reduced by further engineering effort. Its major feature, the seamless switching between processing units and operation implementations, was discussed at the example of the dense matrix-matrix multiplication. Besides general-purpose computations, DIANA was successfully employed for large-scale distributed visualization of the finite-time Lyapunov exponent.

Chapter 4

Generic Compositing with Per-Pixel Linked Lists

The object space is often already partitioned in distributed simulations and preferably visualized on the node that also generated and stored the partial results. Generating an image in this case is challenging when semi-transparent objects are involved, as the whole scene needs to be depth-ordered for correct blending. Typical approaches like depth compositing (i.e., only using the nearest fragment for every pixel) or depth peeling are not feasible in this case due to their complexity or inaccuracy. As such, other methods are required for semi-transparent objects that involve either explicit sorting or approximations (see Section 2.1.3). Furthermore, for distributed rendering and large displays, it is important to provide accessible means for compositing the final image. Visualization methods can then be developed and deployed without worrying about how the final image is combined.

This chapter introduces the concept of *per-pixel linked lists* (PPLLs) as an abstraction for exchanging fragment data in these and further use cases and discusses merits and challenges of this technique. It is partially based on previous publications [Kauker et al., 2013a,b].

The protein data sets used in this chapter have been obtained from the *Protein Data Bank* (PDB) [Berman et al., 2000]. The triangle meshes of their structure and their surface have been generated with *VMD* [Humphrey et al., 1996] and *QuickSurf* [Krone et al., 2012], respectively.

4.1 Per-Pixel Linked Lists

Rasterizing scenes with semi-transparent objects is challenging and many works emerged that tackle this problem [Maule et al., 2011]. The classical approach, the painters algorithm [Foley et al., 1990], renders opaque objects first and then blends the depth-sorted semi-transparent objects with regard to the depth buffer. Depth Peeling [Everitt, 2001] is a method to render objects in multiple passes. In each pass, only those fragments are accepted that are in front of the nearest fragment but behind the fragment of the last pass. As a result, each pass peels one depth layer of the rendered objects and all depth layers can be blended correctly afterwards. This approach implicitly stores the ordering of the fragments for each pixel and not their explicit depth. Depth peeling has been optimized in many ways, for example by capturing multiple fragments per render pass [Bavoil and Myers, 2008; Liu et al., 2009], resolving z-fighting of fragments [Vasilakis and Fudos, 2011], or constraining the memory usage [Bavoil and Enderton, 2011]. The A-buffer [Carpenter, 1984] is a generic concept which stores fragments and their attributes for deferred blending. It is based on linked lists to store all fragments, which could not be implemented efficiently on *graphics processing units* (GPUs) before DirectX 11 or OpenGL 4.3. The R-buffer [Wittenbrink, 2001] was proposed as hardware implementation of a pointerless A-buffer but is not suitable for real-time graphics due to fragments being stored on disk and composited in a post-processing step. The stencil-routed k-buffer [Bavoil et al., 2007] introduced a generalization of the Z-buffer where up to k fragments can be stored for each pixel.

These previous works are well suited for local rendering but are not feasible for distributed rendering. Variants of depth peeling produce increasingly sparse partial images and additionally require the corresponding depth buffers, resulting in high amounts of data to transfer. Furthermore, sorting the depth layers from multiple nodes is complex since all layers need to be considered for correct blending, which might not be possible due to memory constraints. While A-buffer variants are space-efficient, they suffer from artifacts and might not capture all fragments due to constant memory, resulting in imprecise images. On modern GPUs this can be alleviated through PPLLs, first implemented by Yang et al. [2010], which are effectively only limited by available memory. Different types of memory layouts for PPLLs were evaluated by Knowles et al. [2012].

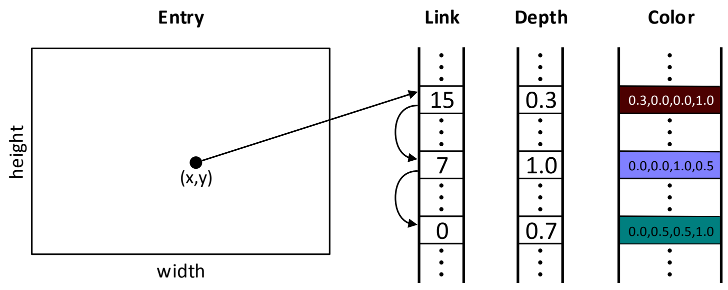


Figure 4.1 — PPLLs implemented as structure of arrays, i.e., each attribute is stored in its own buffer and addressed using the same index.

4.1.1 Data Structures and Operations

The primary goal of PPLLs is to provide unbounded storage for attributes for each pixel of the final image. The essential data structures (see Figure 4.1) to accomplish this are the entry addresses of each pixels' linked list, the links, and one or more attribute buffers.

The attribute buffers store information about the elements that are required in later stages of the rendering. Depth and color are required for blending, but additional per-fragment data is possible, for example the material, normal, etc. The attributes are stored in linear buffers that are addressed using contiguous indices. A counter stores the index of the last inserted attribute. The *list entry* buffer is a 2D buffer that is addressed with pixel coordinates. Each entry contains the end of the linked list for a given pixel. The *link* buffer contains the index of the previous element of the linked list. This data structure allows for all typical operations on a single linked list, for example insertion and traversal.

Initialization

Before collecting the attributes, the entry buffer and the counter need to be reset. The entry buffer is set to 0 for all pixels and the counter is reset to 1. All other PPLL buffers do not need to be cleared, as long as they are accessed with an index that is smaller than $counter - 1$.

Appending

Appending to a pixels' list (see Figure 4.2) is achieved by increasing the attribute counter to reserve an entry in the attribute buffers. The counters' previous value is then used as index for the attribute buffers. To establish the link, the old entry for this pixel is stored in the link buffer and the new index in the entry buffer, respectively.

On concurrent architectures, this requires two atomic operations to prevent data races (see Algorithm 4.1). First, the counter increment needs to be atomic so no other thread reserves the same space in the buffers. Second, updating the

```
$index \leftarrow$ AtomicIncrement(Counter)
Link[$index$] $\leftarrow$ AtomicExchange(Entry[$x$][$y$], $index$)
Depth[$index$] $\leftarrow$ $depth$
Color[$index$] $\leftarrow$ $color$
```

Algorithm 4.1 — Appending to a PPLL. The atomic operations read the old value at the memory location and store the new value atomically before returning the old value.

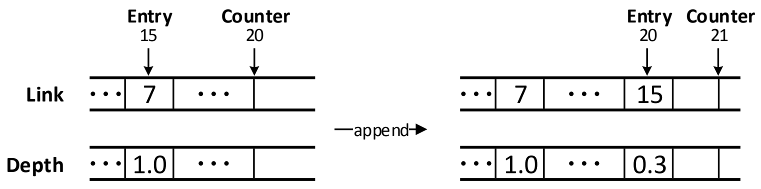


Figure 4.2 — Appending an element to a PPLL.

entry requires an atomic exchange so no other thread stores its new link after the entry address has been read.

Traversal

A linked list can be traversed by starting from the index in the entry buffer and following the links until it contains 0, as outlined in Algorithm 4.2. All pixels' linked list can be traversed in parallel without any data races as the linked lists for any two different pixels do not access the same memory locations.

```
$index \leftarrow$ Entry[$x$][$y$]
$length \leftarrow 0$
while $index > 0$ do
    $depth \leftarrow$ Depth[$index$]
    $color \leftarrow$ Color[$index$]
    // process attributes in some way
    $length \leftarrow length + 1$
    $index \leftarrow$ Link[$index$]
end
// $length$ now contains the number of items in this linked list
```

Algorithm 4.2 — Traversal of a PPLL.

Merging

Handling multiple PPLLs is easier when they are all combined into one. This also simplifies various operations (e.g., blending, compacting) which require a total order over all fragments of one pixel. Two lists can be merged by traversing one and appending to the other (see Algorithm 4.3). This can be repeated until all lists have been merged.

```
$index \leftarrow$ EntrySource$[x][y]$
while $index > 0$ do
    // append to other list, see Algorithm 4.1
    append($x$, $y$, DepthSource $[index]$, ColorSource $[index]$)
    $index \leftarrow$ LinkSource$[index]$
end
```

Algorithm 4.3 — Merging of the PPLL *Source* into another one.

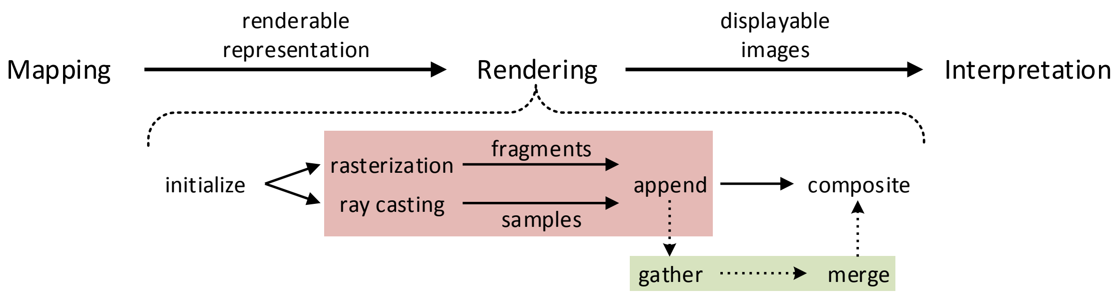
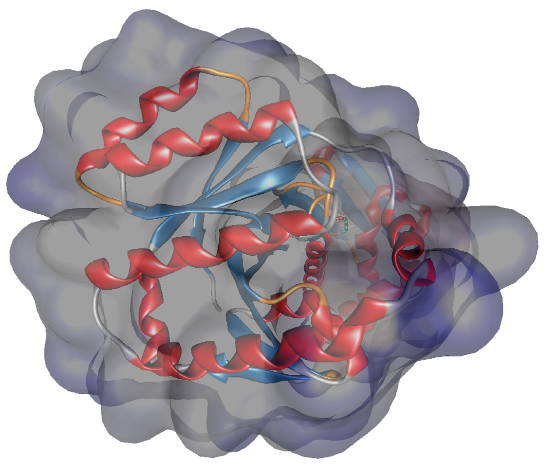


Figure 4.3 — The rendering stage of the visualization pipeline (Figure 2.1) when using PPLLs. The red parts are executed for each object to collect the fragments and samples (see Algorithm 4.1). For distributed rendering (dotted arrows), the display nodes need to gather the PPLL buffers from each render node and merge (see Algorithm 4.3) them into one. The final image is created by sorting the PPLLs and blending them for each pixel in parallel.

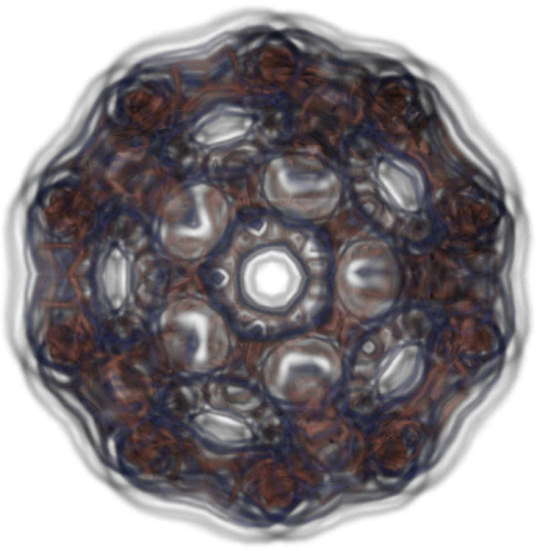
4.1.2 Implementation Overview

PPLLs can be used with any visualization technique that ultimately produces fragments or samples for each pixel, for example any method that employs rasterization or ray casting. During rendering, instead of blending directly, the attributes of the fragment or sample are appended into the linked list for the corresponding pixel (see Figure 4.3). Afterwards, the linked list for each pixel is sorted and can then be blended correctly. Collecting the fragments' attributes during rendering can be optimized in several ways and is possible in a single render pass [Yang et al., 2010; Knowles et al., 2012; Kauker et al., 2013a].

Integrating PPLLs into existing rendering techniques is straightforward. When rendering with fragment shaders, the outputs are stored in the PPLLs — as outlined in Algorithm 4.1 — instead of the usual outputs (see Listing 4.1). The same applies to kernels or compute shaders that would write per-fragment or per-sample attributes into buffers. In OpenGL, a framebuffer object without attachments can be used in conjunction with PPLL to prevent writing to the default framebuffer.



(a) Mevalonate kinase (PDB ID: 1VIS), 113,416 triangles, 3.41 ms, 7,905,877 fragments, 106.48 MB.



(b) C-60 Buckminsterfullerene, $32 \times 32 \times 32$ voxel, 115.21 ms, 33,174,043 samples, 395.65 MB.

Figure 4.4 — (a) Order-independent rendering of meshes of the inner crystal structure of the *mevalonate kinase* (visualized as ribbons) and its Gaussian surface. (b) Volume visualization of the *bucky ball* (C-60 Buckminsterfullerene). Both Figures were rendered in a resolution of 2048×2048 pixels on the system *enka* (see Appendix A.1) using PPLLs.

Listing 4.1 — Appending fragment depth and color in an OpenGL fragment shader instead of writing it to the shaders' outputs.

```
//gl_FragColor = color;
//gl_FragDepth = depth;
ppll_append(gl_FragCoord.xy, depth, color);
```

As mentioned in Algorithm 4.1, appending to the PPLL buffers requires atomic operations to prevent undefined behavior and data races on concurrent architectures. In other words, direct modification of an entries' value is not trivial because multiple threads could write to the same offset of multiple buffers. Writing (instead of appending) to the PPLL buffers with defined behavior is only possible with mutexes, which are not properly supported in OpenGL, CUDA, or OpenCL, i.e., they need to be emulated with atomic operations.

Order-Independent Transparency

One of the uses for PPLLs is to collect all fragments of a scene containing semi-transparent objects for correct blending without the need to sort the geometry beforehand. This is achieved by rendering the scene normally without a depth

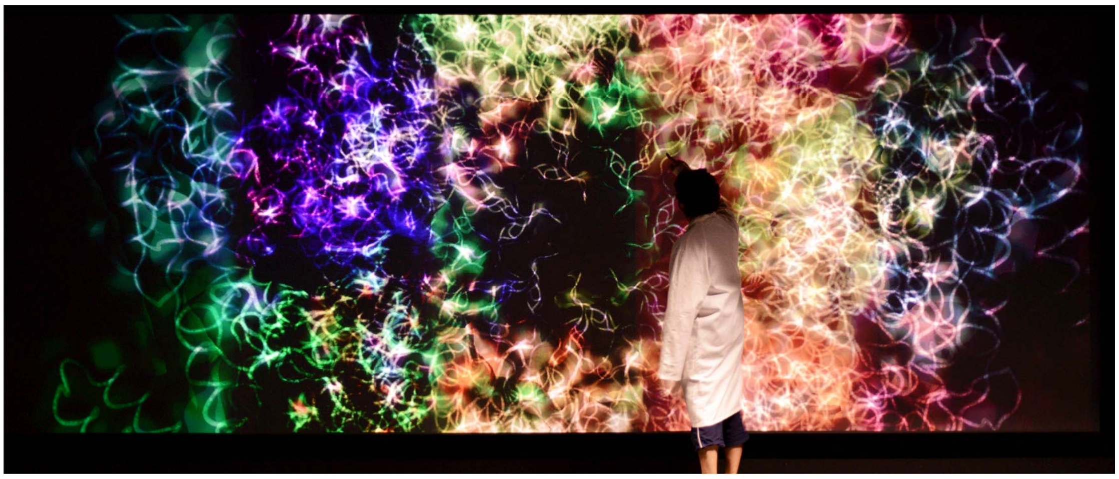


Figure 4.5 — Detail view of a *chaperonin* (PDB ID: 1AON) on the VVand rendered with PPLLs on 10 render nodes.

test and appending all emitted fragments in linked lists. The final image is composited by sorting the linked lists for each pixel (in parallel) and to blend the sorted fragments (e.g., back-to-front) with the desired blending equation. A typical use case for this method is the combination of different visualization techniques and materials (i.e., colors, opacity, textures). Figure 4.4a shows the *mevalonate kinase* [Badger et al., 2005] (PDB ID: 1VIS) rendered using this method.

Volume Ray Casting

This approach also works for techniques based on ray marching by appending samples to PPLLs instead of processing them immediately (see Figure 4.4b). Depending on the use case, the samples' values can be stored instead of their color and the color mapping is done during blending. This allows for changing the transfer function often or rapidly, as the (comparatively more expensive) ray marching does not have to be repeated and the new transfer function is visible immediately.

For this rendering technique many more entries are created compared to rasterization, since the sampling distance along a ray has to be small enough to reconstruct the signal of the volume accurately. As such, it is feasible to combine multiple samples into one PPLL entry, instead of storing them all explicitly. Depending on the volume and the transfer function, this might introduce a noticeable error. Ideally, the number of samples that are combined into one PPLL entry should be a parameter that users can change interactively.

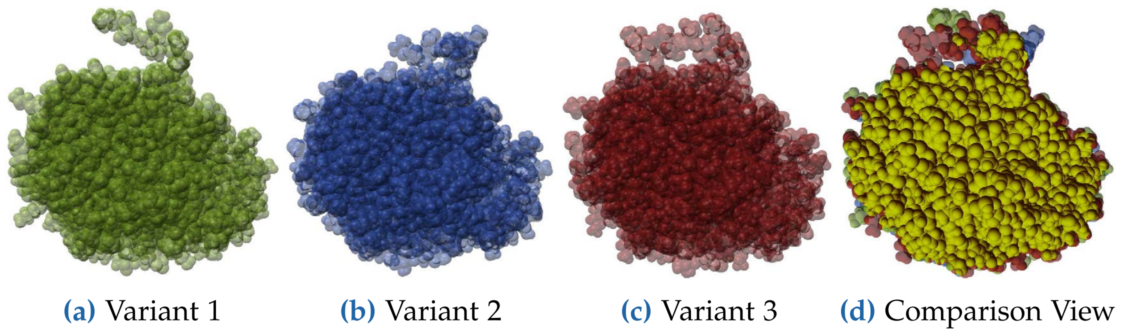


Figure 4.6 — Different variants of a *lipase* (PDB ID: 2VEO) rendered individually (a)–(c) and merged (d) for comparison of the variants.

Distributed Rendering

PPLLs allow for an oblivious approach to distributed visualization where creating, merging, compositing, and showing the PPLLs can be done be different processes. In particular, no details of any of those stages are relevant to the others as only PPLLs are passed and processed. The fragments are rendered normally into PPLLs on the render nodes and then collected by each display node which merges the incoming PPLLs into one and composites that. The *chaperonin* (PDB ID: 1AON) in Figure 4.5 has been rendered using this approach on 10 render nodes and 5 display nodes.

This process can be optimized when render nodes determine the front-most opaque fragments first with z-buffering. These fragments are then sent to display nodes, which can show them as a preview. Meanwhile, each render node collects the semi-transparent fragments in a PPLL, discarding those that are behind the opaque fragments. When this is finished, the PPLL is sent to the display nodes, which only need to merge those that are in front of the nearest opaque fragment of all render nodes. This merging can be organized hierarchically to further reduce the transferred buffers and processing times.

Comparative Analysis

Protein data sets from the PDB can contain multiple variants, for example due to different conformations or simulated boundary conditions. Figure 4.6 shows this for a *lipase* (PDB ID: 2VEO). While the different variants can be examined separately (see Figures 4.6a to 4.6c), it might be difficult to discern the shared surface or differences. Using PPLLs they can be merged into a single view (see Figure 4.6d) for comparison of their variants. For this technique, a fragments' normal is required in addition to its depth and color.

The variants are individually rendered with semi-transparency into PPLLs; this step is parallelizable, for example by rendering each variant on a dedicated node. When merging the PPLLs into one, the variant ID (e.g., a unique integer for each variant) is also stored for each fragment in an additional attribute buffer. During compositing, the fragments are sorted per-pixel and the depths and normals of subsequent fragments are compared. If they are similar w.r.t. a threshold, they are shared by the corresponding variants. For each fragment that is shared among some variants, only one fragment is blended with a color different from the variants. All other fragments are blended normally, i.e., according to the stored color of the variant.

This approach shows common parts of the variants together with their differences. Additionally, their separate presentation can be shown by considering only the corresponding fragments. This also allows for partial combination of the variants.

To reduce visual clutter, the shared fragments can be blended opaquely, while the individual fragments can remain semi-transparent. Furthermore, since the rendering of the variants might not align perfectly, a search in the image-space neighborhood can improve the detection of shared fragments. This search is limited to the front-most fragment in the neighborhood because searching multiple lists is time-consuming and the shared fragments are opaque.

4.2 Evaluation

PPLLs lend themselves to local rendering (see Figure 4.7) but are especially useful for distributed visualization as they allow for correct compositing of semi-transparent objects without sorting. Yet, their applicability for large displays is questionable due to their high memory requirements even for simple scenes (see Figure 4.4a). The following analyzes distributed mesh rendering with PPLLs to determine whether they are suitable for visualization on a large display like the VVand.

This analysis is realized using a generic PPLL render module in *MegaMol* (see Section 2.1.5). Frames are requested by broadcasting the view matrix and the image resolution to all render nodes, which run a dedicated agent that rasterizes parts of the mesh and returns the fragments as PPLL structure (see Section 4.1.1), i.e., counter, entry, links, depths, and colors. All processes communicate via InfiniBand using MPI. The render nodes broadcast their PPLL buffers to all display nodes, so image space interaction like zooming and panning is possible without requesting a new frame, similarly to the *finite-time Lyapunov exponent* (FTLE) visualization in Section 3.2.3.

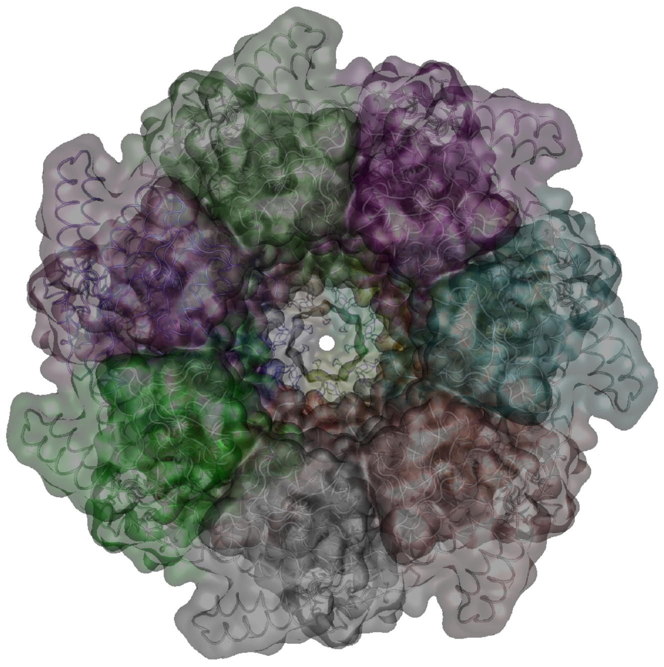
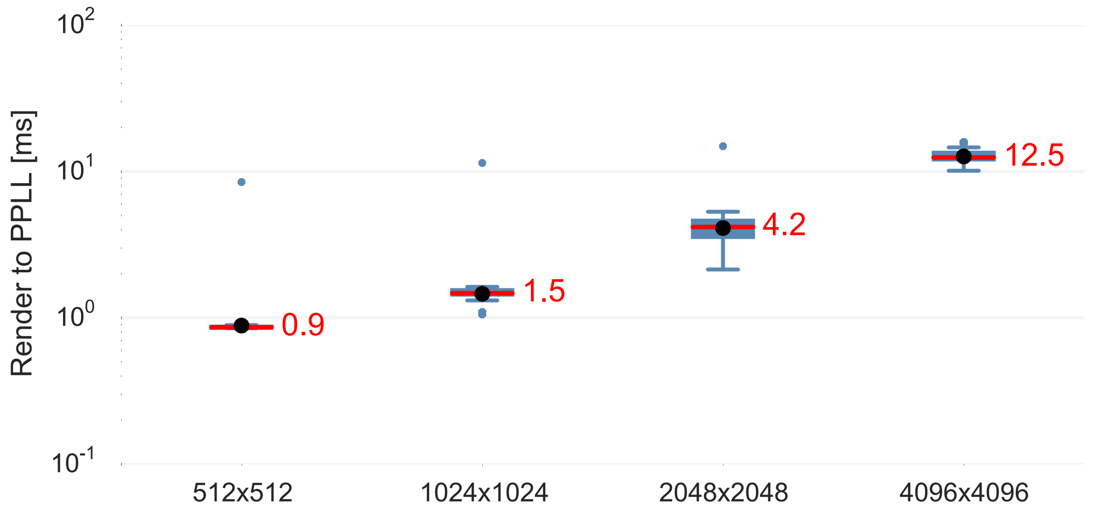


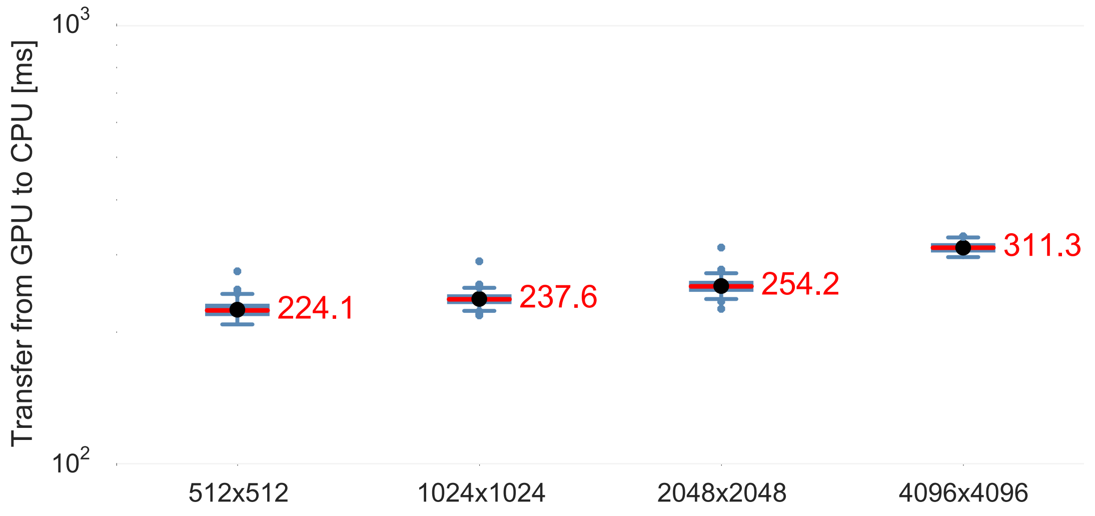
Figure 4.7 — Front view of a *chaperonin* (PDB ID: 1AON). The 2,631,800 triangles of its structure and surface were rendered in 13.01 ms in an image resolution of 2048 × 2048 pixels on the system *enka* (see Appendix A.1) and resulted in 17,205,397 fragments (212.90 MB).

The triangle mesh of a *chaperonin* (PDB ID: 1AON, 2,631,800 triangles) is used for the measurements. It is rendered on 10 render nodes into PPLLs, i.e., each render node processes 10 % of the mesh, with a constant alpha of 0.3. A full view of the protein (see Figure 4.7) is rotated around the y-axis by one degree each frame for 360 frames, i.e., until it was completely rotated once. This test is repeated for the image resolutions 512 × 512, 1024 × 1024, 2048 × 2048, and 4096 × 4096 pixels. The only optimization in this setup is the quantization of the color from four float channels into one unsigned integer, i.e., each float color channel is transformed from $[0.0, 1.0]$ to $[0, 255]$ and stored in the bytes of the integer using *packUnorm4x8* in the shader.

For the data collection, the mesh is automatically rotated and the frame requested. Then the 10 render nodes (see Appendix A.3) process their parts of the mesh and send the resulting PPLLs buffers to each display node (see Appendix A.2). The display nodes greedily poll for the results (i.e., similar to a spinlock) and copy incoming data to the GPU. The first received PPLLs of a frame are copied directly into the primary local PPLL buffer that is used for compositing. All subsequent PPLLs are copied into a second local PPLL buffer and then merged with the primary. This is repeated sequentially until all render nodes responded. The primary PPLL buffer is then composited into a framebuffer object with the requested image resolution, which is then rendered as quad on the VVand.



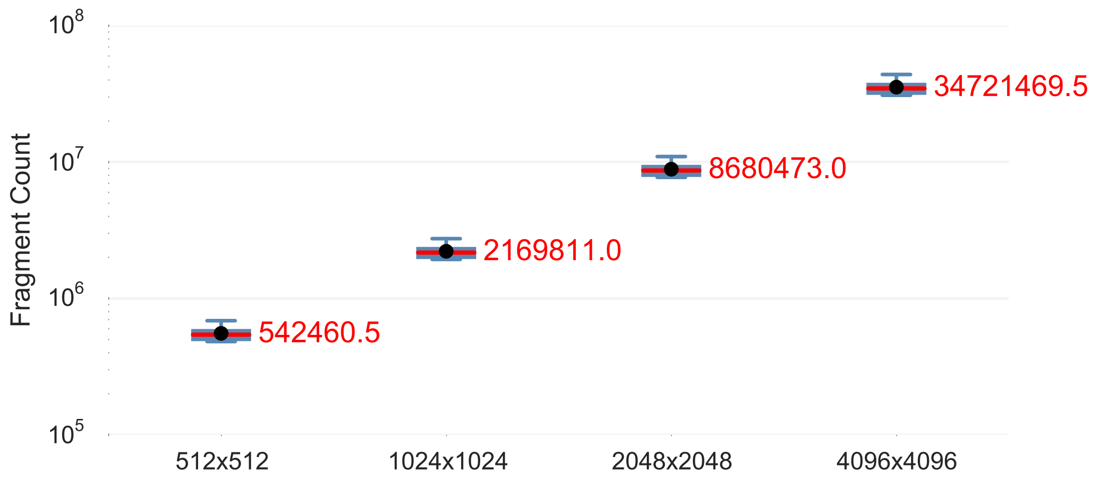
(a) Average durations for rendering the mesh on the render nodes.



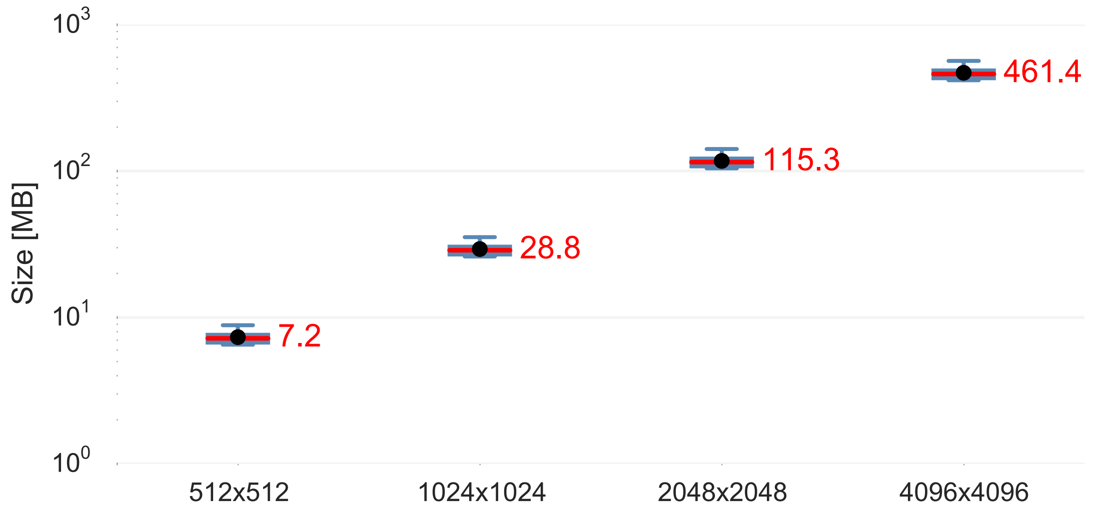
(b) Average durations to copy the PPLL from the GPU to the CPU.

Figure 4.8 — Box plots (see Section 2.4) of the average durations for (a) rendering the *chaperonin* (PDB ID: 1AON) and (b) transferring the PPLLs to the CPU.

Figure 4.8a shows the durations for rendering the mesh averaged over all render nodes and frames. The render times do not sway much or at all. While the time does not even double when increasing the resolution from 512×512 to 1024×1024 pixels, it almost quadruples for 1024×1024 to 2048×2048 pixels, and roughly triples for 2048×2048 to 4096×4096 pixels. Most of the duration for the lower resolutions is comprised of the overhead to set up the draw call and initiate it, thus they behave so differently. The duration to transfer the PPLLs from the GPU to the CPU does not sway or change much (see Figure 4.8b) due to the bandwidth of the GPUs.



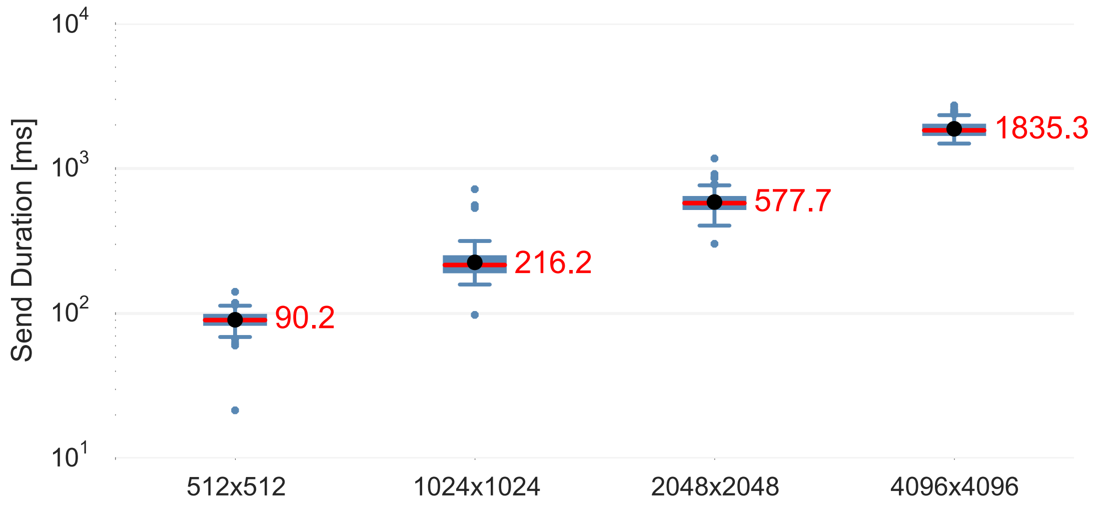
(a) Average number of fragments received on the display nodes.



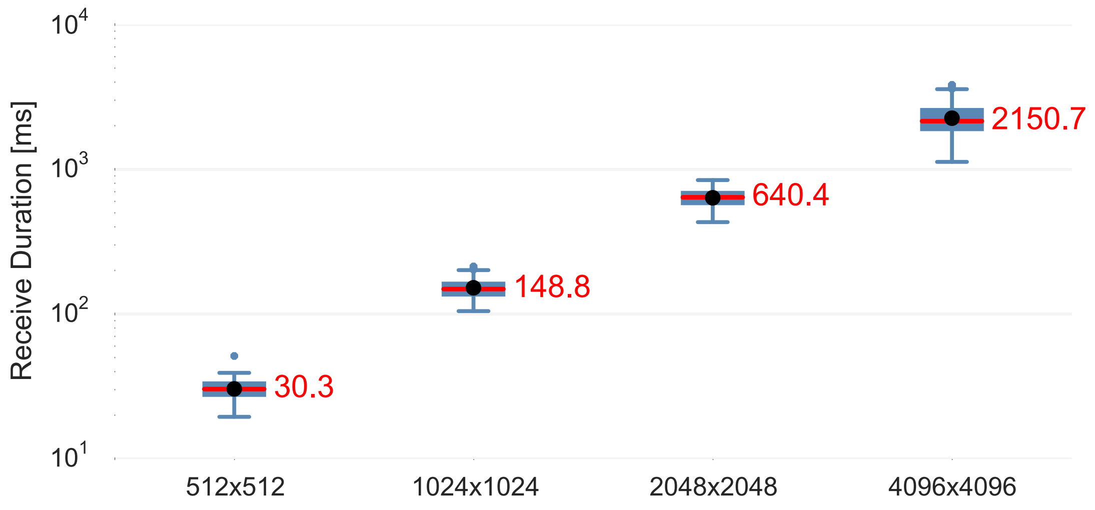
(b) Average megabytes received on the display nodes.

Figure 4.9 — Box plots of the average number of fragments that were received (a) and the corresponding amount of data that was transferred (b) on the display nodes, respectively.

The number of fragments behaves as expected (see Figure 4.9a), i.e., it roughly quadruples when the resolution is doubled in both dimensions. Consequently, the amount of data that the display nodes receive (see Figure 4.9b) changes in the same way. The box plots indicate little swaying of the number of fragments (and the data size) during the rotation of the mesh (see Figure 4.9a), which matches the behavior of the transfer times on the render nodes (see Figure 4.8b).



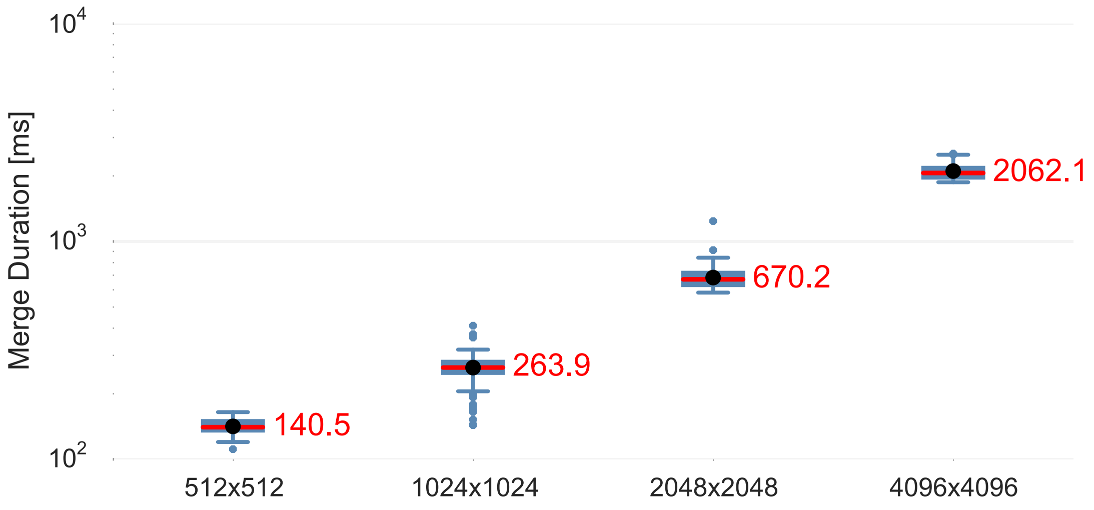
(a) Average sending durations on the render nodes.



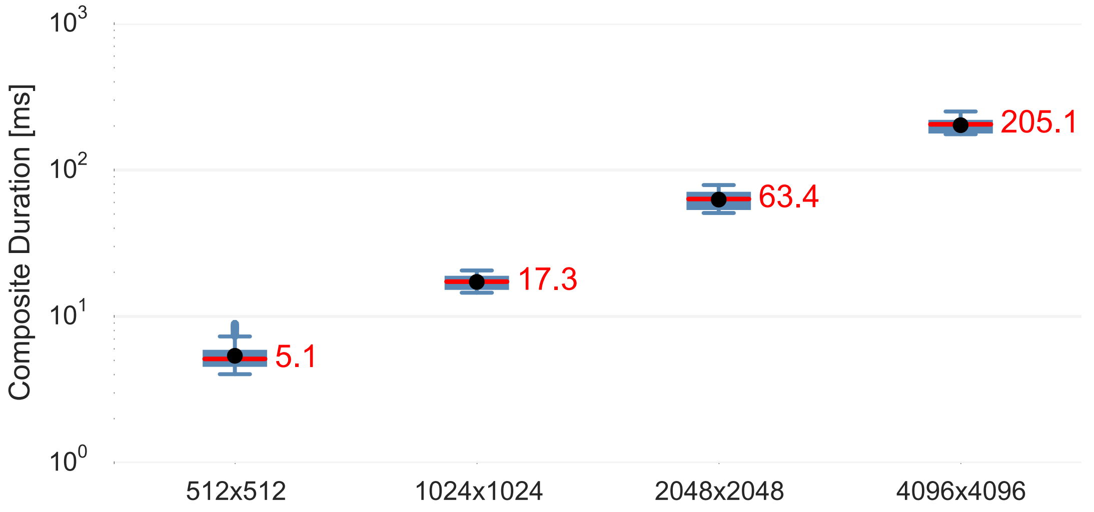
(b) Average receiving durations on the display nodes.

Figure 4.10 — Box plots of the average durations for (a) sending the PPLLs from the render nodes and (b) receiving them on the display nodes.

Figure 4.10 shows the average transfer times from the render nodes (see Figure 4.10a) and to the display nodes (see Figure 4.10b), respectively. Note that this uses the same communication pattern as the FTLE visualization in Section 3.2.3, i.e., every render node sends its data to all display nodes and waits until they received it. Both sending and receiving become slower with increasing resolution. This could be due to the internals of MPI, i.e., more small data can be buffered until the application retrieves it. As such, each display node buffers fewer incoming requests with bigger PPLL buffers from the render nodes, resulting in increasingly longer receiving times.



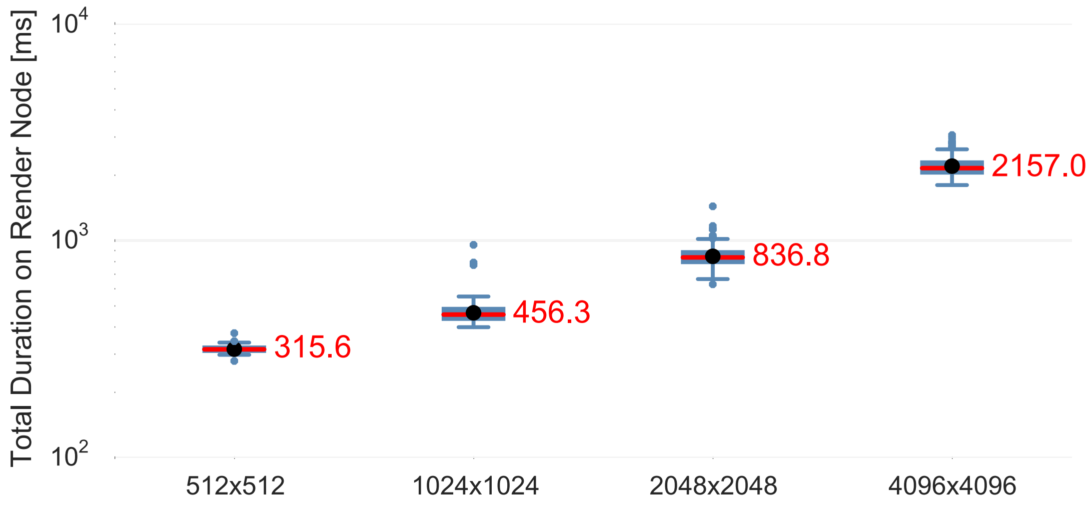
(a) Average merging durations on the display nodes.



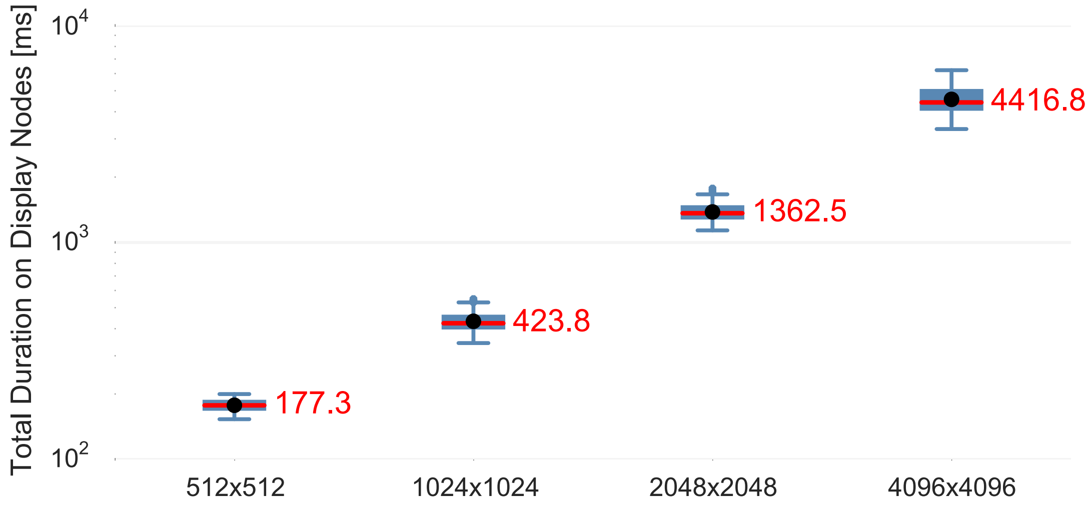
(b) Average compositing durations on the display nodes.

Figure 4.11 — Box plots of the average durations for (a) merging the received PPLLs and (b) compositing them into the final image on the display nodes. Note that the first received PPLLs are not merged but instead directly copied, as there are no entries to merge with.

Figure 4.11b shows the average durations for transferring and merging the incoming PPLL buffers with the local one on the display node. While the merge duration grows with increasing resolution, the merge rate is roughly constant throughout the results at ≈4000 fragments per ms.



(a) Average total durations on the render nodes.



(b) Average total durations on the display nodes.

Figure 4.12 — Box plots of the average total durations for one frame request on the (a) render nodes and (b) display nodes. Their sum is the total duration until a frame is completely rendered and transferred and can be shown to users.

Overall, the system behaves consistent with few outliers and minor swaying. While rendering and compositing of the PPLLs is fast (see Figures 4.8a and 4.11b), the transfer from the GPUs of the render nodes over the network to the display nodes and merging them there is noticeably slow (see Figures 4.8b, 4.10a, 4.10b and 4.11a) due to the data sizes and the communication pattern. On the one hand, this naïve and straightforward implementation of the PPLLs in a distributed visualization system achieves interactive frame rates only for an image resolution of 512×512 and 1024×1024 pixels (out of the tested resolutions). On the other hand, there is much room for improvement, for example using compression or better communication patterns. A major improvement could also be achieved by sending the PPLLs selectively to the corresponding display nodes, instead of scattering them to all.

4.3 Discussion

While PPLLs provide many advantages, they also introduce some challenges. The following discusses important aspects particularly in the context of distributed rendering.

4.3.1 Memory Requirements

The normal approach to PPLLs is to store all fragments in the attribute buffers. This is a causality dilemma, since it is a priori unknown how many fragments a scene contains but memory has to be allocated before rendering so they can be stored. Naïve implementations bulk allocate memory for a fixed number of fragments and discard the ones that would overflow. A better approach would be to render the scene in a first pass, counting the fragments (without actually storing them), and use that counter as upper limit for the memory allocation. An even more sophisticated approach would be to count the fragments for each pixel separately and to apply a prefix sum to these counts. This not only determines the total number of fragments but can also be used to partition the attribute buffers, so that no linked list is necessary. Instead, the fragments can be stored contiguously for each pixel, resulting in per-pixel arrays. While several advantages emerge with both of these approaches, they require at least one additional render pass and potentially more computations (e.g., for the prefix sum).

Yang et al. [2010] proposed rendering into PPLLs in two passes. The first pass renders only the opaque parts into a framebuffer with normal depth testing. The second pass renders only the semi-transparent parts into PPLLs without depth testing. In that pass, only those fragments are stored that are in front of the (front-most) opaque fragment. This reduces the number of stored fragments to a minimum since non-visible fragments are discarded. This optimization is not feasible for techniques that require all fragments, for example constructive solid geometry or depth-of-field [Kauker et al., 2013b].

Even though PPLLs are designed to store all fragments, implementations can limit the number of fragments per pixel, similar to the k-buffer [Bavoil et al., 2007]. Another approach would be to merge fragments when they are close. This works when no other fragment is emitted between them but leads to errors otherwise, since blending is not commutative. In addition, this might only be reasonable for depth and color (e.g., by averaging and blending the values) but might not be possible with other attributes like textures or object identifiers.

4.3.2 Size Reduction

PPLLs can be used for lossless distributed visualization with correct blending (see Section 4.1.2). Even though they are a sparse representation of the data, they have considerable memory requirements (see Section 4.3.1). While lossless compression can help in these cases, it is only practical if the data is compressed so much, that transfers and subsequent decompression are noticeably shorter. This is an issue in environments like the VVand due to the high bandwidth of the network when using InfiniBand.

4.3.3 Sorting

The PPLLs have to be sorted for correct blending both for local as well as distributed rendering. A straightforward implementation would sort and blend the fragments for one pixel in a single pass in a local array. On GPUs, this array is stored in the shared memory of the multiprocessor. Consequently, the size of the array influences the number of threads for each multiprocessor and in turn the frame rate. This is the fastest approach for small arrays, i.e., scenes with low depth complexity. For bigger scenes, it might be more reasonable to sort these globally in a separate pass, especially if they are then shown for multiple frames.

4.4 Summary

This chapter presented an A-buffer implementation for modern GPUs using per-pixel linked lists, offering a convenient approach to order-independent transparency for local and distributed rendering but they have high memory requirements. Large resolutions of the presented test case such as 2048×2048 and 4096×4096 pixels were rendered successfully by the evaluated, straightforward implementation on the VVand but interactive frame rates were achieved only by the lower tested resolutions. While this implementation already offers all the necessary functionality for advanced distributed rendering, there is a lot of room for improvement requiring additional engineering effort.

Part II

Resilience

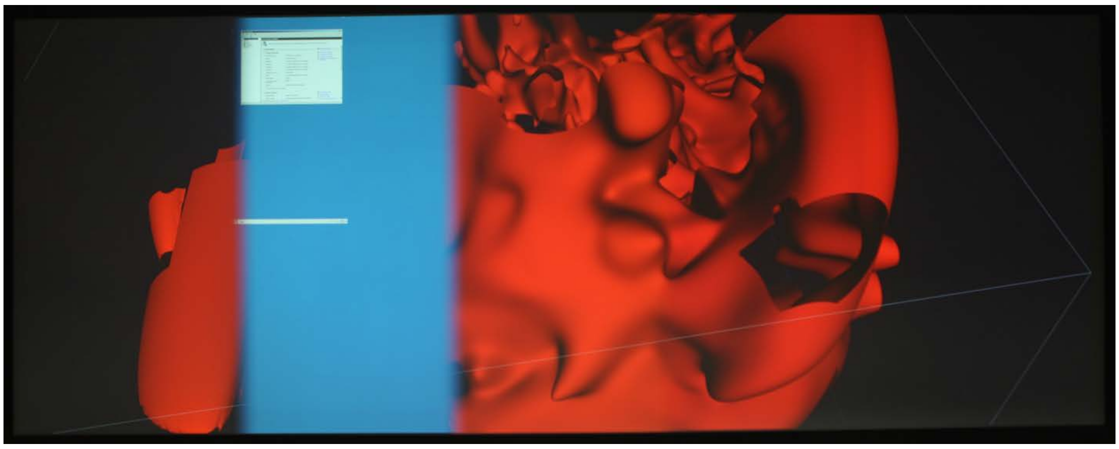


Figure II — Distributed visualization of iso-surfaces of a turbulent flow. The image is incomplete due to a crashed process and it is unclear whether the hole on the right is due to the data set or a failure.

With the introduction of the Computer Failure Data Repository, Schroeder and Gibson [2007] argue that fault tolerance is one of the hardest problems of *high performance computing* (HPC) and that supercomputers depend on reliability and availability. As the mean time between failures decreases with growing system sizes [Schroeder and Gibson, 2006; Xue et al., 2007] and failures also occur on small scale [Haque and Pande, 2010], algorithms as well as systems need to be improved not only with respect to scalability and maintainability, but increasingly for resilience.

Chapter 5 discusses fault tolerance in the context of distributed visualization. Existing methods and their applicability as well as additional strategies are discussed. Their implications and trade-offs are illustrated with several examples.

CHAPTER 5

Resilience in Distributed Visualization

Fault tolerance is well researched in general and many effective methods have been produced. Yet, applying traditional approaches to distributed visualization is antithetical. They require to restart an application or to block until a failure is resolved, in both cases disrupting application responsiveness and thus user experience. Furthermore, erroneous or missing data is often fatal for visualizations, resulting in program cancellation rather than incomplete or partial images, which are important to validate simulation parameters.

This chapter introduces and discusses existing fault tolerance concepts for *high performance computing* (HPC). A taxonomy of these methods is then extended with visualization-specific methods that are necessary for distributed visualization. This chapter is partially based on a previous publication [Panagiotidis et al., 2014b].

5.1 Fault Tolerance in High Performance Computing

In large-scale distributed systems, faults can occur at multiple locations and because of different causes, requiring multitudes of recovery methods. Physical faults of hardware or infrastructure often require technicians, leaving the affected parts of a computing environment unusable until they are repaired or replaced. Furthermore, these faults cannot be influenced because they happen unexpectedly due to frailty, wear and tear, or external effects (e.g., power outage, natural disasters). Interconnection problems are difficult to pinpoint since high latency, high load, or packet loss might be attributed to actual faults (defects in routers or cables), unusual events (e.g., fault recovery), or even shared use. Memory corruption can happen for both volatile and persistent storage. General software errors like crashes might be recoverable through various means, but often systems are so complex and tightly coupled that they crash without an opportunity for detection or recovery, for example when using fault-prone communication patterns. Software can also enter unusable or non-responsive states due to untested special cases or side effects, concurrency issues (e.g., race conditions, deadlocks), or degradation [Zhao et al., 2010].

Cappello [2009] provides a thorough overview of challenges and techniques for fault tolerance introducing three classes. *Failure Avoidance* methods try to predict and prevent errors based on analysis of log data and by migrating processes before they fail. *Failure Effects Avoidance* is a passive way of continuing the execution of an application during failures through resource replication, algorithm-based fault tolerance, and naturally fault-tolerant algorithms. *Failure Effects Repair* is the active mending of faults by re-executing the affected parts based on checkpoints (rollback-recovery) or by continuing execution because applications will recover later (forward recovery).

5.1.1 A Taxonomy of Fault Tolerance Methods

Egwutuoha et al. [2013] reviewed approaches for fault tolerance in high performance systems and classified them into a taxonomy with five categories: migration, redundancy, failure masking, failure semantics, and recovery methods. These categories are summarized in the following.

Migration methods prevent failures by moving processes or virtual machines away from nodes that are likely to fail. These preventive actions are based on heuristics derived from *RAS log files* (reliability, availability, serviceability) and are thus prone to false-positives and false-negatives, which may affect the system performance more than necessary. According to Cappello [2009], RAS logs need to be standardized so analysis tools can be developed easier and analysis techniques for these logs need to be improved.

Redundancy schemes use additional resources in a system to prevent failures. In case of *hardware redundancy*, additional physical components (nodes, cables) are employed and take over when failures occur. For *software redundancy*, multiple processes operate on the same task or computation. While hardware redundancy aims at keeping a system available, software redundancy can also be employed for verification. By computing a result in multiple ways — i.e., different processors, methods, or implementations — one can eliminate failing components or deviating results. In both cases, the necessary data needs to be *replicated* or be available otherwise during faults, for example through (fault-tolerant) centralized storage. Furthermore, when multiple data sources or results are available, a *quorum* or voting process is required to determine which data set to propagate or use in later steps.

Failure masking is a concept to maximize system availability by hiding failures in the system from observers, like clients of an *application programming interface* (API) or users of a distributed application. Masking methods use groups of redundant workers (i.e., processes or nodes) which appear as single entity to observers. In a *flat group*, masking is achieved through voting on computational results from each individual component, removing failing or unresponsive ones. Contrary, *hierarchical groups* are coordinated by one component that decides which worker replaces the failed one. Consequently, that coordinator is a single point of failure. While flat groups do not have a single point of failure, they introduce some overhead and latency due to the voting process.

Failure semantics describe how developers anticipate errors or failures, so that the system's behavior and output is defined for these predetermined cases. *Crash failure semantics* refer to a systems' inability to cope with failures, so correct operation is only ensured beforehand. *Omission failure semantics* denote that some messages are lost and delayed or corrupted with negligible probability. *Performance failure semantics* is a weaker form of omission failure semantics, where a system stays operational while messages are lost or delayed, and to a lesser extent corrupted. Even though some of these semantics can be achieved through defensive techniques, not all possible failures can be predicted and thus handled.

Recovery methods are the most investigated approach for fault-tolerance [Cappello, 2009] in HPC. *Forward recovery* relies on specifically designed algorithms that eventually recover from errors over time, putting emphasis on high availability of a system instead of immediate recovery. Typical methods for forward recovery include Hamming codes and erasure codes. *Rollback-recovery* methods are used to enable restarting a system from a previous coherent state and to resume operations from that state. To that end, *checkpoint and restart* protocols are widely used. Checkpoints — snapshots of the systems' state at a given time

— are stored persistently in periodic intervals or on demand. The amount of data ranges from minimal information used to derive other data to whole copies of a process' memory, opened file handles, and active network connections.

5.2 Strategies for Fault-Tolerant Distributed Visualization

While existing fault tolerance methods (see Section 5.1) aim to increase the long-term resilience of distributed computations, major concerns for visualization also comprise latency and interactivity. In other words, fault tolerance for visualization is already relevant for comparatively small data sizes and time scales. Furthermore, input data for the visualization is already either stored persistently or processed redundantly (assuming fault-tolerant simulation or storage). Consequently, fault tolerance in visualization is required for temporary failures so users can interact while recovery is in progress or to inspect partial data due to storage or computation failures in order to validate the data or the simulation parameters at least.

There are some approaches to fault tolerance in visualization applications. *CUMULVS* [Papadopoulos et al., 1998] — a middleware for viewable and steerable fields in large-scale distributed simulations — employs user-directed checkpointing of data specified by the application developers. A checkpoint daemon retrieves the most current coherent snapshot of the application state, and handles migration, resource management, and restarting of parallel applications after a failure. Samanta et al. [2001] replicate primitives of 3D geometry in an off-line pre-processing step to k nodes in a cluster of n nodes with $k \ll n$ to visualize large scenes in parallel. Gao et al. [2008] partition and distribute data redundantly with a master process scheduling visualization tasks into queues. Errors and timeouts are handled through requeueing with lower quality or resolution, while a checking task detects faulty nodes and excludes them if their results differ from the majority. This task is run at the beginning of a visualization session and randomly during its run time.

These approaches only apply well-known techniques (see Section 5.1) to specific visualization systems or problems but do not address the special environment and requirements of distributed visualization on large displays (see Figure II). In case of a temporary failure, the visualization pipeline is potentially stalled because nodes to the left of the fault in Figure 2.3 wait for acknowledgment of transfers while the nodes to the right do not receive any new data. This is particularly troublesome for display nodes, as usually a synchronized swap to a new frame is performed only when every display node signals that it received

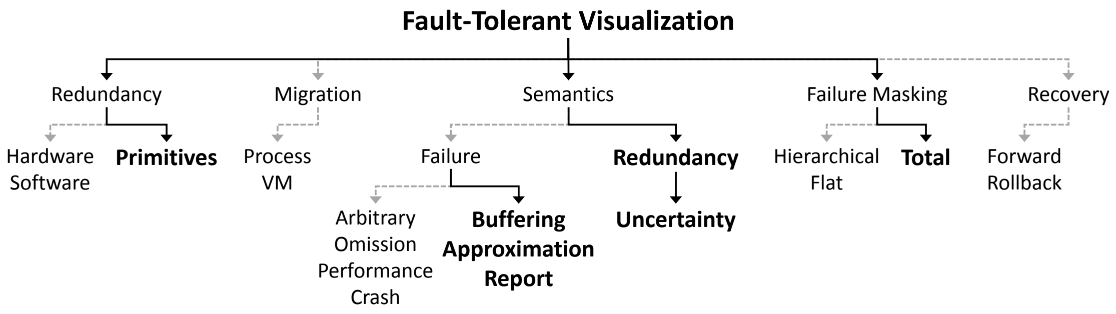


Figure 5.1 — A taxonomy for fault-tolerant visualization based on an existing taxonomy for fault tolerance in HPC [Egwutuoha et al., 2013] (dotted paths). Bold parts are strategies to improve resilience of distributed visualizations where established methods are unfeasible or impractical.

all necessary data. Yet, as long as render and display nodes are available, users should be able to interact with the newest data available to the render nodes in any way that does not perform steering, i.e., interactions that do not require or induce a data update.

Display nodes and their output devices cannot be made redundant easily since they are arranged in a certain topology, and physical placement as well as replacement are subject to many limitations. For one, restarting crashed display nodes disrupts the user experience and potentially blocks the synchronized frame swap. While migrating the processes from the crashed display node to spare nodes is possible, it is complicated by the fact that those spare nodes would need to be connected to similar output devices which is partially possible using KVM switches. Though multiple projectors for the same physical projection area as well as logical visualization domain can be used, they are substantially more expensive than spare nodes, need to be calibrated, cannot be run in parallel all the time, and have a noticeable delay from powering on until operating normally. Ultimately, defect monitors of an array cannot be redundant but need to be replaced due to the arrangement of the monitors.

The following strategies complement the taxonomy of Egwutuoha et al. [2013] to address the aforementioned challenges (see Figure 5.1). They include methods to visualize data with different temporal context as well as spatial gaps while also indicating these irregularities in the visual representation that is shown to users. A fault-tolerant distributed visualization should stay available and usable as long as possible even in the case of failures and during recovery of any part of the system. Therefore, *total failure masking* was introduced to emphasize the differences between interactive and non-interactive systems. Then, failure semantics were expanded and refined to account for user interaction during failures, missing data, and uncertainty due to fault tolerance methods. Finally, *primitives redundancy* was introduced as this intermediate data only occurs in visualization and requires multiple stages of redundancy in a distributed visualization.

5.2.1 Total Failure Masking

Fault tolerance methods strive to reduce the impact of failures for the system and users. For visualization on distributed displays, it is desirable that users ultimately do not notice that the system is recovering from a failure. Essentially, users should be able to use the system partially while it is recovering due to other mechanisms (e.g., checkpoint-restart).

Total failure masking is distinct from other masking types because it necessitates cooperation of different approaches on all parts of a visualization system. For one, all communication must be non-blocking or time out for unresponsive nodes. Nodes expecting data need to handle this as well, for example by utilizing older data. For some visualization methods, it might be possible to approximate missing parts from the neighborhood of erroneous regions, if it is available during a failure. Note that when employing previously cached results, approximations, or reconstructions to mend faults in a visualization, both the temporal discrepancies in the data and the speculative nature need to be presented and clarified to users (see Section 5.2.2).

5.2.2 Failure and Redundancy Semantics

According to Egwutuoha et al. [2013], "failure semantics refers to the different ways that a system designer anticipates the system can fail, along with failure handling strategies for each failure mode". Their definition only mentions failures, but for visualization — as post-processing and analysis of data — system failures as well as uncertainty during the visualization process needs to be expected. While employing fault tolerance methods in simulations allows for detection of problems and recovery, incomplete data (e.g., from crashes, before recovery) might still be meaningful for users, for example to verify simulation parameters or as partial result until the system recovers fully. As such, it is necessary that visualization methods and systems can deal with gaps in data, either through uncertainty semantics or through approximation, and go beyond crash semantics. Visualization systems should be responsive regardless of fault, at least presenting the last coherent images. Finally, the system state needs to be available for users to understand why a (distributed) visualization exhibits different behavior, for example during recovery.

Buffering semantics is the simplest form of dealing with faults in a distributed visualization. By using or emitting the latest results that are available, users can still interact with the data. This can be applied partially to the object or image space, for example meshes, volumes, or images. Additional metaphors are then necessary so users can understand that data from different times is mixed to provide a complete model or image, for example for partial validation or prevention of crashes. In this context it is important to track the *age* of data and images (e.g., in number of frames or time steps since the failure occurred) as indicator for validity and temporal coherence of the image that users interpret.

Approximation semantics are required to obtain a complete image when data is missing or outdated. In these cases, visualization methods need to approximate the necessary data from previous or neighboring data. Possible methods range from interpolation over image-space techniques like image inpainting [Bertalmio et al., 2000] or image editing [Pérez et al., 2003], to diffusion [Orzan et al., 2008], and mesh editing [Yu et al., 2004]. Users need to be made aware that the presented image is only partially correct and that some parts were approximated, so these parts need to be distinguishable or highlighted.

If the data is still available but the exact visualization is too time-consuming (e.g., with hard time constraints), it is also possible to adapt the visualization parameters for a less speculative approximation. As an example, using less integration steps or less accurate integration schemes may produce acceptable results within a given time. Rendering coarser meshes or in lower resolutions also decreases rendering time and can be post-processed with hardware acceleration on modern *graphics processing units* (GPUs), for example using texture filtering or tessellation shaders.

Uncertainty semantics are important in combination with redundancy and approximations. For redundancy methods, a quorum normally decides which result should be used. For visualization, it is sensible to pass on all of these results and show the diversity of the data, so users become aware of it and judge it accordingly. Therefore, all results and their deviation should be incorporated in a visualization. This is not an easy task, since visualization systems already struggle for perceptive channels to present data and adding uncertainty as another dimension intensifies this problem. Uncertainty visualization — a young research field with many open questions and challenges — deals with this specific issue of visually presenting deviations, inaccuracies, and errors [Johnson and Sanderson, 2003; Brodlie et al., 2012; Potter et al., 2012]. Promising approaches include displacement glyphs, geometry changes, bump maps, lighting attributes, and fuzzy surfaces.

Reporting semantics require information about the visualization application and its infrastructure. For the application, most of the information regarding fault tolerance — which nodes are failing, how are results masked, which recovery or approximation measures are currently in place — is already available but not exposed in a meaningful way. Approaches to include such information in user interfaces is often simplified through some form of progress bar without further detail. Information about the infrastructure (e.g., utilization of *processing units* (PUs) or the network) is often already available by means of traditional system monitoring (e.g., Nagios, Munin) but scattered in various places and disconnected from the visualization system. However, exactly this information is relevant for users operating large displays to judge whether a system is behaving normally, failing, or in the process of recovering. On this basis, users can decide how to interpret the current visualization — i.e., which parts are accurate or trustworthy — or whether a computation run experiences so many failures that it is either too inaccurate or too slow to be pursued further.

5.2.3 Redundancy

Typical interaction with a visualization includes rotating, zooming, panning, picking, etc. These interactions can be applied solely on the render nodes with the visualization primitives that were derived during mapping (see Section 2.1.1 and 2.1.3). To account for faults of the render nodes, the primitives need to be replicated to multiple render nodes.

Primitives redundancy is challenging and expensive but necessary for continued interaction with existing data. It requires redundancy on multiple stages of the distributed visualization. Compute nodes have to scatter their output data (i.e., input for the visualization) to multiple mapping nodes every time they produce new data (e.g., every time step of a simulation). The visualization primitives then need to be scattered to multiple render nodes. In doing so, requests for new frames (e.g., due to changed viewpoints) can be fulfilled by render nodes even during faults of compute and mapping nodes. Furthermore, since the input data is already replicated on multiple mapping nodes, the mapping itself can be modified during faults of compute nodes. This allows users to change the visualization method (e.g., from iso-surfaces to direct volume ray casting) while compute nodes recover from faults or work on the next time step.

Similar to software redundancy, placing visualization primitives on multiple render nodes enables verification of the correctness of the resulting image, for example through a quorum. This requires all render nodes to participate in each frame request and minimizes latency in the event of an error, since a replica of the image was already rendered. While this approach is optimal with regard to fault tolerance, it might not be feasible, for example when not enough resources are available for the desired frame rate. Therefore, a trade-off between performance and latency is necessary to allow for redundancy as well as interactivity and responsiveness.

Selective partial rendering is a strategy similar to hierarchical failure masking. Each set of visualization primitives is *authoritative* on a render node — which always renders these first and completely — and to multiple *auxiliary* nodes due to primitives redundancy. The auxiliary nodes only render redundant visualization primitives after they have rendered their authoritative ones. The auxiliary parts can also be rendered with reduced quality or only partially (e.g., compacting meshes, skipping triangles). Consequently, a complete image without overhead is available when no failure arises, given all primitives are authoritative on one render node.

Primitives redundancy compensates failures in object space through replication of the mapping output. For image-space partitioning, a similar type of redundancy is needed to guarantee a complete image every frame. Generally, visualization methods favor large coherent regions due to better saturation of PUs (e.g., less kernel launches) and interconnects (e.g., larger and fewer transfers) as well as spatial coherence in the data (e.g., gradients, cache locality). Thus, failures during rendering result in large fault regions in the final image. Depending on the actual size of these regions, it might be impossible to see high frequencies or global structures and tendencies. Furthermore, it may be time-consuming or difficult to approximate or reconstruct the fault region due to its size.

Mosaic rendering is a strategy for resilient image-space partitioning where each render node processes the same amount of pixels distributed over multiple, disconnected regions. Even though this requires more scattering of data, more and smaller buffers (affecting throughput), and more kernel launches per PU, it has some interesting properties. Since there are more tiles now, their redundancy can be controlled more fine-grained. Although there are more fault regions when a render node fails, each fault region is smaller, so high frequencies are less likely to be completely missing, thus allowing for perception of global structures. In the extreme case — fault regions the size of few pixels — faults might even result in visual noise, which can be mended through simple interpolation. For bigger fault regions, reconstruction and approximation is much more feasible now, for example through image inpainting [Bertalmio et al., 2000] or image editing [Pérez et al., 2003].

5.3 Discussion

Section 5.2 presented fault tolerance strategies at a high level and introduced some specific approaches. The following discusses example implementations in the context of a distributed visualization system on a large display (e.g., the VVand) which employs none of those strategies; as such, the following is neither a comprehensive nor an exhaustive description of how the strategies have to be realized. At some point during the run time of the system a fault occurs, i.e., parts of the current frame cannot be rendered, because either data is missing or the corresponding mapping node has crashed. The affected frame then needs to be salvaged and the system needs to adapt so that subsequent frames can be rendered successfully. Throughout all the time, the system status needs to be exposed, so users are aware that a failure happened and what parts of the system it affected.

The following scenarios are only feasible if the system survives the fault, i.e., it does not crash, stays responsive, does not wait indefinitely, and so on. In other words, total failure masking is a strict requirement. Furthermore, primitives redundancy can be achieved by scattering visualization primitives to multiple render nodes and is thus not discussed in detail.

5.3.1 Salvation and Recovery

To complete the current frame there are three options: (a) ignore the error, i.e., show the frame without the missing parts; (b) try to complete the frame, i.e., recover the missing parts; or (c) employ semantic strategies. Recovering missing parts is only possible when the relevant data is available on other nodes, for example due to redundancy schemes. It also increases the completion time for the frame and might delay subsequent frames (Section 5.3.6 discusses options to address this). If this is not possible or feasible, it might be beneficial to provide an approximation based on previous frames or neighboring regions in the current (incomplete) frame. This is discussed in Section 5.3.3 and Section 5.3.4.

For subsequent frames, the system needs to decide whether it continues with the strategy for the current frame (i.e., ignore, complete, provide semantics) or if it is possible to achieve a better state. If the data is available redundantly (i.e., on nodes that are still available), then a redistribution of the visualization tasks or a repartitioning of the object or image space can alleviate this fault. This might lead to a degradation of performance, since some nodes are now processing more or additional data, but might be ultimately better than recovering missing parts during rendering (i.e., via salvation of faulty frames). In cases where this is not possible, for example because the data is unavailable or a repartitioning is not possible, the user needs to be informed to take further action like restarting the system.

5.3.2 Reporting Semantics

Users need an indicator to judge whether a system is operating faultless or even operational or responsive at all. Traditional approaches for this include spinners and progress bars but these are impractical in setups like the VVand. While a progress bar might indicate that the system is still working, it does not convey what is currently being processed or whether faults occurred and how big their impact might be. Though this could be shown on demand as textual information, there would be a lot of overdraw for all the necessary information even on the VVand. Furthermore, a visual element that sticks out (e.g., a bright red circle) will attract more attention than an error message that has to be read.

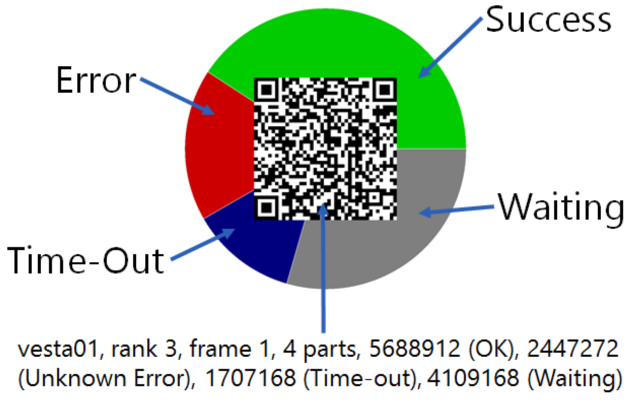

Success
Error
Waiting
Time-Out
vesta01, rank 3, frame 1, 4 parts, 5688912 (OK), 2447272 (Unknown Error), 1707168 (Time-out), 4109168 (Waiting)

Figure 5.2 — A progress glyph to show the state of a distributed visualization system in a compact way. The QR code contains additional information while the arcs represent parts of the object or image space that are processed distributedly.

Figure 5.2 shows a glyph that contains all relevant data for distributed visualizations. The basic concept here is to encode the status visually for every part of the system (e.g., using colored arcs). In addition, a QR code provides a textual description or a hyperlink to a monitoring website with additional information. This glyph can either be shown at a fixed location in image space (e.g., at the lower border of the display) or in object space at the actual place where the data would be shown (e.g., on the front face of the bounding box). Information can be encoded both with color as well as size, i.e., the arc size could represent the data or image size.

To provide such information to users it has to be polled from corresponding APIs. Though there are tools to retrieve the relevant information, they are usually not integrated into applications. Furthermore, application-specific metrics can provide useful information but have to be actively calculated (e.g., number of casted rays in volume ray casting). Chapter 7 discusses these issues in more detail for the example of parallel volume rendering.

5.3.3 Buffering Semantics

Developers of a distributed visualization system should anticipate that parts of the image might be missing due to failures. A naïve strategy is to cache received images (e.g., up to a threshold to not run out of memory) and provide missing parts from that cache. For situations with frame coherency (i.e., only few changes between subsequent frames), this can yield acceptable results but only to a certain degree.

Caching allows to replay previous frames but cannot provide different views of the data; for that, the render nodes would have to create new images. This issue can be alleviated with specialized image formats such as *Layered Depth Images* [Shade et al., 1998] or *Volumetric Depth Images* [Frey et al., 2013]. These allow for interaction with the images without the need to render the visualization primitives again. This is also possible when using *per-pixel linked lists* (PPLLs) (see Chapter 4), since they can be transformed to *Volumetric Depth Images*.

Techniques that retain multiple fragments or samples per pixel solve several problems but have several disadvantages. For one, keeping more information results in higher memory usage which cannot be easily alleviated even with compression and especially in high-bandwidth networks (see Section 4.3.1). Using them is also more complex than straightforward rendering both w.r.t. the necessary code as well as the additionally required data structures.

5.3.4 Approximation Semantics

An approximate reconstruction of the missing parts of an image can help to perceive global trends and structures at least. This is especially helpful when those parts cannot be calculated exactly (e.g., data unavailable) or timely (e.g., hard time constraints, exact calculation too time-consuming).

Figure 5.3 shows a *finite-time Lyapunov exponent* (FTLE) visualization (see Section 3.2.3) where some parts became unavailable due to failures. Image-space techniques like Poisson Image Editing [Pérez et al., 2003] can reconstruct these regions to some extent (see Figure 5.3c). Note that in this case the ridge in the upper left was reconstructed incorrectly; in particular, the reconstruction even depicts a very different behavior than actually prevalent in vector field. This technique might be improved by including previous frames for the reconstruction and not only the border.

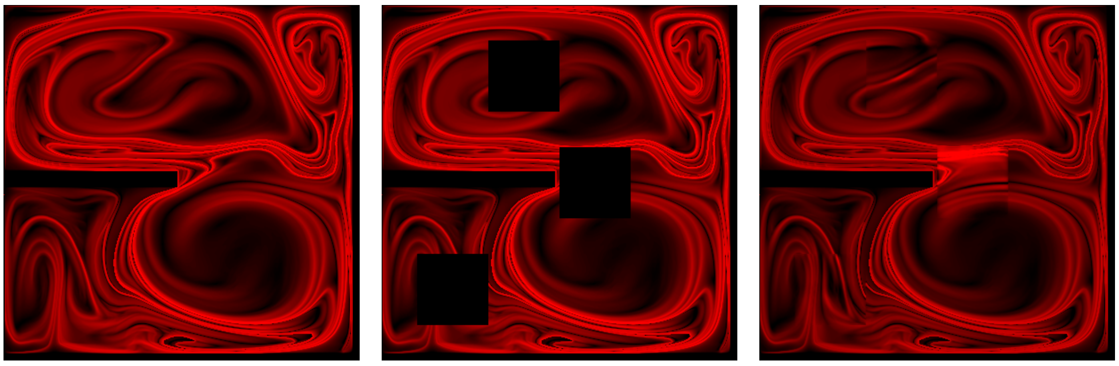
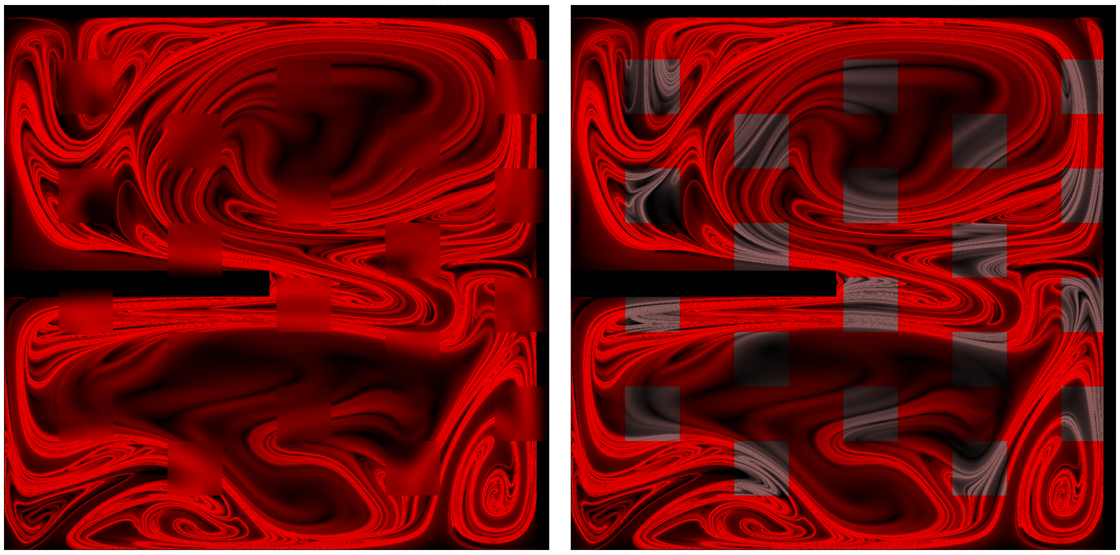

(a) Fault-free. (b) Parts missing. (c) Reconstruction.

Figure 5.3 — Failures during a flow visualization (a) result in missing parts of the final image (b). Such areas can be reconstructed with image-space techniques like Poisson Image Editing (c). The results can vary from a good approximation of the actual behavior in the vector field (middle region) to completely wrong (upper left, ridge should be continuous).

When the data is still available, but the exact calculation is too time-consuming (e.g., FTLE with long advection times or high image resolutions), it might be feasible to use the vector field as guidance for approximation schemes. In the case of FTLE, one could also calculate the missing parts in lower resolution or with shorter advection time as an approximation or employ a faster technique, for example line integral convolution.

5.3.5 Uncertainty Semantics

When mixing different frames (e.g., as salvation strategy to provide a complete image) or employing any kind of approximation strategy, it is important to inform users that the image parts result from different temporal contexts. Such techniques are actively researched (e.g., Johnson and Sanderson [2003] or Brodlie et al. [2012]) and domain dependent, but there are some obvious options that can be used in all cases, like blurring, desaturating, or fading out the affected parts (see Figure 5.4). Osorio and Brodlie [2009] discuss these and other techniques in greater detail in the context of flow visualization. These effects can utilize the *age* of the corresponding fault to increase the effect over time, since approximations gradually worsen and might become increasingly meaningless to users.



(a) Blurring. **(b)** Desaturation.

Figure 5.4 — Fault regions that were filled with parts from previous frames or approximations need to be distinguishable from the correct parts. This can be accomplished by (a) blurring or (b) desaturating those regions.

Uncertainty semantics can also be applied in non-failure cases of a distributed visualization when redundancy is employed, for example by using different hardware, APIs, or methods. In this case, all results could be shown, for example using comparative rendering techniques (see Section 4.1.2). One could also define an error metric (e.g., color difference, PSNR, *multiscale structural similarity* (MSSIM) [Wang et al., 2003]) and provide this either visually in the image (e.g., mapped to color) or in status indicators like the glyph in Section 5.3.2. Employing uncertainty visualization is challenging and one needs to be careful not to imply features that are not in the data through the modified visualization.

5.3.6 Selective Partial Rendering

Redundancy is one of the major concepts of fault tolerance. In the context of distributed visualization where data for a given object or image space is available to multiple render nodes, it needs to be decided which of those nodes should process the data and at which time point. Always rendering all of the available data increases the response time needlessly (especially when no failure occurs). Explicitly requesting for a faulty part also increases the response time in the event of a failure, but could ensure an otherwise optimal system.

Authoritative and *auxiliary* nodes are one way of deciding when to render what. As described in Section 5.2.3, the authoritative node is responsible for providing the highest quality of the partial image. Assuming a balanced system, all those nodes should finish roughly at the same time. After a short delay and if not canceled by the display nodes, the auxiliary nodes start rendering their redundant data at lower qualities or resolutions. As a result, the image will be completed with minimal delay, albeit with lower quality or upsampled in faulty areas.

Reversing this strategy is also possible, i.e., rendering the auxiliary parts first and the authoritative ones afterwards. Since the auxiliary parts should be rendered with settings that allow for a considerably faster rendering time, a preview of the complete image will be available quickly. The low quality parts of functional areas are then substituted with their high quality counterparts, leaving faulty regions as they were. The auxiliary nodes then decide on a new authoritative node. As with the other variant, the high quality rendering could be skipped, allowing users to 'fast-forward' through the data using the preview images.

On the one hand, rendering high quality parts first takes longer until images are provided to users and results in high latency when coping with faults. On the other hand, rendering low quality parts first might mislead users, for example when looking for certain patterns which do not emerge in low resolutions (e.g., fine *Lagrangian coherent structures* (LCS) [Haller, 2001] in FTLE). Switching automatically between both variants might be a feasible approach, for example by incorporating RAS logs or real-time monitoring information.

5.3.7 Mosaic Rendering

The object and image space is partitioned to speed-up the processing of large data sets. Through redundancy in the partitioning, faults can be handled without affecting the user experience. Mosaic rendering is a strategy for redundant image-space partitioning with the goal to increase the effectiveness of other fault tolerance methods.

The size of the partition parts is often chosen to increase spatial coherency during rendering and to decrease the overhead of kernel and shader launches as well as context switches. Consequently, failures then result in large fault regions, as shown in Figure 5.5a. Reconstructing these perfectly with redundant data almost doubles the render time, since a similar amount of time that was just completed in parallel for the non-faulty regions has to be expended again.

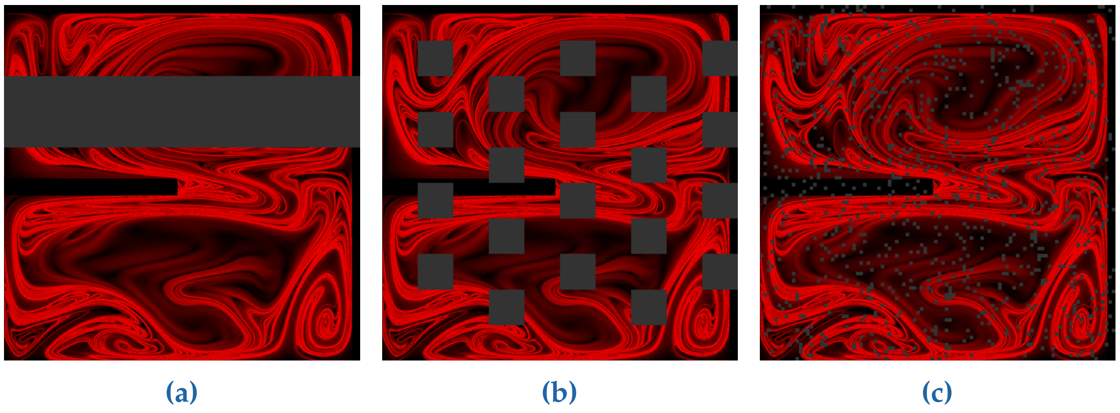


Figure 5.5 — Distributed visualization favors large, coherent, and continuous regions, resulting in corresponding fault regions which cannot be reconstructed well. Mosaic Rendering is a strategy that distributes the same number of total pixels into smaller regions. Depending on their distribution, it can be ensured that neighboring borders are available (b) for an upper bound of failing nodes. With shrinking size of the mosaic tiles, the faults can be perceived as noise (c) and could even be approximated by interpolation.

Even with selective partial rendering (see Section 5.3.6), only the delay can be reduced until the recovery begins. Furthermore, such large fault regions can be difficult if not impossible to approximate or might lead to imperfect or even wrong results (see Figure 5.3c).

The impact of failures can be reduced by using smaller, non-neighboring parts during the partitioning. For one, global structures and trends might still be perceivable without any fault tolerance method (see Figure 5.5b). With shrinking partition sizes, faults might be perceived as noise (see Figure 5.5c) and filled by interpolation between the neighbors.

For approximation, it is favorable to ensure that all neighbors of a fault region are still available for faults of up to k nodes. This can be modeled as a coloring problem by constructing a graph of the partitioning where vertices denote the parts of the partition and direct neighbors in the partition are connected with edges in the graph. The chromatic number χ of that graph is then the number of nodes to store data redundantly.

While mosaic rendering can improve approximations of faulty regions, it can result in non-negligible overhead during rendering due to the discontinuity of the regions processed by a render node: more buffers are required (resulting in more data transfers and meta data), more draw calls and kernels need to be submitted, there is less spatial coherence during rendering, and caching might be less effective.

5.4 Summary

This chapter discussed the applicability of fault tolerance methods for distributed visualization environments like the VVand and the resulting necessity for additional strategies that are specific to this context. Several new strategies were introduced, integrated into an existing taxonomy, and illustrated with several examples to discuss their implications and trade-offs.

Part III

Exploratory Analysis

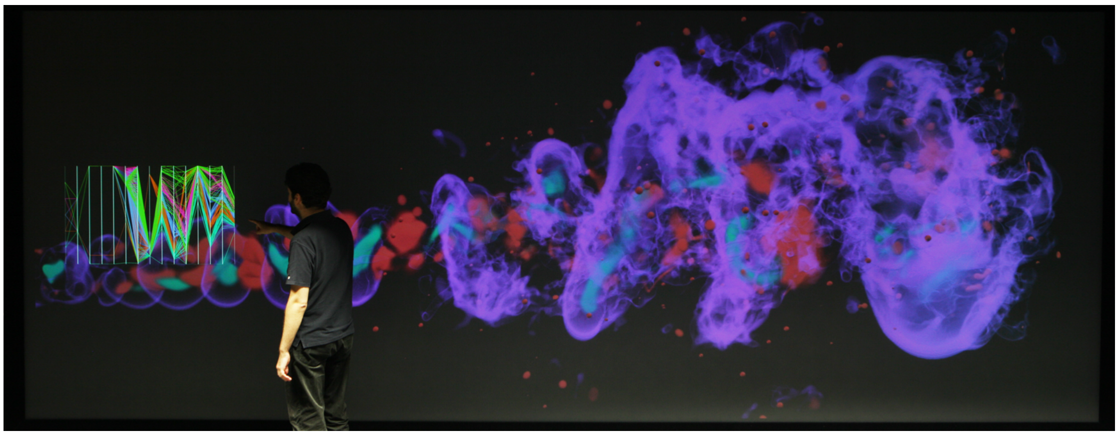


Figure III — Real-time analysis and tuning of parallel, interactive volume visualization on the VVand.

Interactive visualization systems are complex since they consist of many subcomponents that load and filter raw data, derive visual primitives, create 2D images, and respond to user interactions. These tasks become even more complex for visualization on large, high-resolution displays (see Section 2.3), which involve multiple nodes that need to cooperate (see Section 2.1.3) to create a coherent view on the data. Building and understanding these visualization systems is challenging, because typical approaches to monitoring and analysis are targeted at infrastructure (e.g., networks, applications) or non-interactive use cases. Previous analysis approaches also lack the context, i.e., the images and interactions, which are crucial to understand the behavior of the hardware and the algorithms. The applications themselves often also do not provide means for analyzing or tuning them. Finally, this is a meta-problem: large and complex raw data is visualized for a better understanding of it, which yields multi-dimensional and time-varying data from the visualization application, which in turn has to be visualized to analyze the application itself. This performance data of visualization systems is not necessarily flat but may be comprised of relations and complex interaction between its entities. Such dense network data is difficult to analyze due to cluttered visualizations or insufficient interaction techniques.

The following two chapters discuss approaches to address these challenges. Chapter 6 presents two methods for local graph exploration of dense and cluttered graphs. Chapter 7 presents an exploratory approach for analysis and tuning of volume visualization on the VVand.

CHAPTER 6

Local Graph Exploration

The relations between data elements are an important aspect of information spaces. Such relations are often modeled as graphs where nodes represent data elements and edges indicate given relations between these elements. Formally, a graph is defined as a tuple $G = (V, E)$ consisting of n vertices $V = \{v_1, \ldots, v_n\}$. The set of edges $E \subseteq V \times V$ denotes the relations between vertices, while their weight $w : E \longrightarrow \mathbb{R}$ describes an attribute of the relations.

Graphs are often visualized to provide a better understanding of the relations between vertices. Graph drawing is a well-established field, with many ways to visualize relational data and networks directly [Battista et al., 1998; Kaufmann and Wagner, 2001] or derived from multivariate, temporal data [Archambault et al., 2014]. A common visualization of graphs is the node-link diagram, where *nodes* and *links* represent the vertices and edges of the graph, respectively. Methods for node-link diagrams often employ a force-directed approach [Fruchterman and Reingold, 1991] or stress majorization [Gansner et al., 2005], often motivated by cognitive and aesthetic reasons [Purchase et al., 1996; Ware et al., 2002], to lay out the nodes. Despite sophisticated algorithms and optimizations, these layouts still result in local and global hairball-like structures for complex and large data. This visual clutter is undesirable because it leads to degradation of performance at some exploration and analysis tasks [Rosenholtz et al., 2005], for example due to many link crossings.

Improving graph layouts can ease problems like overdraw of the background, occluded nodes and edges, and visual clutter (see Ellis and Dix [2007] for a comprehensive listing of techniques to reduce such clutter). *Confluent drawing* [Dickerson et al., 2004] merges edges in order to draw non-planar graphs in a planar way. Holten [2006] suggested bending (non-hierarchical) adjacency edges into splines that are routed to the least common ancestor of the connected nodes according to the hierarchy for generating edge bundles or employing a force-directed approach [Holten and van Wijk, 2009]. *NodeTrix* [Henry et al., 2007] groups nodes to matrices, thereby combining their edges. Geometry-based edge clustering, as introduced by Cui et al. [2008], combines edges by routing them through common control points to increase comprehensibility. All of these methods try to show global connectivity trends, at the same time making it more difficult or even impossible to determine which nodes are actually connected, because multiple edges are drawn on top of each other. Several splatting approaches deviate from the typical link drawing and instead use densities to depict the relations in the data [van Liere and de Leeuw, 2003; Burch et al., 2011; Hurter et al., 2012]. Level-of-detail approaches combine link accumulation with density-based node aggregation [Zinsmaier et al., 2012]. Besides the discussion and analysis in the respective publications, there are many evaluations on the topic of modifying links for graph visualization [Holten and Wijk, 2009; Burch et al., 2012].

Most of these approaches aim at revealing the background, thereby making labels and other information readable, or target at modifying link clusters or their representation, respectively. As a side effect, they introduce ambiguities when links are drawn on top of each other, making them indistinguishable. Links also become harder to follow when routed through common points or the nodes they connect cannot be discerned. In many cases, edges themselves represent and contain important information which is now no longer visible or easily accessible.

This chapter presents two approaches to address these challenges. *EdgeAnalyzer* employs a lens metaphor for exploration of edges by grouping links inside the lens and visualizing those groups independently from the node-link diagram. *Graph Metric Views* combine histograms of various metrics of a data set with a node-link diagram of relations of the same data set — similar to marginal histograms aligned with 2D scatter plots — to show additional information that cannot be integrated into the node-link diagram easily. This chapter is partially based on previous publications [Panagiotidis et al., 2011a, 2014a].

6.1 Exploration of Bundled Edges

Edge bundling techniques reduce clutter, show global trends, and unveil features in the connectivity of graphs. At the same time, it becomes tedious and challenging to follow distinct links, find the nodes they are connecting, or extract information they contain. Using certain interaction techniques allows to retain the advantages of edge bundling as well as exploring the now ambiguous, overlapping links.

Holten [2006] proposed to select links by intersecting them with a drawn line. Thereby, all non-crossing links are removed from the view and the bundling effect can be disabled to see direct connectivity without clutter and bundling ambiguity. This approach is rather intuitive, but meaningful interaction with edges requires previous bundling. The selection is only based on the geometric property of links and therefore does not facilitate selection of multiple links according to aspects besides hierarchy or geometry.

FromDaDy [Hurter et al., 2009] is a tool for analyzing aircraft trajectories. While trajectories are not links, interacting with them is quite similar. Here, a circular brush with adjustable size is used to select or de-select trajectories. Multiple brushing operations are supported before the analyst can move all selected trajectories to a new view to reduce clutter. To select elements by non-spatial meta data the layout of the trajectories can also be changed to altitude, time, or speed. However, to select all trajectories with a certain altitude, a layer across the whole screen has to be brushed. Furthermore, extracted trajectories are spatially separated from their context.

EdgeLens [Wong et al., 2003] and *EdgePlucking* [Wong and Carpendale, 2007] are two approaches to explore regions containing ambiguities due to link congestion. Both approaches rely on bending links 'out of the way' to reveal underlying nodes and make links distinguishable. *EdgeLens* is a force-based technique where multiple small lenses repel links and reduce the opacity of nearby links. With *EdgePlucking* analysts bend links like guitar strings. All links along a brushing operation follow the movement of the mouse cursor. Both techniques are well suited for examining local connectivity that is hidden under larger or longer links, however, they do not allow selection of links and subsequent interaction with them.

Tominski et al. [2006] present lenses that modify the visualization and the layout of a node-link diagram locally. The *local edge lens* hides links that are not connected to nodes outside of the lens. The *(generalized) bring neighbor lens* attracts neighbors of nodes inside of the lens. The *composite lens* combines these effects, as their user study found that it is beneficial. These lenses allow for local exploration of dense regions in hierarchies and graph visualization but only work on the visual level. Further grouping of the edges according to other criteria and different visualization of the links in the focus region is not considered.

The use of lens metaphors has been applied to a variety of application domains. Furnas [1986] illustrates the importance of nearby context and uses the analogy to optical 'fisheye' lenses. *Magic Lenses* [Bier et al., 1993] are additional see-through interfaces between the application and the cursor providing contextual interaction options to modify data and the appearance of focused objects. Phelps and Wilensky [2001] developed a system similar to *Magic Lenses* but put a stronger focus on the separation of functionality and a modularized architecture. *Semantic Lenses* [Rotard et al., 2007] are a focus+context technique that use a lens metaphor to show additional semantic information about focused parts of a document.

Some shortcomings of the mentioned works are addressed in *EdgeAnalyzer* [Panagiotidis et al., 2011a], a focus+context interaction paradigm for direct interaction with links in densely connected graphs. A lens metaphor is used to let analysts inspect single or multiple links by focusing on them. These focused links can be grouped according to different meta data aspects or geometric criteria during interaction. The grouping mechanisms allow for arranging edges according to an exchangeable similarity measure. This reduces the number of interaction elements and steps by operating on sets of edges that are determined independently of the graph layout or edge bundling. Since regions of a view that are investigated with the lens can be ambiguous or cluttered, alternative visual representations for edge groups are provided.

6.1.1 Advanced Edge Interaction

EdgeAnalyzer is designed to enable interaction with large numbers of crossing links in dense graphs as well as with indistinguishable (overlapping) links in bundled graphs. A lens metaphor is employed since directly selecting individual links with a pointing device is challenging and error-prone in such graphs. Using *EdgeAnalyzer* is a multi-stage process that follows the visual analytics mantra stated by Keim et al. [2006]: analyze first; show the important; zoom, filter and analyze further; details on demand.

The basis of the interaction is a node-link diagram, showing the important data of a previously executed analysis step. To analyze the data within the underlying visualization further, analysts drag a lens (see Figure 6.1) over it, marking all links that intersect the lens as focused. The global context, here the node-link diagram, stays visible for navigation and interpretation, as motivated by Furnas [1986]. The size of the lens can be changed to adjust the area of focus according to the analysts' needs. Then analysts can sort and filter focused

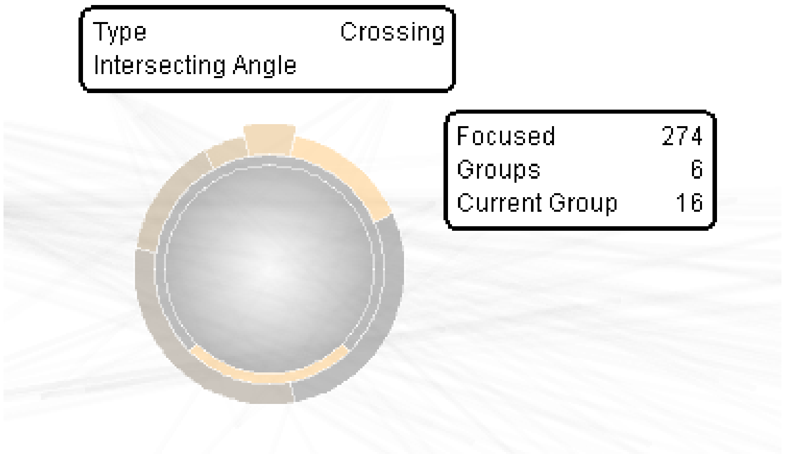
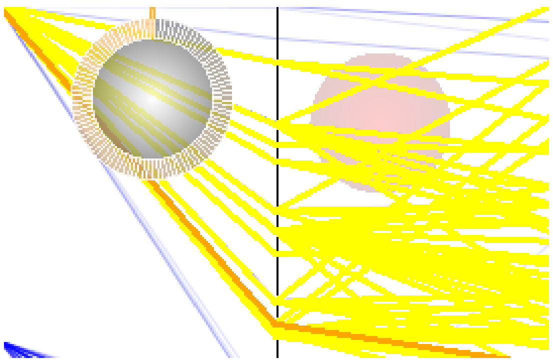


Figure 6.1 — The edge lens and the arc wheel with tooltips that list the edge grouping hierarchy and details about the edges and groups inside the lens.

links by arranging them in groups according to selectable and interchangeable criteria. These grouping criteria can incorporate any data available, such as geometric properties (e.g., angles, lengths, intersections) or meta data (e.g., names, dates). A visual representation for each edge group is drawn within the focus region.

Analysts can interact with fewer elements — instead of all edges within the focus region at once — by combining edge grouping and visual representations for edge groups. Afterwards, they can iteratively inspect edge groups and regroup edges of interest, effectively drilling down on the edges hierarchically. Finally, the analyst selects edges from edge groups for additional application-dependent actions, for example showing detail views or updating other visualizations. During this interaction, additional information can be shown in a tooltip near the lens, such as the number of focused edges and the sizes of edge groups. At any time, the interaction methods of *EdgeAnalyzer* and the underlying visualization can be disabled to resolve conflicts (e.g., same hotkeys) and all lenses can be hidden.

A modular approach is followed throughout *EdgeAnalyzer*: (a) the lens metaphor to select multiple links in a focus region without losing the global context, (b) grouping methods that arrange focused edges into sets of similar edges according to given grouping criteria, thereby filtering irrelevant edges, and (c) visual representations for edge groups, providing also an alternative view of the edges within the focus region. In terms of implementation, each part is treated as a black box, i.e., defined by interfaces and exchanging only minimal intermediate data. This way, additional lens shapes, grouping methods, and visual representations can be introduced easily to support different problem domains and data sets. Analysts can choose from available implementations of these parts during run time independently from each other, enabling them to use the most suitable analysis tools.


(a) Two top-level lenses (Boolean OR).

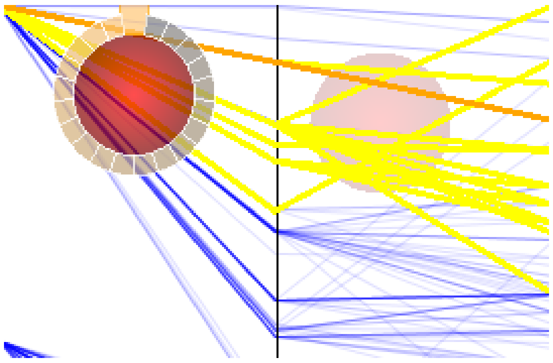

(b) Child and parent lens (Boolean AND).

Figure 6.2 — Multiple lenses in a parallel coordinates plot. Focused poly-lines are depicted in yellow. The right lens focuses the same poly-lines in both figures. (a) Top-level lenses are independent of each other. (b) The child lens (left) only affects those poly-lines that cross its parent lens (right). This allows for Boolean operations on the poly-lines, i.e., top-level lenses realize an *OR* and child lenses an *AND*, respectively.

6.1.2 Focusing Edges Using a Lens Metaphor

Analysts move a lens (see Figure 6.1) that is drawn semi-transparently on top of an existing node-link diagram. While moving or resizing the lens, all intersecting or completely enclosed links are focused. This intersection test is generic by approximating the shape of the links with straight multiple line segments. As such, only these line segments need to be intersected with the shape of the lens. This approach allows any shape for the lens and links. A circular shape is the default because it resembles lenses known from everyday life and as such might be familiar to users. Intersection points with the outline of the lens are stored per line segment and can be used to group edges and to create visual representations of edge groups.

Lenses can be moved independently of each other. Top-level lenses consider all links for the intersection test and are colored differently. Child lenses only consider focused edges of their direct parent and use the same color. Figure 6.2 shows an example: the right lens (pale red) focuses the same edges in both views and the left lens is the active one. In Figure 6.2a, the left lens is a child of the right lens, therefore only those edges are focused that cross both lenses. In Figure 6.2b, all edges are focused that cross the left lens, because it is a top-level lens and independent of the right lens.

6.1.3 Grouping Edges

Interacting with edges becomes challenging when analysts are confronted with many edges while inspecting focus regions. In order to reduce the number of interaction elements, focused edges are grouped into sets according to a similarity measure which may use any information in the underlying data model. Since non-geometric properties (e.g., names, dates) of the focused edges can be used, this also helps to indicate aspects of the data that were not visible in the node-link diagram. As a side effect, this makes grouping methods dependent on the problem domain, used data set, and geometry of the node-link diagram.

If analysts move the lens quickly, they are probably navigating and not inspecting details [Gutwin, 2002]. Therefore, the grouping effect of the lens can be delayed by a configurable amount of time (normally in the order of milliseconds) after the lens stands still. Once edges have been grouped, analysts can inspect each edge group one after another. At the same time, various information about the generated groups and the contained edges is provided. Edges within a group can be regrouped using different criteria or they can be selected and exported for further analysis in an external tool. This iterative regrouping establishes a hierarchy, allowing analysts to drill down on edges in order to sort and to inspect them according to different aspects of the data.

Unconditional Grouping

A simple grouping method is to unconditionally build one group containing all edges or a group for each edge. This allows analysts to distinguish edges in the focus region and the connecting parts of the node-link diagram. Further interaction on all focused edges, like highlighting in other views, is easily possible with this kind of grouping.

Clustering

Edge groups can be derived using a distance function and a clustering algorithm. Depending on the distance function, this approach groups edges according to a variety of attributes. For example, grouping edges according to their origin could be accomplished using the Euclidean distance between the starting points of two edges. The angles between an edge and the lens can be compared to group edges with similar directions within the focus region. K-means clustering [MacQueen, 1967] then assigns the edges into groups. Finding a good value for k could be accomplished by iteratively increasing k until the sum of the clusters' variances does not change significantly.

Geometric Grouping

This grouping method sorts edges into the same group, if a common geometric property is below a pre-defined threshold. For example, if the intersection points of two edges are close enough or the angles between their intersecting line segments are small enough, then those two edges are grouped together.

Hierarchical Grouping

In hierarchical data sets that additionally contain non-hierarchical edges, it can be observed that edges connecting nodes of one sub-hierarchy with nodes of another sub-hierarchy have a common path, for example due to edge bundling. If such a common path of two edges exceeds a configurable length, then those two edges can be considered similar and are grouped together. Separation of flows between different subtrees is an example usage for this grouping method.

Grouping based on Meta Data

Edges can be grouped according to comparable information, if the nodes or edges in the node-link diagram contain such information. For example, if every edge contains a label, it is possible to put all edges with the same label in the same group. This can be used on any kind of information, as long as equality between two such items can be determined.

6.1.4 Hierarchical Edge Groups

Analysts may group edges subsequently, allowing a drill down on the edges, and creating a hierarchy of edge groups. New hierarchy levels can be created using the selected edge group, and going back to the previous hierarchy levels disposes subsidiary levels. Only edge groups of the most recent hierarchy level can be inspected. When creating a new hierarchy, the edges inside the currently selected edge group become the new set of focused edges. This allows the recursive application of grouping and visual representation methods. Moving the lens disposes its complete hierarchy, since subsequent grouping in one region might not be meaningful in another region. Information about the edge group hierarchy is shown in a tooltip above the lens and in the surrounding arc wheel (see Figure 6.1). The tooltip lists from bottom to top the most recently used grouping and information about the currently selected edge group.

6.1.5 Visual Representation of Edge Groups

Single links and their characteristics are poorly visible to analysts if there is much overlapping or crossing of links in the focused region. Furthermore, analysts need a visual clue of edge groups as interaction element. Therefore, visual

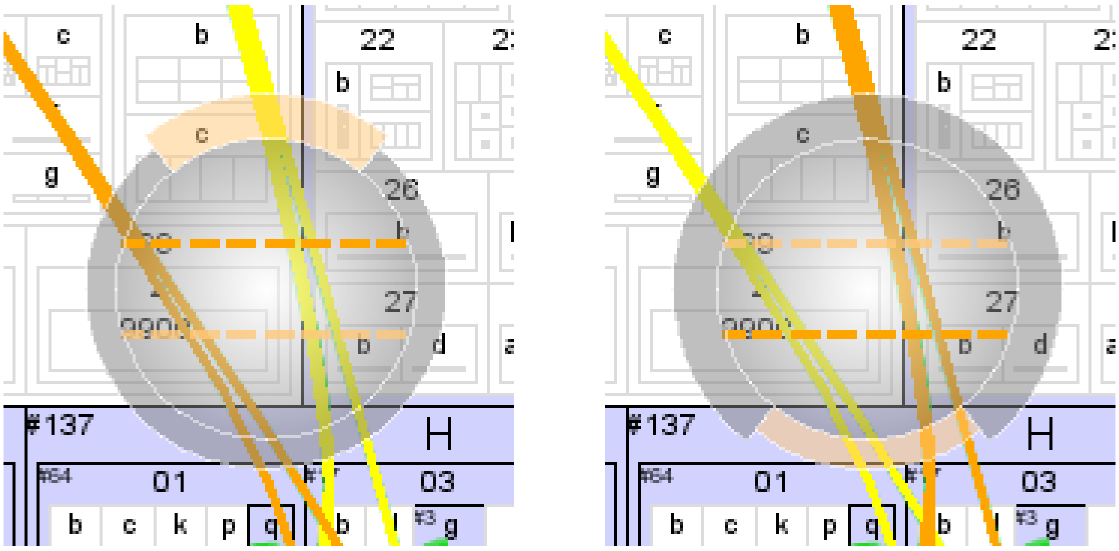


Figure 6.3 — Similar to rows in tables, each dashed horizontal line represents one edge group. Selecting an edge group highlights the corresponding links in the node-link diagram, their representation inside the lens (here: the dashed lines), and the corresponding segment of the arc wheel.

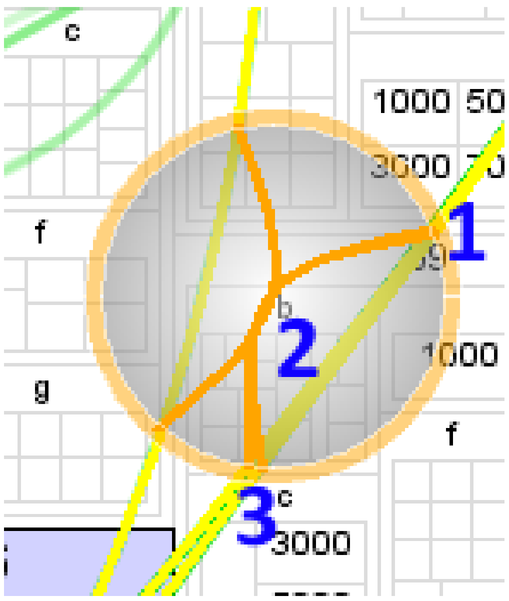
representations of edge groups are drawn on top of the lens without changing the global layout. Analysts can switch between a set of implemented visual representations during run time independently from the grouping mechanism actually used. Selecting an edge group highlights all edges in that group and its corresponding visual representation (see Figure 6.3).

Tabular / Rows

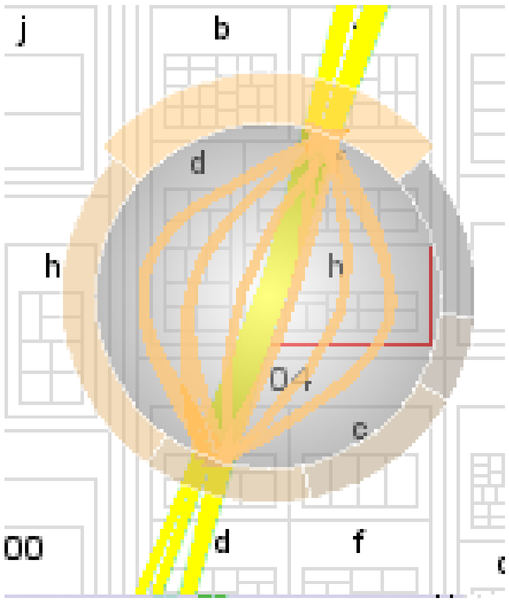
Each edge group is shown as a dashed horizontal line inside the lens (see Figure 6.3), similar to how data elements are arranged in tables. This is useful to find areas with few or many edge groups quickly. As a downside, these lines can clutter up the lens, ultimately overdrawing the node-link diagram in the focus region completely.

Local Rebundling

Links that are drawn on top of each other, for example due to edge bundling, are difficult to discern. In this case, the links can be locally re-bundled by routing edges of the same edge group through a common point (see Figure 6.4 left, all links belong to one group) and repelling distinct edge groups from each other (see Figure 6.4 right, multiple edge groups) at the same time. This is achieved by distributing the common points for edge groups on a line perpendicular to the principal direction of all focused edges. A spline is drawn for every link from the first intersection point (1) through the common point of its group (2) to the



(a) Same edge group.



(b) Different edge groups.

Figure 6.4 — Local re-bundling of edges that are drawn on top of each other. A spline is drawn for every edge from the first intersection point (1) through the common point of its group (2) to the last intersection point (3). Edges of the (a) same edge group pass through a common point while (b) edges of different groups repel each other (i.e., each edge group has its own common point).

last intersection point (3). Provided that a meaningful principle direction can be derived, this approach fans out the edge groups evenly and helps analysts to better match links to their groups.

Arc Wheel

To avoid additional overdraw in the center of the focus region, each grouping hierarchy level can be drawn as rings that grow away from the lens, similar to the approach presented by Stasko and Zhang [2000]. Every ring is split into as many arcs as there are edge groups (see Figure 6.1). The size of an arc depends on the number of edges within its group relative to the total number of focused edges. The arc of the currently selected group is situated on the top of the circle. The segments are colored according to an interpolated palette that fades from the edge group color to a medium gray, making it easier to differentiate between the first and last group. By stacking multiple arc wheels, this method is also capable of visualizing edge group hierarchies.

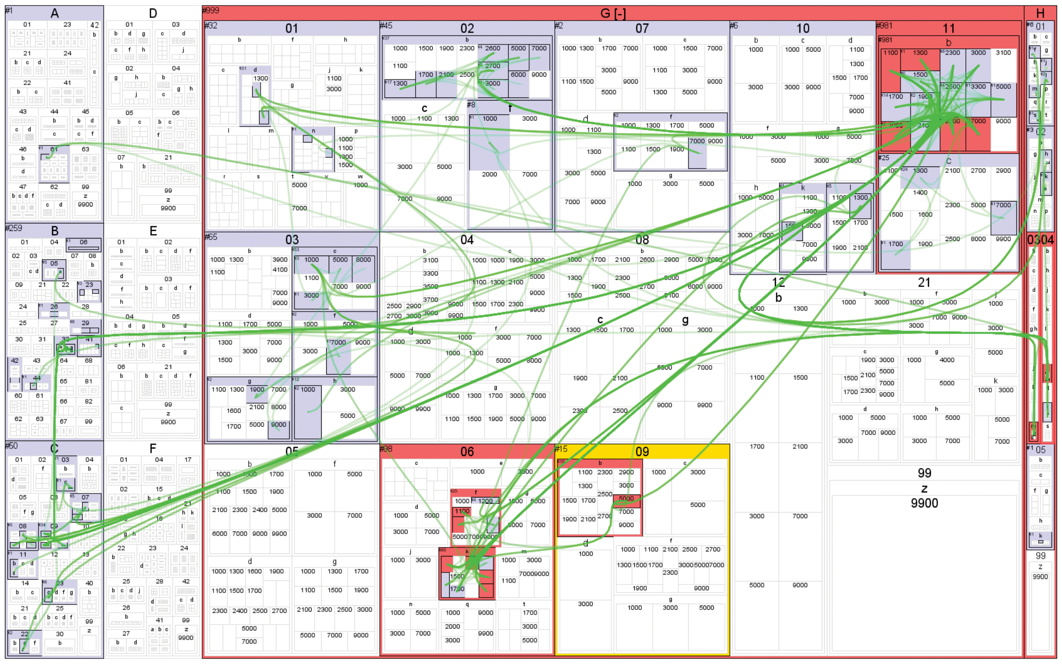


Figure 6.5 — The International Patent Classification (IPC)[1] visualized as tree map. Bundled edges drawn on top depict patents that are co-classified in the two IPC classes that the spline connects.

6.1.6 Evaluation

EdgeAnalyzer has been evaluated in the intellectual property domain, where finding as much prior art as possible is an important task. This need for completeness results in a highly iterative process between analyzing patent documents in a result set and adjusting search queries accordingly. *PatViz* [Koch et al., 2009] is a framework to support this iterative patent search. Analysts define search queries and inspect the results in multiple coordinated views. These show visualizations of various patent properties, like the geographic location of the issuer on a worldmap or their membership w.r.t. the International Patent Classification (IPC)[1] as a tree-map. On top of this tree-map, bundled edges between nodes can be drawn for patents that are classified in multiple IPC classes (green splines in Figure 6.5). In this context, *EdgeAnalyzer* has been integrated to explore edge bundles and the corresponding patent documents.

Informal User Study

An informal think aloud user study was conducted to evaluate the concepts of *EdgeAnalyzer*. Due to the large number of possible combinations of parameters and methods, only a single movable and resizable lens was available and the arc wheel was disabled.

[1]http://www.wipo.int/classifications/ipc/en/

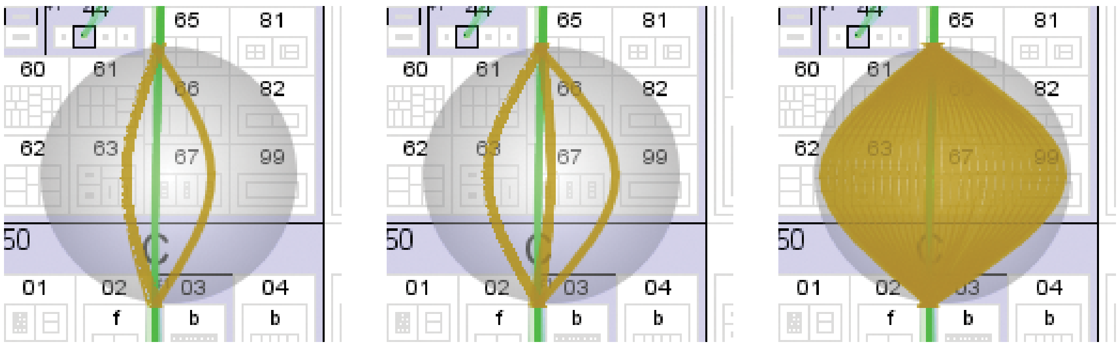


Figure 6.6 — Participants of the user study could switch between these edge groups.

Eleven participants, with no previous knowledge about *EdgeAnalyzer*, were asked to perform two tasks in a patent co-classification analysis. The data set at hand consisted of 1000 patents and 169 co-classification relations. Participants were given a very brief introduction to the *PatViz* system and the IPC schema. Then they were given a few minutes to use the lens and ask basic questions regarding its usage until they felt confident enough to start. None of the participants took more than two minutes for this free training phase.

The first task was to explore a dense and bundled edge cluster that was partially collapsed to a single line, a common situation in dense node-link diagrams with bundled edges. During this task, edge groups were always depicted using the local re-bundling method. Participants could switch between these grouping mechanisms (see Figure 6.6): grouping by common points of intersecting line segments, grouping by angle between intersecting line segments, and unconditional grouping. Three participants found that unconditional grouping is superior in this particular task, possibly because the large number of focused edges was instantly visible since for every edge group (in this case equivalent to an edge) a spline was drawn. Participants commented that the geometric grouping methods seemed to yield unreliable and incomprehensible results. Nevertheless, two participants preferred them as they produced fewer interaction elements. The participants also mentioned that multiple lenses could have been useful for this task.

The second task required participants to find the tree-map node with the most connections using the row representation of edge groups (see Figure 6.3). All participants found the right answer and when asked about their approach, eight participants answered that they first used the opacity of edges as a starting point and then used the lens for further inspection. Two were surprised how deceptive the opacity of the edge bundles was, since no visual difference could be perceived when the amount of edges exceeded a certain number.

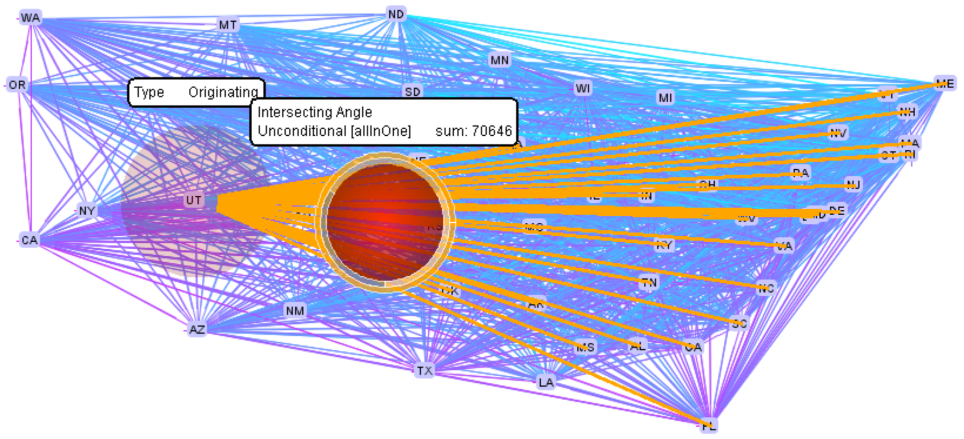


Figure 6.7 — Multiple EdgeAnalyzer lenses in the migration graph with unbundled links.

After completing the two tasks, the participants were asked to state their opinion. All participants appreciated *EdgeAnalyzer* as interaction technique for edge bundles. Five participants could also imagine to select edges by stretching some shape over the edges instead of moving a fixed shape (similar to selection in Holten [2006]). The option to change the size of the lens was used frequently. Seven participants appreciated that the lens stays visible after a focus region has been selected. Two other participants disliked the overdraw of the focus region by the lens and the visual representation of edge groups. The tooltip was highly valued by all participants. Three participants wanted to move the tooltip. Seven participants stated that this interaction technique was confusing at first and took some time to get used to.

As consequences from this study, more meta data based grouping mechanisms were implemented, since users expect a comprehensible, predictable, and repeatable outcome when grouping edges. By default, the arc wheel is now enabled and unconditional grouping and local re-bundling is used. Furthermore, focused edges are always highlighted entirely (and not only inside the lens), all non-focused edges can be hidden to further reduce clutter, and edge grouping is not delayed since participants mentioned they would have preferred instantaneous feedback.

Arbitrary Graphs

The major concept of *EdgeAnalyzer* is the selection of multiple entities in visualizations, grouping them to (non-visual) criteria, and subsequent interaction with the resulting groups. While it was originally designed to allow for interaction on edge bundles, *EdgeAnalyzer* also allows for sophisticated exploration of node-link diagrams with straight links and other line-based visualizations.

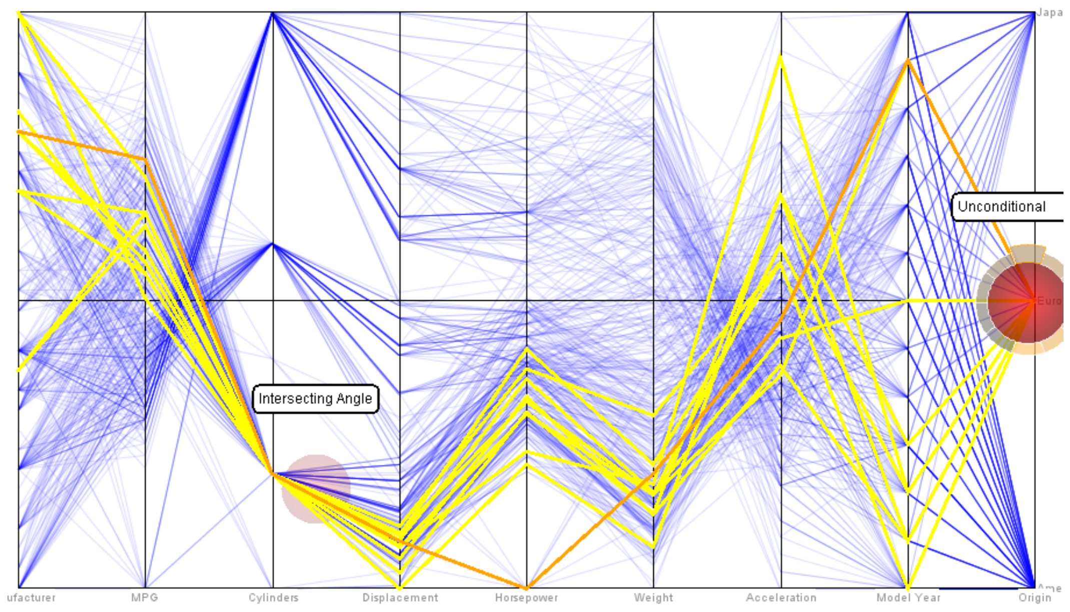


Figure 6.8 — Using multiple lenses to select different correlations in a parallel coordinates diagram.

Figure 6.7 depicts the population movement between states as a graph, derived from the US Census 2000 migration data set[2]. Analysts can quickly find interesting data by using geometric edge properties, for example the magnitude of migrations in a certain direction. For example, grouping edges based on the number of intersections with the lens separates crossing edges from edges that start or end in the focus area. As a result, edges not inside the lens are filtered out and analysts can easily interact only with those that start inside the lens, in this case inspecting migrations from and to 'Utah'. Subsequently, migration edges to the eastern region can easily be select and accumulated by using a child lens that operates only on the focused edges of the parent lens.

Parallel Coordinates

Similar to the work of Guo et al. [2010], *EdgeAnalyzer* can also be applied to a parallel coordinates plot [Inselberg and Dimsdale, 1990]. Figure 6.8 shows the *cars* data set in a parallel coordinates view where multiple lenses are used to highlight and explore parts of the data. While this is not actually a node-link diagram, the grouping mechanisms of *EdgeAnalyzer* can be applied to the line strips and individual lines of the parallel coordinates plot. The left lens focuses data items with few cylinders and a negative slope to the displacement value using an intersection angle grouping mechanism. Its child lens on the right further reduces the set of focused edges to cars originating from Europe.

[2]http://www.census.gov/population/www/cen2000/migration/

6.2 Linked Metric Views

Node-link diagrams often depict the relationships of larger and richer data sets, i.e., only a subset of the data is used as the vertices, edges, and weights of a graph. In some use cases, the data that is left out is important to explore the data set as a whole and to find insights that cannot be found through the node-link diagram alone. Some approaches embed that information in the foreground or background of the node-link diagram, for example *PatViz* [Koch et al., 2009] in its tree-map view (see Section 6.1.6). Depending on the degree of clutter of the node-link diagram, interaction is required to see the data, for example using a lens metaphor to distort links in the region of the lens (e.g., [Wong et al., 2003]) or provide supplementary visual elements (see Section 6.1).

NetLens [Kang et al., 2006], a tool for visual query refinement, allows users to explore relational data by showing different metrics as bar charts. Through selection and filtering, it is possible to drill down into many aspects of a data set. Though designed for networks and relational data, *NetLens* does not actually show these connections. In *MoireGraphs* [Jankun-Kelly and Ma, 2003], nodes are arranged in a radial layout and a focus+context metaphor is employed to show interesting nodes enlarged with their neighborhood. Similarly, *VINCENT* [Kerren et al., 2012] employs a radial layout for network data with attached bar charts of network centralities. These centralities are node metrics based on graph-theoretic properties of a graph, like the degree or closeness. Histograms of these centralities are shown in separate views, detached from the graph visualization.

Coordinated views [Roberts, 2007] are often used when one visualization is not sufficient for all available data. For graphs, frameworks like *Cytoscape* [Shannon et al., 2003] or *Gephi* [Bastian et al., 2009] provide these additional perspectives on the data set. However, these are disconnected from the layout of the graph, i.e., they cannot be easily correlated to the nodes and links.

Graph Metric Views [Panagiotidis et al., 2014a] is an approach based on coordinated views to tackle these challenges. Additional metrics of a data set is shown side-by-side with the relational structure as node-link diagram. The metric views are stackable and can be used in combination with any node-link diagram layout. Besides data-inherent information, the metrics can also be layout-related or graph-theoretic. These metrics can be linked to the node-link diagram through aggregation, selection, and filtering in the metric views.

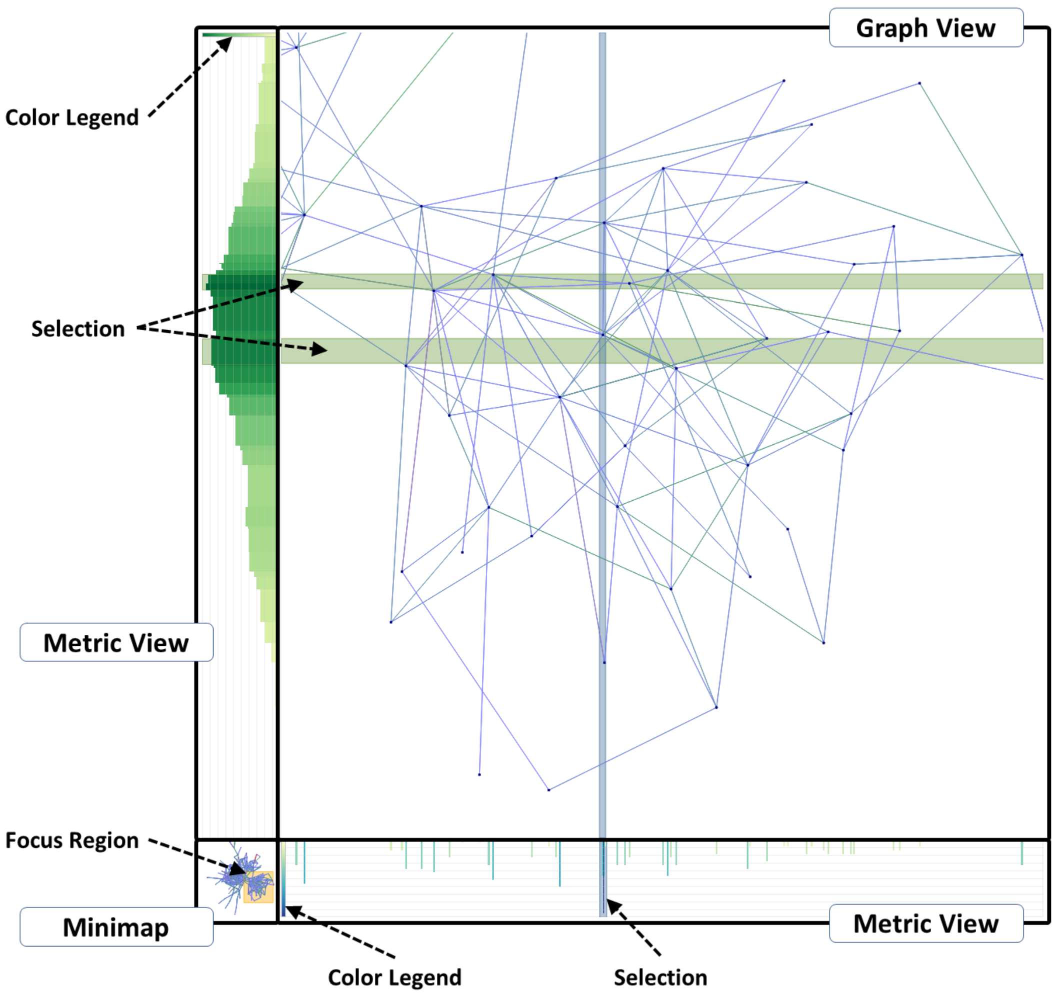


Figure 6.9 — The node-link diagram shows a relation within the data set while coordinated views attached to the sides show (aggregated) metrics of other aspects in the data as histogram-like charts.

6.2.1 Graph Metric Views

Figure 6.9 depicts an overview of *Graph Metric Views*. The *graph view* shows the node-link diagram which is used by users to navigate the graph and find interesting regions, i.e., the users' focus region. During the exploration, the focus region is highlighted in the *minimap*, which always shows the full node-link diagram, i.e., the context. The *metric views* at the sides of the graph view show aggregated metrics of additional data that cannot be integrated into the node-link diagram easily. These metric views depict a one-dimensional projection of the display space of the node-link diagram to one of its axes. They can be arranged and combined in arbitrary order (see Figure 6.10) to allow for easier comparison of the metrics.

Graph Metric Views are designed around the Visual Information Seeking Mantra [Shneiderman, 1996]: overview first, zoom and filter, then details-on-demand. On the one hand, users can explore the node-link diagram in search for interesting regions and then attach metric views for filtering the graph based on patterns that emerge in the histograms. On the other hand, users can also use the metric views to spot outliers and interesting regions in the data and then link those regions to the graph for further investigation.

Metrics are accumulated into bins with regard to the display space of a graph layout, similar to histograms. For this accumulation, nodes and links are projected onto a one-dimensional axis, i.e., metric views to the left and right relate to the ordinate of the node-link diagram and the views at the top and bottom relate to the abscissa, respectively.

The computation of smooth histograms is a classical problem of statistical graphics. Typically, kernel density estimation techniques are employed [Silverman, 1986]. For histograms of node characteristics, there is the traditional problem of building histograms from point samples. For edge-oriented metrics, the histograms are built for intervals that correspond to the projection of the edge to the histogram axis. In either case, a box-filter kernel and pixel-sized bins are used for the representation of histograms. The width of the box filter can be chosen by the user; the default value is given by the width of the node's visual shape in image space.

6.2.2 Metric Types

While some metrics can be determined separately from the graph structure, others are only meaningful with regard to nodes (e.g., vertex degree) or links (e.g., length, edge weight), respectively. Regardless of the metric itself, they can be accumulated and aggregated, for example using only minimum, maximum, or average values instead of the sum, where applicable. For example, the minimum and maximum length of links can be used as an indicator for the aesthetics of layout methods (e.g., lengths should be uniform [Bennett et al., 2007]).

Layout-Related Metrics

Metrics that are based on the visual representation of a graph, i.e., the rendering of the nodes and links as graphical shapes, provide straightforward information, like the number of nodes and links or the sum of the edge weights along the abscissa or ordinate. They can give insight into dense or cluttered regions where distinct nodes cannot be perceived due to overdraw or the number of links cannot be estimated due to edge bundling or because the resolution of the graph

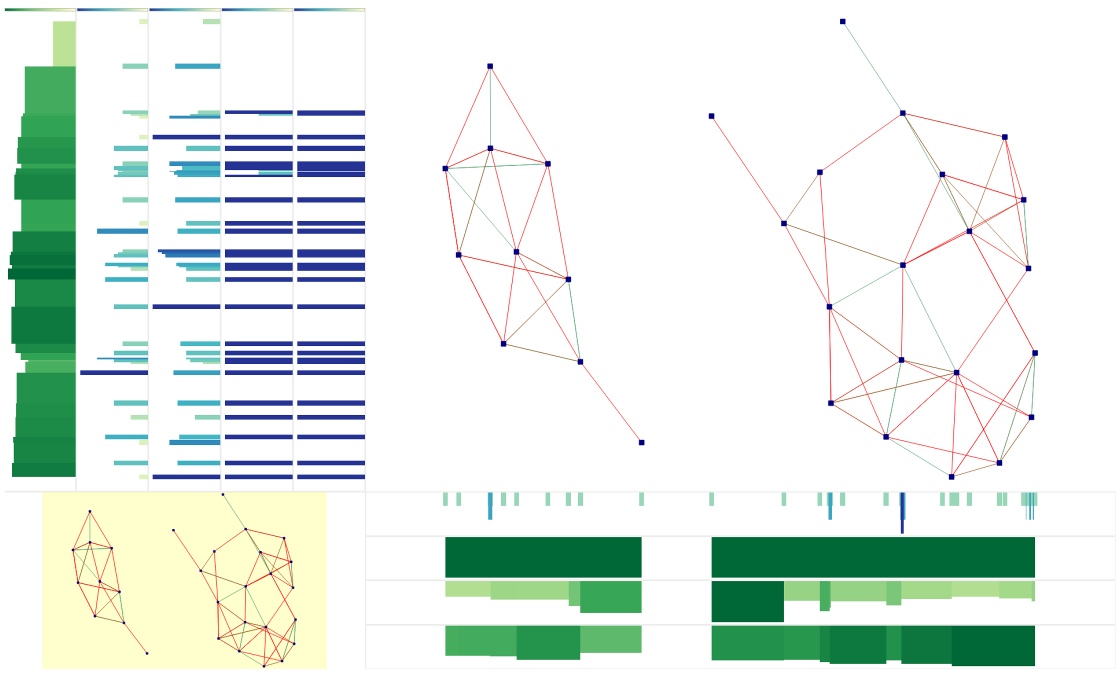


Figure 6.10 — Multiple graph metric views can be used in arbitrary order.

view is not high enough. These metrics can be further refined, for example considering only incoming or outgoing links per node, link intersections, or angles between links.

Graph-Theoretic Metrics

The connectivity of a graph is often the central aspect of interest in network analysis. Derived metrics, like community structures or centralities, are also important. In some use cases, existence of paths and their lengths need to be examined.

Data-Inherent Metrics

Arbitrary values in the data set from which the graph is derived, i.e., those that are neither layout-related nor graph-theoretic, should also be correlated to the layout. For example, call graphs used in software engineering contain many attributes of interest besides the *has-called* relation: file sizes, lines of code, hierarchy levels, various types of complexity, etc. The development and test environment is also a source of valuable information (e.g., profiling counters, code coverage), as well as the version control system (e.g., number of revisions, committer history).

6.2.3 Interaction

The different views (see Figure 6.9) are linked with each other through brushing and linking. The graph can be navigated in the graph view by means of scrolling, zooming, picking, and filtering. These interactions automatically update all metric views so that they always correspond to the visible area of the graph. To preserve the users' mental map, the minimap always shows the whole graph (see bottom left of Figure 6.9) and allows for quicker navigation of the node-link diagram by the same means as the graph view. Additional information, like node labels or edge weights, are available during all interactions through overlays and tooltips.

Filtering in the metric view enables investigation of interesting regions of the graph that are occluded in the node-link diagram. Multiple regions of the histograms can be marked in each metric view separately. Marking regions in the metric views also highlights the corresponding parts in the graph view so that users can see which parts of the graph contribute to the metric value. This is important to link the elements of the node-link diagram mentally to peaks or other visual features in the metric views. These highlighted areas can then be used to filter the node-link diagram, so only those regions remain visible that are currently investigated. From there, additional elements of the node-link diagram, like the hidden neighborhood of the visible nodes, can be gradually unhidden. The current state of all views (i.e., their arrangement, employed metrics, filter settings, etc.) and the graph (i.e., which parts are hidden, selected, etc.) can be stored persistently and shared for collaboration or to continue an analysis at a later point in time.

6.2.4 Evaluation

Important metrics for software systems are properties like coupling and dependencies, which are useful indicators for judging the maintainability of the software. These metrics can be modeled as directed call graphs: vertices represent full-qualified method names, while edges represent the *has-called* relation. Specifically, an edge $e_{i,j} : v_i \rightarrow v_j$ indicates that the method v_i called the method v_j and its weight $w(e_{i,j})$ corresponds to the frequency of the calls.

Figure 6.11 shows such a call graph of Cobertura[3], a code coverage tool for Java. This call graph contains 5000 vertices and 11,319 edges. Laying this graph out according to an energy-based model (e.g., Fruchterman-Reingold [Fruchterman and Reingold, 1991]) results in a cluttered view. Those cluttered and dense areas indicate high coupling (since many methods call each other),

[3]http://cobertura.github.io/cobertura/

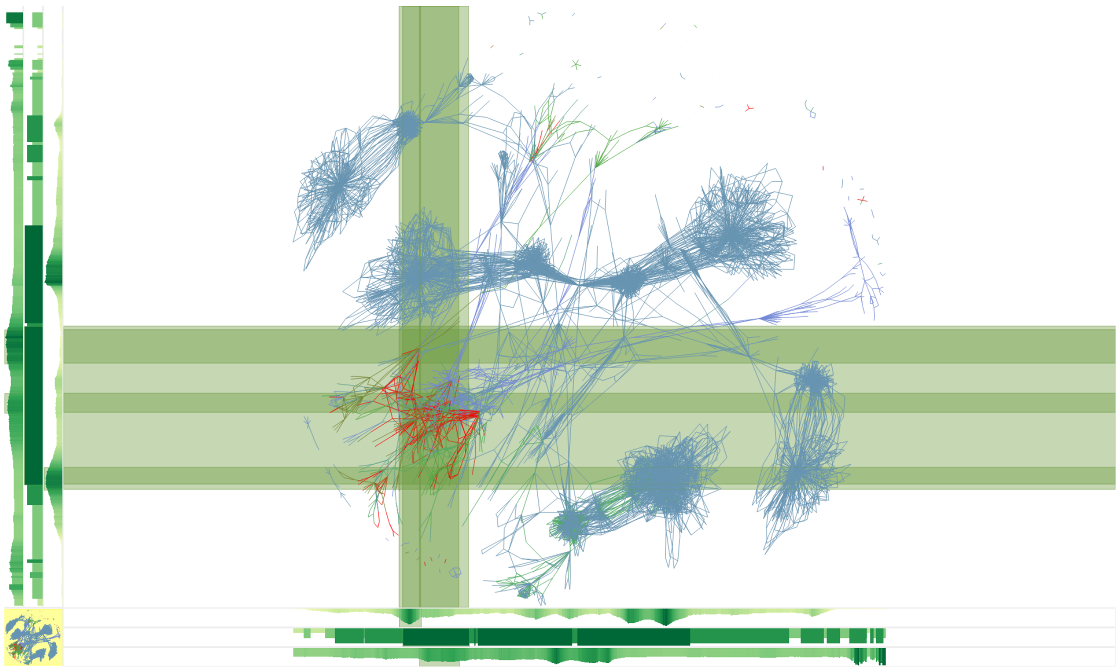


Figure 6.11 — A software call graph depicting the frequency of method calls. The edge weight was mapped to a topological color table, thus red links indicate the most frequent calls. The metric views show (from outermost to innermost metric) the average and maximum package distance, as well as the number of edges. Through selection in the metric views, the interesting region in the node-link diagram is visually linked to the software metrics.

which is unfavorable for designing software systems. From this initial layout it is unclear where these belong to, i.e., to the software itself (i.e., Cobertura) or its dependencies (i.e., the Java SDK or third party libraries). Though this information is important, it is challenging to embed this into the node-link diagram, due to the prevailing overdraw and clutter.

The edge weights are mapped using a topographic color coding, i.e., low edge weights are shown in blue, medium weights in green, and high weights in red. This reveals the calls with the highest frequencies which should be investigated first (see Figure 6.11). To estimate the degree of coupling of these calls, a data-specific metric is attached that shows the distance in the package hierarchy between two nodes as property of the links. Both the average and maximum package distance is used to reveal outliers that might be concealed in the average (see the outer metrics in Figure 6.11). An additional metric view of the number of links is used to determine whether the clearly visible peaks in the metric views stem from the high density of links in that regions. Highlighting those regions allows to verify that these peaks correspond to areas with few links and

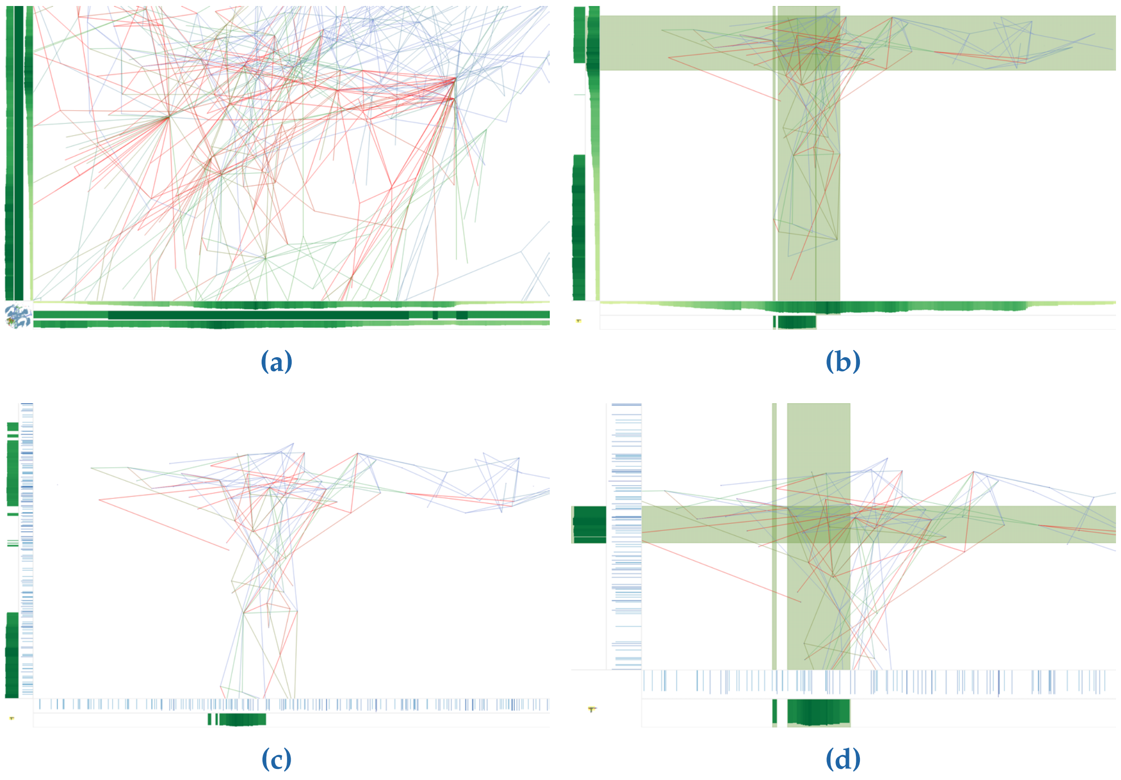


Figure 6.12 — Exploration of the region with the most frequent method calls. (a) The visible part of the node-link diagram mostly contains calls with the highest package distance. (b) Filtering on the metric views allows us to find the regions in the node-link diagram that contain the highest average package distances. Parts of the node-link diagram that were not selected have been hidden. (c) A metric has been attached to show the package depth of the methods. (d) The data set has been reduced to few methods which can be investigated manually and looked up in the source code.

thus few method calls. At this point the hypothesis is formed, that these are very specific methods that are used from many places (due to the high package distance) and called frequently.

To further investigate this, it is necessary to navigate the node-link diagram so that only those parts of the graph are inside the graph view that are responsible for the peaks in the metric views. In doing so, new peaks are revealed since irrelevant parts are now longer shown in the metric views. Repeating this process eventually yields a view of the graph where the second metric, the maximum package distance, is constant for the visible part of the node-link diagram (see Figure 6.12a). Accordingly, this metric view can be removed, leaving more space to the other views. The remaining metric values are very similar in this region and as such hard to distinguish. This can be improved by filtering on the metric views to only show regions where the average package distance is above a threshold and by hiding nodes and links that are not highlighted (see Figure 6.12b).

The graph view now shows a region with high average package distance and few links with respect to the rest of the node-link diagram. Replacing the link count metric with a node metric that represents the maximum hierarchy level of methods (see Figure 6.12c) reveals an outlier, i.e., a singular method with the highest hierarchy level in this part of the node-link diagram. The method can be quickly identified as *CopyFiles.copy* in the package *net.sourceforge.cobertura.reporting.html.files*. New peaks can be found by gradually increasing the thresholds on the metric views. Highlighting these peaks and subsequent filtering leaves only few methods which can be inspected manually through their tooltips (see Figure 6.12d). At this point the source code can be consulted to understand these methods and why they are highly coupled.

6.3 Discussion

The approaches presented in this chapter address challenges when dealing with large amounts of relational data. While they alleviate several issues in this context, they also introduce new ones.

6.3.1 Flexibility and Performance

EdgeAnalyzer can be used in conjunction with various lens shapes and visualizations using line-based information encoding, such as common node-link diagram types or parallel coordinate plots. New shapes, grouping methods, and visual representations for edge groups can be added independently from each other due to its modular approach. The generic approach to identifying edges through intersection of line segments with the lens performs well enough for the tested data sets (e.g., 8000 edges), but bigger data sets might lead to a less fluid user experience. This can be alleviated using many strategies: specialized intersection tests for the specific lens and edge shape, optimization data structures (e.g., quad-trees), and parallelization.

Graph Metric Views can be used with any graph layout and for all types of metrics that can be accumulated meaningfully. Such new metrics can be integrated without changes to other parts. *igraph* [Csárdi and Nepusz, 2006] is currently used for all graph-theoretic operations and to compute the layout. A dedicated thread is used to improve the responsiveness during long running tasks (e.g., finding cliques, computing a new layout). In the meantime, users can continue to explore the graph or interact with the metric views while the status of tasks is indicated by a progress bar. The computation of some metrics is expensive, for example calculating link intersections (quadratic complexity) or finding shortest paths for each node. Offloading these to a thread is inappropriate since the corresponding metric view would be empty or showing outdated information in the meanwhile. While users could disable the responsible metrics until they want to explore them, it is not obvious beforehand which to disable, because it might depend on the graph, the data set, the metric itself, and the total number of active metrics. A reasonable approach could be to monitor the run time for each metric calculation and disable those above a threshold when many updates or interactions are queued, similar to the delayed edge grouping of *EdgeAnalyzer*.

6.3.2 Visual Scalability

Visual representations of edge groups in *EdgeAnalyzer* provide an interaction element as well as an alternative view on cluttered regions. However, the edge grouping in *EdgeAnalyzer* might create so many groups that interaction with them is unfeasible or they clutter the focus region even more.

Reducing the number of edge groups could be accomplished with a k-bin meta grouping: edges are sorted w.r.t. their total order and then n/k edges are put in k bins. This can be combined with other grouping methods and would always create k bins, as long as a total order is defined for the grouping criterion. Alternatively, grouping methods could create hierarchies instead of flat edge groups. For example, if users want to group by time, then it would be reasonable to automatically group according to years first and then months.

Cluttering of the focus region is an inherent problem of having to deal with too many edges or edge groups in general. The arc wheel (see Section 6.1.5) is a first take on this problem, showing edge groups and the grouping hierarchy together (see Figure 6.1). Though the focus region is left unmodified, too many edge groups lead to small and thus indistinguishable arcs. Reducing the number of edge groups is one approach for this problem, but more sophisticated techniques like hyperbolic scaling need to be considered for this problem.

Conceptually, an unlimited number of *Graph Metric Views* can be shown at the sides of the graph. On the one hand, all metric views on a given side become harder to read when adding further metrics without changing the overall size for metric views on that side. On the other hand, resizing those metric views to improve the readability reduces the available screen space for the graph. This is a fundamental trade-off between available screen space and the information to show and needs to be investigated in user studies for different scenarios.

The projection of attributes of a data set onto one dimension of the node-link diagram in *Graph Metric Views* introduces ambiguities. For example, two distinct regions in the same vertical area cannot be effectively discerned in a vertical metric view. This could be alleviated by inspecting their contribution to the metric separately, for example by moving one region out of the graph view, but for that the graph has to be moved and horizontal metrics will change. Alternatively, the same metric could be added horizontally, but this fails if other nodes or links are inside the horizontal area of the ambiguous vertical regions. This situation can thus only be resolved by exploring the regions locally within the node-link diagram, for example using *EdgeAnalyzer*.

6.3.3 Supporting Exploratory Analysis

Using multiple child lenses of *EdgeAnalyzer* facilitates hierarchical filtering and grouping of links. Furthermore, combination of top and child lenses enables operating on links in a Boolean-manner, where top lenses can be thought of as 'OR' while child lenses are similar to 'AND'. This drill-down into data sets can often be accomplished through other means, for example by filtering and selecting in tables, or by picking nodes and links directly. Nevertheless, for dense views or large amounts of data, these become impractical and direct manipulation and interactive exploration is the only feasible approach.

While many metrics are generic and can be used for many data sets, the most interesting aspects are the data-specific ones. As such, exploring multi-dimensional data sets using *Graph Metric Views* requires addition of metrics specific to these data sets. A drill-down is then accomplished by attaching metrics and looking for visual signatures and prevalent structures in the metric views (e.g., outliers or intervals with high values). Through selection and filtering only relevant elements of the graph remain. This top-down interaction to match structures of the node-link diagram to the metric views as well as vice versa can be repeated iteratively.

Both *EdgeAnalyzer* and *Graph Metric Views* are enhancements to existing link-based visualizations and provide means to explore interesting areas iteratively. For both approaches, it is a priori unclear which grouping method or metrics to use. Knowledge of the data and the domain is therefore required. Provenance methods can support understanding how certain information has been found and how to reproduce the necessary steps.

6.4 Summary

This chapter discussed iterative methods for local graph exploration in node-link diagrams. *EdgeAnalyzer* employs a lens metaphor to select multiple edges and group them locally to provide alternative views on dense regions. *Graph Metric Views* show various metrics related to the network data that cannot be integrated into the node-link diagram due to visual clutter and exhaustion of visual channels. Both of these methods aim to help analysts of network data to drill-down into the data sets iteratively and to link insights from and to the relations in the corresponding graphs. They complement existing graph drawing techniques to reveal patterns that would otherwise be hidden due to overdraw and clutter or are not visible at all. In addition to the relationships in the data, they derive additional information based on the geometry, meta data, layout, or graph-theoretic properties.

CHAPTER 7

Tuning Parallel Volume Visualization

Volume visualization systems are often highly customizable by means of numerous parameters, like transfer functions and sampling rates. Choosing the right parameters for an aesthetic and correct image while maintaining a fluid user experience contrasts with growing system sizes and increasing complexity of hardware, software, and data, as well as the demand for higher image resolutions. Previous works in the field of performance analysis and visualization establish methods and frameworks but often concentrate on non-interactive applications (e.g., simulations) or infrastructure (e.g., networks) and focus on the collection and analysis of the performance data itself [Isaacs et al., 2014]. The context, i.e., the physical domain of simulations or the visualization, is either unavailable, static, or pre-recorded. As such, it is challenging to judge how user interaction and the combination of parameters affect the resulting image and user experience.

This chapter presents an approach to analyze and tune a parallel, interactive volume visualization on the VVand. By showing generic and specific metrics together in real-time with the volume during normal usage and interaction, users can mentally link the perceived and measured performance to their interactions to understand the behavior of the system as well as the application. This chapter is partially based on a previous publication [Panagiotidis et al., 2015a].

7.1 Performance Visualization

Analyzing and tuning distributed, interactive visualization systems is challenging since existing approaches to performance visualization and analysis often lack the application context [Isaacs et al., 2014]. Specialized methods and tools are only available as separate applications that focus on presenting and working with performance metrics. Furthermore, they focus on non-interactive use cases, like simulations or infrastructure, in order to detect bottlenecks or anomalies. They collect low-level hardware counters during run time that are analyzed in a post-processing step after the application that should be analyzed has terminated. These approaches are impractical for interactive, distributed visualization systems, where users need to know the context, i.e., the images and the corresponding interactions, to understand the system behavior.

In their performance visualization survey, Isaacs et al. [2014] conclude that there is an increasing need for highly scalable visualizations and improved integration of multiple views of performance data. Heath and Etheridge [1991] employ user-defined annotations in the performance data to correlate it to application code. Wylie and Geimer [2011] show traced performance metrics in the simulation domain separately from the visualization of the simulation data. Schulz et al. [2011] project performance metrics of a hydrodynamics code onto the respective visualization. These approaches require either manual annotation, are not integrated well into the visualization, or reduce the information channels by changing the visualization to show performance metrics.

7.2 Volume Ray Casting

Volume visualization is a technique to project a 3D scalar data space onto a 2D view plane. It is a well-studied problem with a rich body of methods and nowadays well supported on *graphics processing units* (GPUs) [Hadwiger et al., 2006; Beyer et al., 2014]. A prominent method is ray casting, in which the volume is sampled along rays that cover the view plane. This can be efficiently implemented on GPUs since rays can be integrated independently and are coherent, i.e., neighboring rays operate on neighboring data. Furthermore, the volume can be represented as 3D texture that can be sampled efficiently with trilinear interpolation by modern GPUs.

Ray casting can be a time consuming process for large data sets and image resolutions. Several optimizations exist to improve the performance and quality. Early ray termination stops the integration for a ray when the accumulated opacity exceeds a certain threshold [Levoy, 1990]. Empty space skipping uses larger sampling distances along rays if only fully transparent regions are traversed [Cohen and Sheffer, 1994; Klein et al., 2005]. The render time can also be controlled explicitly while balancing spatial and temporal errors [Frey et al., 2014].

7.3 Metric Collection and Presentation

The exploratory approach of this chapter presents relevant metrics to users together with the visualization. To that end, the metrics need to be queried and visualized in addition to the volume rendering. When interacting with the volume, users can directly see the quantified impact of their actions on the performance. Through interaction with the metrics, users can explore and find suitable parameter settings to optimize the performance of the volume rendering. This feedback loop is a quicker way to understand the behavior of the system and the volume visualization than traditional post-mortem approaches.

7.3.1 Relevant Metrics

Generic metrics provide information and hints about the hardware. These include utilizations of *processing units* (PUs) and their memory, transfer rates of the network, or frames per second. Some of these can be polled directly from *application programming interfaces* (APIs) such as NVIDIA's management library (NVML) or NVAPI, or from the operating system, for example through the Windows Management Instrumentation (WMI). Others need to be derived from transferred data or rendered frames during a measured interval.

To gain a deeper understanding, application-specific metrics are necessary. For direct volume visualization, the number of samples or casted rays as well as the percentage of rays affected by early ray termination are interesting. The effect of thread divergence for GPU implementations of the ray casting can be consulted to better understand the impact of acceleration techniques (see Section 7.2).

During exploration and interaction, users change the parameters of the visualization. Showing these along with the metrics allows linking them to performance phenomena.

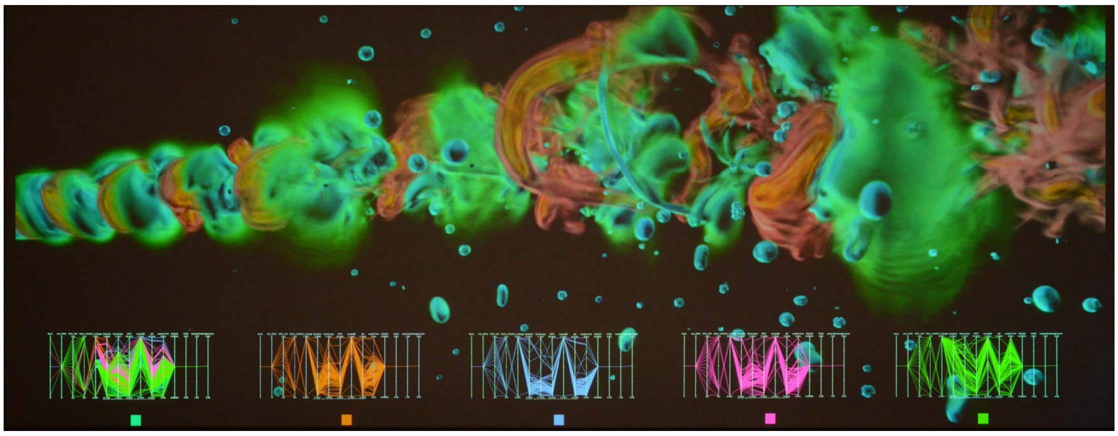


Figure 7.1 — The metrics can be shown as parallel coordinates plot on each display node individually and together on one node.

To quantify the trade-off between correctness and performance, a quality assessment is necessary. For this, users need to specify a ground truth. Then, every frame, the image resulting from parameter changes is compared to that reference. While metrics like *multiscale structural similarity* (MSSIM) [Wang et al., 2003] and PSNR offer a good indication about differences of images, they can only serve as hint, as they were not created for scientific or non-photo realistic visualization scenarios.

7.3.2 Metrics Visualization

The relevant metrics (see Section 7.3.1) are collected every frame on each node that participates in the volume visualization, i.e., the display nodes of the VVand. Presenting them is challenging due to the amount of information and because it is unclear beforehand what phenomena and correlations are interesting. Additionally, a compact visual representation is desirable to minimize occlusion of the volume. Techniques like scatter plots, bar charts, or line charts are thus unfavorable.

The collected metrics for each frame and node can be interpreted as a multi-dimensional data point. Therefore, parallel coordinates are a suitable tool, since they can present many dimensions simultaneously. The distinct patterns between dimension axes also help to identify the relations between metrics and to spot outliers.

After the rendering of each frame of the volume has finished, a parallel coordinates plot is drawn on top of it. There, each poly-line represents a set of metrics, colored according to the originating display node (see Figure 7.1). This enables a quick comparison of the metrics between the nodes.

7.3.3 Gathering and Aggregation

The metrics can be shown on each display node individually or together on one node (see Figures III and 7.1). The number of poly-lines per node can be freely chosen by users, but a lower number (e.g., in the range of 20 to 30) helps to reduce visual clutter while showing recent information. In any case, collected metrics could be stored persistently for offline analysis.

Depending on the goal of the analysis, the individual metrics are less interesting as averages or global minima or maxima. In this case, it is helpful to gather the metrics on one node and aggregate them per dimension. For example, when using a frame lock on large displays (like the VVand), the slowest node determines the overall frame rate.

7.3.4 Interaction

To support real-time analysis, interaction with the parallel coordinates is as important as interaction with the volume. Typical actions include rearranging and scaling the axes and brushing the poly-lines for selection. The parallel coordinates plot can also be moved, resized, or completely hidden for a better view of the volume. The collected metrics can also be cleared, for example before starting an analysis series.

7.4 Evaluation

This approach of overlaid metrics for real-time analysis is evaluated in the context of a parallel volume visualization of the Jet data set ($720 \times 320 \times 320$ voxels) that shows the pressure output from a simulation. The volume is shown on five display nodes of the VVand where a sort-first approach is a natural fit, i.e., the display nodes have access to the whole data set and each display node renders only its part of the whole viewport (see Figure 7.2). For a coherent view, a frame lock is used, i.e., the display nodes synchronize after each frame of the volume rendering. At this point, all relevant metrics have been collected and can then be shown locally or gathered on one display node. Collecting the metrics and rendering the parallel coordinates plot on one node is in the order of 10 ms in this setup.

Rays are sampled using a simple form of empty space skipping, i.e., a multiple of the normal step size (here $n = 6$ times) is used, if the last sample was fully transparent. For adaptive sampling, when a non-empty sample is obtained along a ray, the traversal goes back $n - 1$ steps and the segment is sampled again with the normal step size. Early ray termination aborts further traversal of a ray, when its accumulated opacity reached a configurable saturation threshold. Blinn-Phong shading with central differences for gradient estimation [Hadwiger et al., 2009] is used for illumination.

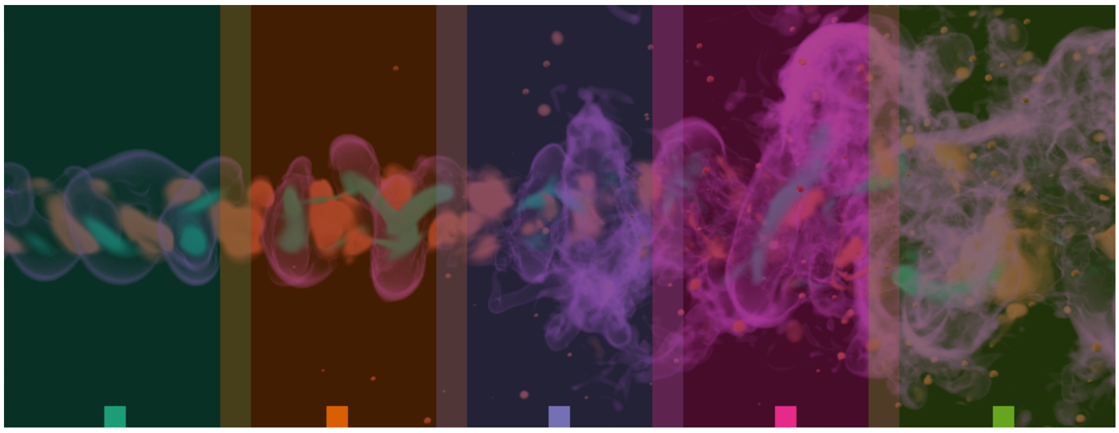


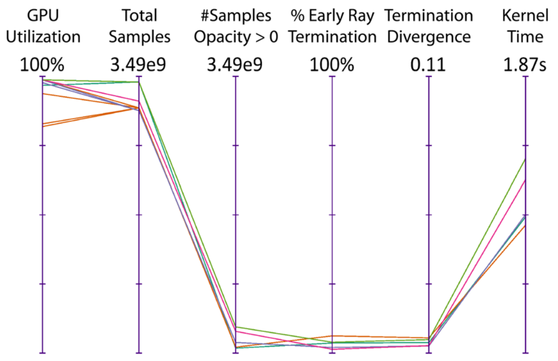
Figure 7.2 — Each of the five display nodes the VVand covers a certain physical area that overlaps with its neighbors for smooth blending between projectors.

7.4.1 Impact of Adaptive Sampling

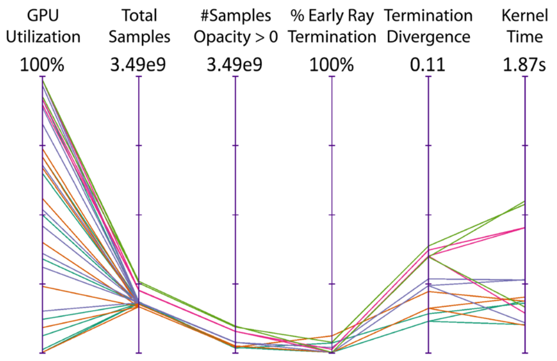
The metrics can be used to investigate the effectiveness of adaptive sampling. For that, a side view is chosen that shows the whole volume (see Figure 7.2).

Without adaptive sampling, the metrics are similar for all render nodes (see Figure 7.3a). In particular, the GPU utilization is high on display nodes that only show little of the volume (i.e., the two left-most display nodes). This is due to the high number of samples taken, even though most of the space is empty for those nodes. Enabling adaptive sampling results in a more non-uniform metric distribution (see Figure 7.3b). Each node is sampling less, as the empty space is skipped, and consequently, the GPU utilization now better matches the expectation from the resulting partial images on the display nodes. This comparison is not an in-depth analysis of the system, but serves more as a verification that the implementation behaves as expected and as quantifiable approach to understand the extent of the induced performance imbalance.

On a global level, even though the overall render time decreases, some display nodes are now idler than before. This is due to them finishing the ray traversal faster and then waiting for the slowest node to release the frame lock. On a local level, the thread divergence increases because threads with terminated rays idle while others in that warp still trace rays. Consequently, the attainable speedup for this optimization technique is limited (see Novák et al. [2010] and Frey et al. [2012] for discussions of and approaches to this issue).



(a) Adaptive sampling disabled.



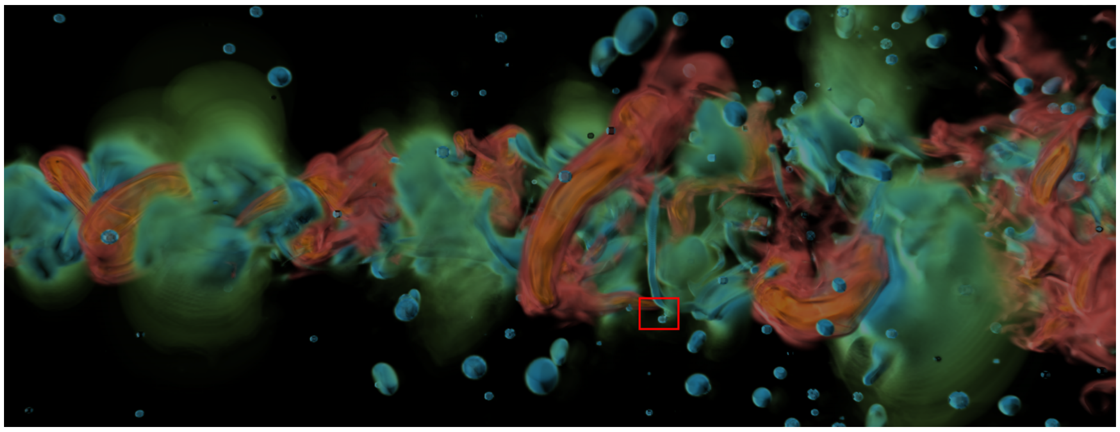
(b) Adaptive sampling enabled.

Figure 7.3 — Metrics for each display node for the view in Figure 7.2 when (a) adaptive sampling is disabled and (b) enabled. The poly-lines are colored according to the color overlays in Figure 7.2.

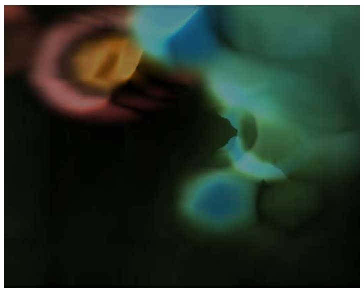
7.4.2 Tuning for Interactivity

For a fluid user experience, the volume visualization needs to maintain a certain frame rate, while at the same time the resulting image quality should be as high as possible. Achieving this is challenging, since it depends on the parameters of the ray caster as well as the view. Furthermore, an optimal parameter setting cannot be determined automatically, since the available quality metrics quantify whether images are closer to a ground truth, but might not be suitable for the VVand or scientific volume visualization [de Freitas Zampolo and Seara, 2005]. As such, suitable parameters need to be sought using an exploratory approach. First, a view of the volume is chosen and a reference image is recorded with the highest possible quality settings. Then, users explore the range of parameter settings and assess the quality visually as well as metric-based in comparison to the reference image. Finally, a set of acceptable parameters is selected by analyzing the parallel coordinates plot of the metrics.

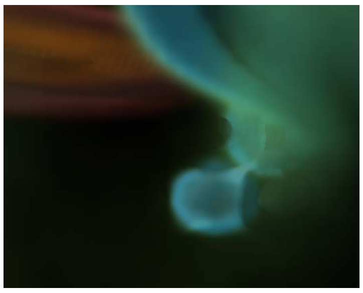
Only two parameters are considered for this analysis: the sampling distance in object and image space. Their impact on the speedup is largely independent, so the combined speedup can be assumed to be roughly multiplicative. While other parameters can be investigated, it may not be sensible. For example, lighting is necessary not only for an increased quality but also to improve perception of depth and structures and thus is always enabled.



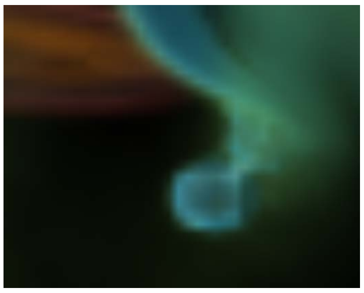
(a) Side view of the volume.



(b) 10 voxels,
1 ray per pixel.



(c) 0.5 voxels,
1 ray per pixel (reference).



(d) 0.5 voxels,
1 ray per 10×10 pixels.

Figure 7.4 — The reference image (a) is used to compute the quality metrics for each parameter set. Even though the values are comparatively close for the tested ranges (see Figure 7.5), the differences are noticeable (compare the reference (c) with the maximum tested sampling distance in object space (b) and in image space (d), respectively).

In the following, the reference image was created using one ray per pixel with a step size of 0.5 voxels along a ray (see Figure 7.4); this took ≈2.82 s. All subsequent frames will be compared to this ground truth using the image metrics MSSIM $\in [0, 1]$ and $PSNR \in [0, \infty)$, where low values indicate high deviation from the reference. For this use case, the quality metrics are computed for the whole image, while the other metrics are aggregated over all display nodes (e.g., only the highest render time is shown).

The goal for this analysis is to render the volume at 20 FPS, i.e., the slowest node may take 50 ms to render. Therefore, each of the two sampling distance parameters is tested individually with the other being fixed, starting from their highest quality setting (0.5 and 1.0, respectively) until the deviation to

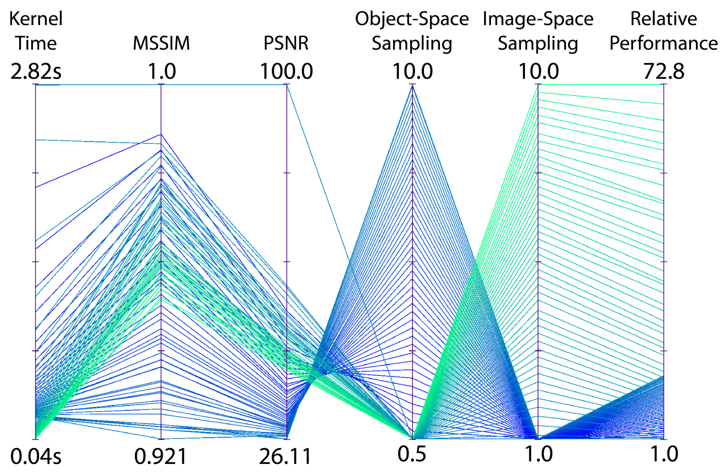


Figure 7.5 — Metrics for different values of the sampling distance in object and image space (see Figure 7.4).

the reference image is too high. The corresponding parallel coordinates plot (see Figure 7.5) clearly shows the trade-off between render time and quality. Next, the quality metric axes in this plot are scanned from top to bottom (high to low quality) in order to find values of the sampling distances that yield similar image quality and a combined speedup that is high enough, i.e., their approximate product (see Relative Performance in Figure 7.5) needs to be at least ≈56. Suitable values for the sampling distances in this case are 1.6 voxels and 1 ray per 4.75×4.75 pixels, resulting in MSSIM=0.9653 and *PSNR*=42.4 and a frame rate of 20 FPS. Note that the image factor can be chosen comparably large as the resolution of the powerwall significantly exceeds the (projected) resolution of this volume data set.

In the view of Figure 7.6, the frame rate drops significantly to ≈12.5 FPS (≈80 ms). Using the same approach as for the previous view yields a sampling distance of 2 voxels in object space and one ray per 5.5×5.5 pixels in image space for a render time of 50 ms. The image quality decreases only slightly from MSSIM=0.9739 and *PSNR*=41.5 (using the optimized values for Figure 7.4) to MSSIM=0.9697 and *PSNR*=38.9. Since the parameters have changed, it is necessary to re-evaluate the previous view; in this case, it is rendered faster with only minor decreases in quality with respect to the previous parameter settings (30 ms, MSSIM=0.9625, *PSNR*=41.3).

7.5 Discussion

Showing the metrics during regular usage with the volume visualization provides some benefits, such as immediate feedback on the state of the system. However, this approach also suffers from some drawbacks w.r.t. scalability and applicability.

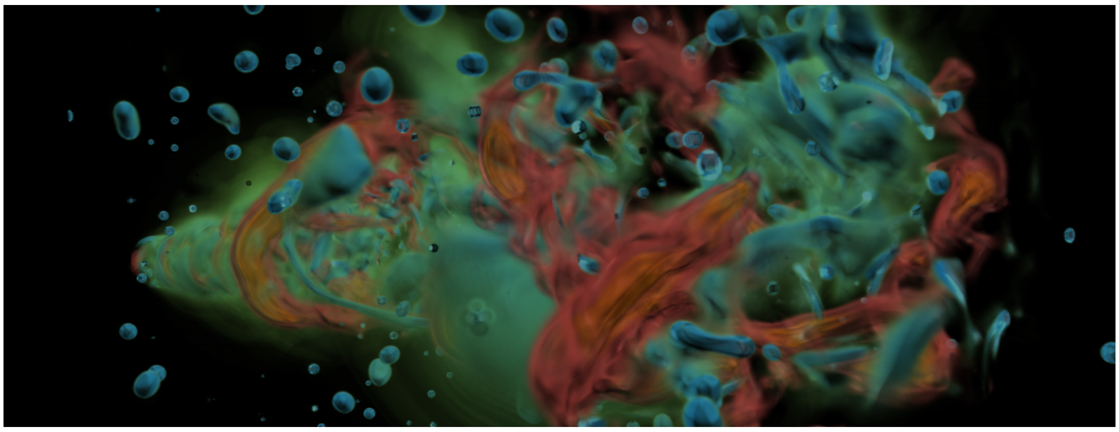


Figure 7.6 — The optimized parameters for Figure 7.4a yield a render time of ≈80 ms or 12.5 FPS for this view. Consequently, the sampling distances need to be optimized for this view using the same procedure (see Section 7.4.2).

7.5.1 Real-Time vs. Post-Mortem Analysis

Collecting and visualizing the metrics in real-time might affect the application too much. In contrast, post-mortem analysis approaches can offload their metrics without interfering with the application, for example using dedicated threads or external monitoring. Since the data is then stored persistently, it can be analyzed using different tools and methods. Therefore, real-time and post-mortem analysis should be employed complementary to increase the understanding of the system. For the use cases presented here (see Section 7.4), gathering the metrics and rendering them as parallel coordinates was in the order of 10 ms.

7.5.2 Accountability of Metrics

Some of the metrics are difficult to interpret, since they are not a straightforward measure of a specific part of the system, but depend on many factors. Consequently, multiple metrics are required to fully understand a single one. For example, a high GPU utilization does not necessarily mean that the GPU is used to full capacity. For one, the driver or the hardware could have reduced the clock frequency to stay within specified *thermal design power* (TDP) specifications to prevent overheating and thus damage to the device. Another source of throttling is energy consumption, which could be deliberately bounded (e.g., to not exceed a certain limit) or due to automatic energy management.

7.5.3 Scalability

While this approach works well for sort-first volume visualization, there are several challenges when using a sort-last approach. In this case, the volume is partitioned in object space and often a much larger number of nodes is used. On the one hand, the collection and gathering of the metrics would not be impacted much, as their data size is rather small (4 B for each metric per frame and node) compared to the image data that needs to be gathered on the display nodes. On the other hand, the resulting parallel coordinates plot would be unreadable due to the increased number of nodes providing metrics. Advanced techniques are then required to improve the readability of the plot, such as density-based approaches, clustering, or bundling, as well as sophisticated interaction, for example using fisheye lenses. Nonetheless, this approach would be helpful for such larger and more complex setups, but requires further effort for the metric visualization.

7.5.4 Applicability

The goal of this approach was to explore how different parameters influence the visualization in order to tune them for specific goals like interactive frame rates. Furthermore, it aims to increase the understanding of the performance characteristics of the volume visualization and the hardware, so users can predict the impact of certain parameter changes better. This use case would be ideal for an automatic parameter optimization, but this is challenging since the quality metrics are not optimal and thus necessitates a human 'in-the-loop' to account for the context (i.e., volume visualization, the VVand).

The collected metrics provide technical information (e.g., GPU utilization) as possible reasons for the user-perceived experience (e.g., image quality, frames per second), so users can link both of these to parameter settings. This enables a quick analysis of a situation and tuning of the parameters for it, but cannot replace an in-depth analysis with profilers and other specialized tools. In particular, a more sophisticated performance analysis (e.g., finding and eliminating bottlenecks) would require the consideration of even more metrics, such as network or disc utilization.

Supporting more metrics is possible by adding them as further dimensions in the parallel coordinates, but querying more metrics might stall the whole system to the point that it becomes unusable. Additionally, more metrics increase the clutter in the parallel coordinates plot in the same way as increasing the number of nodes.

The same optimization techniques could then be used (see Section 7.5.3) but the visualization of the metrics might then require more space on the display to be readable. As a result, more of the volume would be occluded, and even though this is only a minor drawback on displays like the VVand, it limits the applicability of this approach on other display setups.

7.6 Summary

This chapter discussed an exploratory approach to real-time analysis and tuning of a parallel, interactive volume visualization on the VVand. Metrics are collected from the hardware as well as the software (i.e., the GPU and the ray casting kernel) and are visualized as parallel coordinates plot in real-time. This facilitates the joint exploration of the data set as well as the impact of different user-chosen parameter settings on the performance and the image quality compared to a high-quality reference image. This approach was used to fine-tune a volume visualization on the VVand to achieve interactive frame rates while retaining high quality.

Part IV

Conclusion

Chapter 8

Visualization Challenges in Distributed Heterogeneous Computing Environments

The shift to the *general purpose computing on graphics processing units* (GPGPU) paradigm and the adoption of *processing units* (PUs) has influenced science and industry. Even people at home employ heterogeneous computing unknowingly (e.g., in hardware-accelerated video encoding/decoding or rendering of web sites) or explicitly (e.g., mining bitcoins with *graphics processing units* (GPUs)). Such paradigm shifts can occur quickly and have to be expected in the coming years, since vendors currently experiment with new technologies (e.g., stacked DRAM, reduced cache sizes) which might also require rethinking how to develop *high performance computing* (HPC) and traditional applications. In a shorter period, the computer graphics and visualization community will have to deal with the increased complexity to program GPUs with DirectX 12 and Vulkan in order to fully utilize the fine-grained access to their graphics capabilities, similar to how CUDA and OpenCL enabled the same for their compute capabilities. The physical size and the image resolution of displays are also becoming larger in the same way that storage and compute power grow. Consequently, visualization applications have to be increasingly designed and planned for distributed environments, as these large displays become more widespread. This should not be seen as a burden on the community but as chance since it enables possibilities for visualization techniques that were previously unfeasible.

As such, traditional aspects like usability and performance become more of a basic prerequisite while other aspects become equally or more important in the future.

This thesis investigated three aspects that are so far only hardly considered in distributed visualization on large displays: abstraction, resilience, and exploratory analysis. While previous works on these aspects examine them in isolation, their combination becomes increasingly important as systems and data sets grow inevitably in size and complexity. The approaches presented in this thesis fit naturally into the context of heterogeneous systems and visualization on large displays and simplify many details of this field. DIANA (see Chapter 3) simplifies GPGPU, *distributed memory processing* (DMP), and distributed visualization. *Per-pixel linked lists* (PPLLs) (see Chapter 4) simplify compositing and combination of visualization techniques. Strategies for fault-tolerant distributed visualization (see Chapter 5) increase the resilience but also the user experience of distributed visualization systems. For analysis of dense graph data sets, *EdgeAnalyzer* (see Section 6.1) provides an interaction metaphor while *Graph Metric Views* (see Section 6.2) provide additional data. Presentation of system and application metrics in real-time (see Chapter 7) allows for a better understanding and enables tuning of visualization applications on these systems.

8.1 Research Question 1

Which approaches can visualization systems employ to support evaluating and switching between programming models and visualization techniques?

This is an intriguing question, as the visualization pipeline (see Section 2.1.1) defines modular stages and frameworks like *ParaView* or *MegaMol* already allow for a reconfigurable visualization of data sets. Nevertheless, these and similar approaches operate on such a high level that changing the underlying *application programming interface* (API) is not trivial. For example, *VTK* [Schroeder et al., 2006], the framework employed in *ParaView*, abandoned the deprecated configurable OpenGL pipeline only in 2014[1], 10 years after the programmable pipeline was introduced. This is a real problem, as Vulkan is already on the horizon and promises non-negligible speedups (see Chapter 1). As such, intermediary concepts are needed to accelerate the adoption of new techniques and paradigms in existing and future scientific and industrial applications.

[1]http://www.kitware.com/source/home/post/144

This thesis proposes several answers to the question and presents methods to tackle its underlying challenges. First, switching between APIs and PUs is seamlessly possible with DIANA which even incorporates transparent DMP. Second, changing between visualization methods at the mapping and rendering stage can be supported by using *per-pixel linked lists* (PPLLs) for order-independent transparency and distributed rendering. Third, visualization applications and techniques can be evaluated in real-time with the exploratory approach presented in Chapter 7. Fourth, communication and data flow between the involved processes and nodes can be analyzed using local graph exploration as described in Chapter 6. These graph analysis approaches can also be used for the traces of commands and buffers from DIANA. Even though the traces of individual application runs can be manageable and clear, the analysis of multiple traces from runs on different PUs and with different command variants can become cluttered and confusing.

8.2 Research Question 2

How must resilience in visualization systems improve to support challenges of large displays and increasing system sizes?

Visualization on large displays is essentially a specific application of distributed systems. As such, traditional approaches to fault tolerance apply in this context but are antithetical as explained in Chapter 5. The other approaches in this thesis allow to go beyond the presented strategies and their example implementations.

Using PPLLs for distributed visualization simplifies compositing, but also allows for interaction with the images to some extent without repeated rendering, for example by converting them to *Volumetric Depth Images* [Frey et al., 2013]. The approach in Chapter 7 can be used to extract profiles for performance characteristics of the system. Users could define thresholds for various metrics and label these accordingly, for example frame or transfer rates for fault-free cases. This can be shown in the metric presentation itself as backdrop, so users can see whether the system behaves normally. If it does not, the current situation can be labeled accordingly. Repeating this labeling could ultimately empower the system to automatically detect and classify its state.

In his seminal article, Gray [1986] analyzes a fault-tolerant system and concludes that the concepts of fault-tolerant hardware need to be applied to software to raise the mean time between failures of systems by several orders of magnitude. According to Gray, the key to a fault-tolerant system is a hierarchical decomposition into modules, where failures in each module do not propagate into other modules, both for hardware and software. Besides tolerating hardware and software faults, this includes software modularity (i.e., processes and messages) as well as process-pairs combined with transaction mechanisms for data and message integrity.

DIANA addresses all of these points. For one, it strongly enforces modularization via plugins and opaque data types. Furthermore, the use of *Protocol Buffers* (PBs) follows the message-passing paradigm closely and provides data and message integrity. In addition, the command invocation in DIANA can be used to increase the resilience of an application. As mentioned in Section 3.3.3, memoization can be employed to return cached results, even when no PUs or peers are available to process the computation, resulting in an effective, portable, and transparent implementation of checkpoint-restart. While this requires a lot of storage, it could be limited to the last few application runs or a cache pruning strategy could employed (e.g., least-recently-used). When using the remote invocation in DIANA, failing or unresponsive peers could be handled automatically. Finally, using multiple implementations of a command automatically allows for software redundancy, i.e., an operation can be executed easily in different variants, the results compared, and the differences visualized (e.g., using the comparative analysis approach from Section 4.1.2).

8.3 Research Question 3

How can systems for interactive visualization on large displays be analyzed and optimized?

Visualization is used for analysis of applications and data, so the answer to this question should naturally be: "with visualization". Applications that run on systems like the VVand are inherently distributed but approaches to analyze such systems do not cope well with interactivity. The exploratory approach of Chapter 7 is specifically designed with this question in mind. It can be extended to incorporate data from PPLLs as well as DIANA (e.g., example number of fragments, execution durations) for a more complete view on the system.

The data flow in distributed systems can be modeled as dynamic graph. In the context of distributed visualization on the VVand, additional information, such as the metrics from Chapter 7 or tracing information from DIANA, is available. In addition, the user interaction itself is a valuable piece of information that needs to be correlated to the behavior of the system. Both *EdgeAnalyzer* and *Graph Metric Views* are appropriate methods to investigate these dynamic, multidimensional data sets. Similarly to the parallel coordinates plot in Chapter 7, they can be embedded into the visualization application so that users receive instant feedback and can drill-down into the performance data.

The progress glyph Section 5.3.2 is both an indicator for the current state of the system and a high-level depiction of its topology. Embedding such information allows for a better and timelier understanding of the system state since the information does not need to be manually gathered from various places, for example monitoring services or dedicated profiling tools.

8.4 Open Challenges

Chapters 3 to 7 discussed limitations and some open challenges of the individual approaches. In the context of the research questions and the answers provided by this thesis, the following additional areas could be improved and investigated further.

DIANA provides an interesting take on GPGPU and DMP. Its entry barriers need to be lowered by simplifying the integration and usage of commands. Regarding the overhead of PBs (see Section 3.3.4), it would be worthwhile to experiment with the newly introduced arena allocation, PB 3 (the successor to the version used in DIANA), or *Cap'n Proto*[2]. The overhead for the tracing has to be lowered in order to efficiently support developers in creating complex and large systems that will be deployed on huge heterogeneous systems. The command memoization (see Section 3.3.3) has the potential to be valuable not only for scheduling and resilience but also for reproducibility and verification of individual computations and whole simulations, prediction of future system behavior, and as a foundation for quantifiable computing. Ultimately, memoization could be used to modify partial results in long-running applications through replay and just-in-time replacement of commands. Finally, more PUs should be integrated, like Intel's Xeon Phi, *field-programmable gate arrays* (FPGAs), or the upcoming *Neo* architecture[3].

Per-pixel linked lists are an interesting data structure with many applications in visualization. Even though they have been mentioned already over 30 years ago [Carpenter, 1984] and have been implemented on GPUs five years ago [Yang et al., 2010], there are still many issues to be resolved. Their biggest drawback — high memory requirements — needs to be addressed efficiently without compromising the correctness for compositing. At that point PPLLs will become more useful and widespread and might even replace the prevailing z-buffering approach in rendering. Until then they need to be compacted or

[2] An alternative for Protocol Buffers from their original author, see https://capnproto.org/.

[3] http://rexcomputing.com/

reduced in size, for example with general compression methods. A specialized compression scheme could also be a worthwhile investigation, for example similar to the approach of O'Neil and Burtscher [2011] to compress floating point numbers.

Resilience in distributed visualization will become more important if not critical on the road to exascale and uncertainty visualization might become an integral part of many visualization systems [Moreland, 2012]. Choosing the right trade-offs in this 'tug of war' will be difficult and will require thorough discussion, since nobody wants to sacrifice performance and instead maximize the utilization of every resource. The presented strategies in Chapter 5 are only a starting point and it may become necessary to look into implementations for each of them. The image-space recovery methods discussed in Section 5.3.4 are promising and specialized techniques could yield results that are more satisfying. The progress glyph (see Section 5.3.2) is also an interesting addition to a visualization that promotes integration of otherwise independent systems, such as infrastructure monitoring and application metrics.

Local graph exploration is a useful approach to gain insight in dense relational and connected data. The approaches in this thesis (see Chapter 6) can be employed to various domains but need to be evaluated in the respective applications with experts. As with all techniques that need to scale, graph visualization will profit from higher display resolutions and distributed visualization. For that, optimized techniques that utilize GPUs more extensively [Panagiotidis et al., 2015b] are required in addition to novel visual metaphors.

Exploratory analysis and tuning is useful for interactive visualization systems, as developers can iterate faster in contrast to post-mortem analysis. The applicability of the presented approach to other domains besides volume rendering as well as further metrics should be evaluated. The metrics presentation also requires further work, as it surely becomes too cluttered when used for bigger systems.

8.5 Summary

This thesis presented and discussed approaches of abstraction, resilience, and exploratory analysis for distributed visualization systems in heterogeneous computing environments. These aspects became increasingly important in the last decade due to the rise of GPGPU and evolving programming paradigms. Some of the presented approaches seem counterproductive at first — querying for operations on PUs via SQL; using excessive amounts of memory to collect all fragments of a distributed scene; partitioning into smaller and non-neighboring tiles; increasing rendering overhead; rendering low quality images first — but they all offer benefits that can outweigh their disadvantages. With additional engineering effort, they can be improved to the point where they form a robust and stable foundation for future research and application in the advent of emerging technology.

Appendix A
System Specifications

A.1 enka

Operating System	Windows 8.1 Enterprise, 64-bit
CPU	Intel i7-3820 (3.6 GHz), 4 cores, 8 threads
Main Memory	32 GB
GPU	NVIDIA GeForce GTX 770, 4.0 GB VRAM
GPU Driver	ForceWare 350.12, CUDA 6.5

A.2 VVand Display Node

Operating System	Windows HPC Server 2008 R2, Service Pack 1, 64-bit
CPU	2× Intel Xeon X5650 (2.66 GHz), 6 cores, 12 threads
Main Memory	24 GB
GPU	2× NVIDIA Quadro 6000, 6.0 GB VRAM
GPU Driver	ForceWare 340.52, CUDA 6.5

A.3 VVand Cluster Node

Operating System	Windows HPC Server 2008 R2, Service Pack 1, 64-bit
CPU	2× Intel Xeon E5620 (2.4 GHz), 4 cores, 8 threads
Main Memory	24 GB
GPU	2× NVIDIA GeForce GTX 480, 1.5 GB VRAM
GPU Driver	344.48, CUDA 6.5

Appendix B

DIANA

B.1 Build Information

DIANA was built using Microsoft Visual Studio 2013 (12.0.31101.00 Update 4), Google Protocol Buffers 2.5.0, NVIDIA CUDA 6.5, and Intel Thread Building Blocks 4.2.

B.2 Usage Example

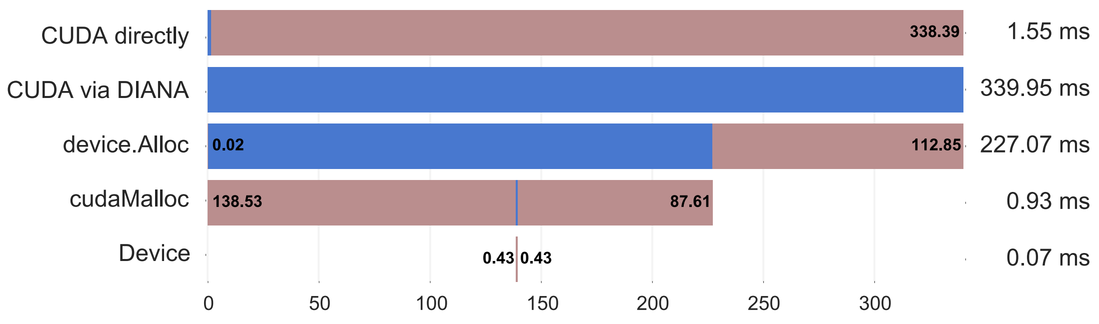
Listing B.1 shows a typical example (without error checking for brevity) for creating a buffer, querying for devices, and invoking a memory transfer in DIANA.

Listing B.1 — DIANA usage example (C++ 11).

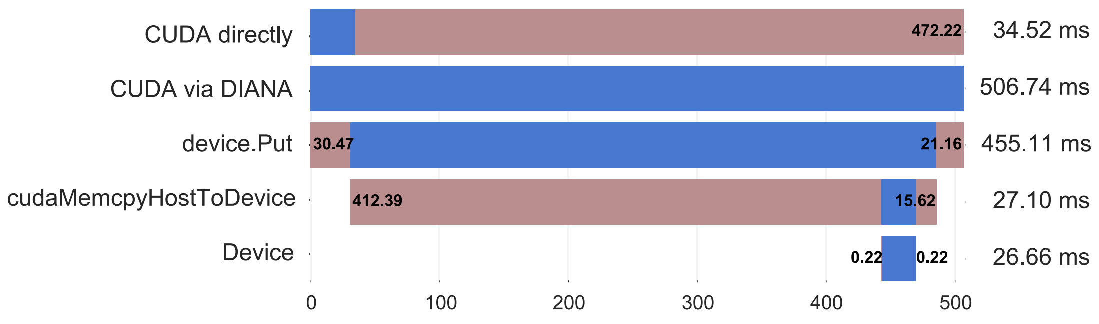
```
std::vector<float> inputData;
// fill inputData, for example by reading from a file

// Create a buffer, large enough to hold all bytes of inputData
auto bufferSize = inputData.size() * sizeof(inputData.front());
auto buffer = diana::createBuffer(bufferSize);

// Query the first device
diana::DeviceID deviceID{ 0 };
{
    auto query = ↩
      std::unique_ptr<diana::DatabaseQuery>(diana::query("SELECT * FROM ↩
      devices ORDER BY OID ASC LIMIT 1"));
    query->exec();
    deviceID = query->asUInteger("OID");
}
```

```

// Blocking host-to-device transfer
{
    auto data = diana::makeProto<diana::commands::device::Put>(buffer, ↩
      bufferSize);
    // Copy data to Protocol Buffer
    data->set_source(inputData.data(), bufferSize);

    auto query = ↩
      std::unique_ptr<diana::DatabaseQuery>(diana::query("SELECT * FROM ↩
      commands c WHERE c.deviceID=:deviceID AND c.name=:name"));
    query->bind(":name", data->GetTypeName());
    query->bind(":deviceID", deviceID);
    query->exec();

    auto commandID = query->asUInteger("OID");
    auto barrier = diana::createBarrier();

    auto commandStatusID = diana::invoke(deviceID, commandID, data, ↩
      diana::barrierNotify(barrier));

    diana::wait(barrier);
}
```

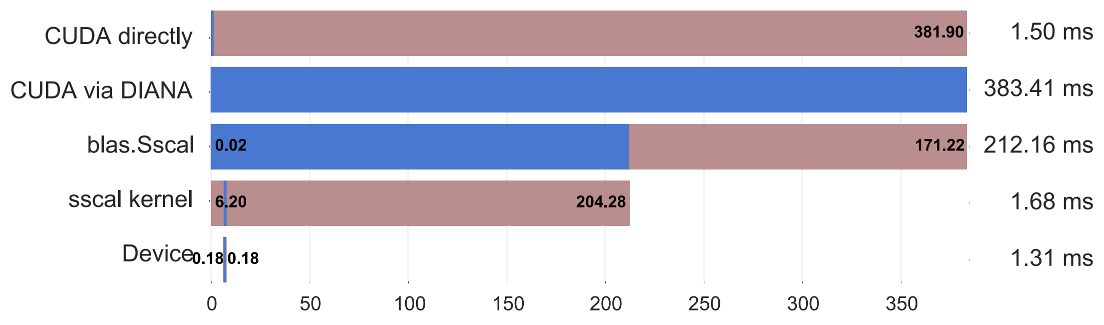
B.3 Overhead: Memory Operations & Kernel Execution



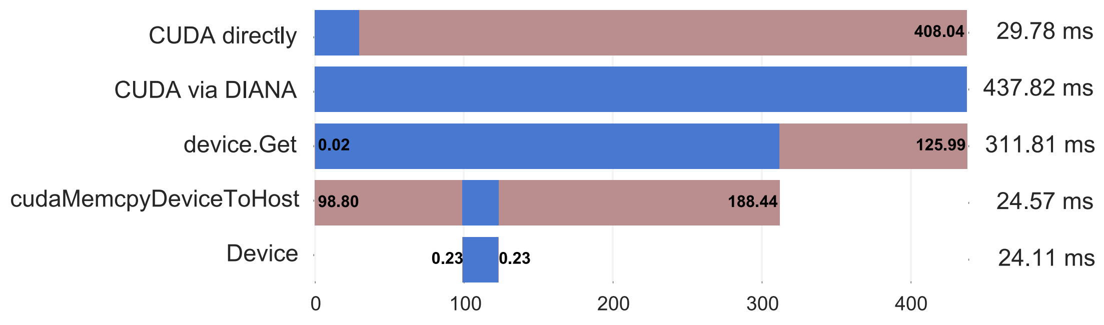
a — device.Alloc (CUDA)



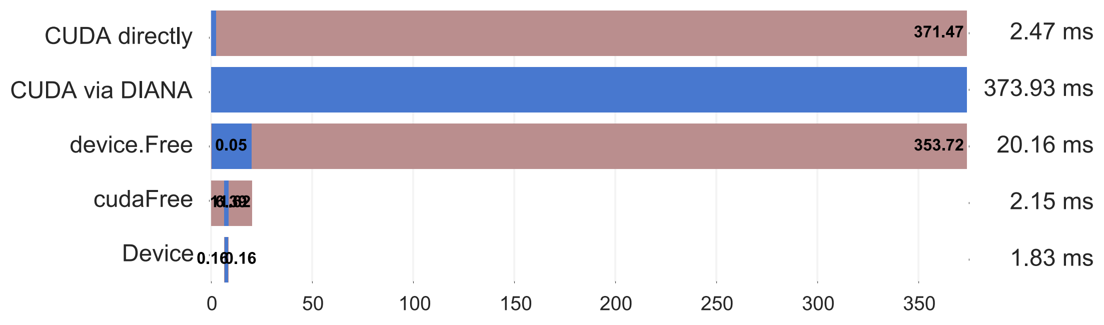
b — device.Put (CUDA)



c — blas.Sscal (CUDA)

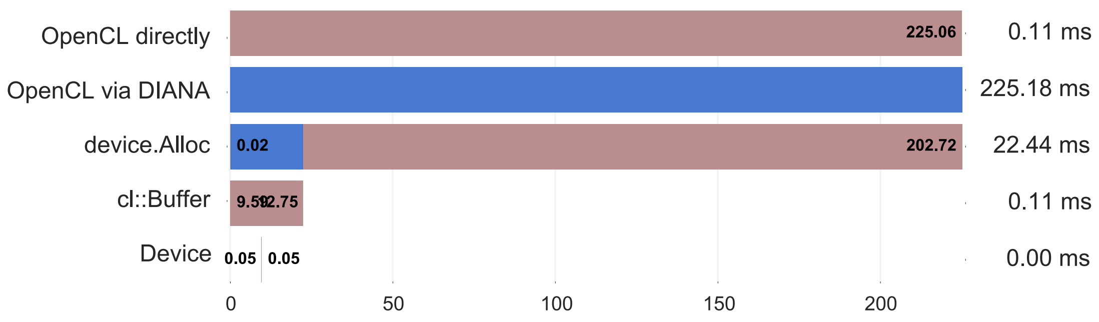


d — device.Get (CUDA)

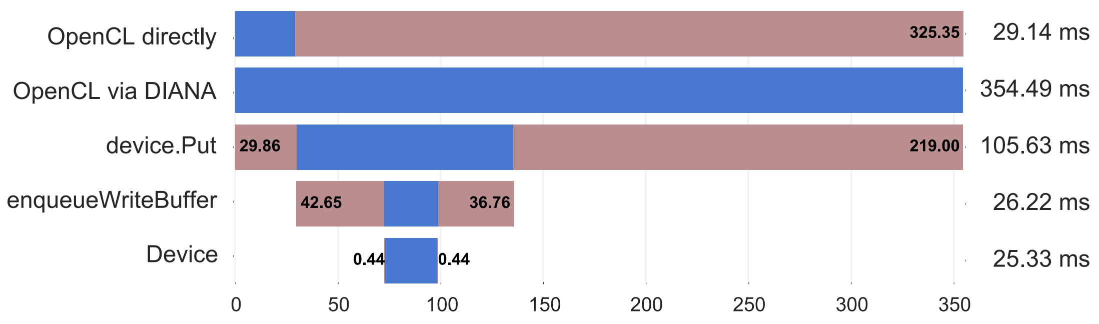


e — device.Free (CUDA)

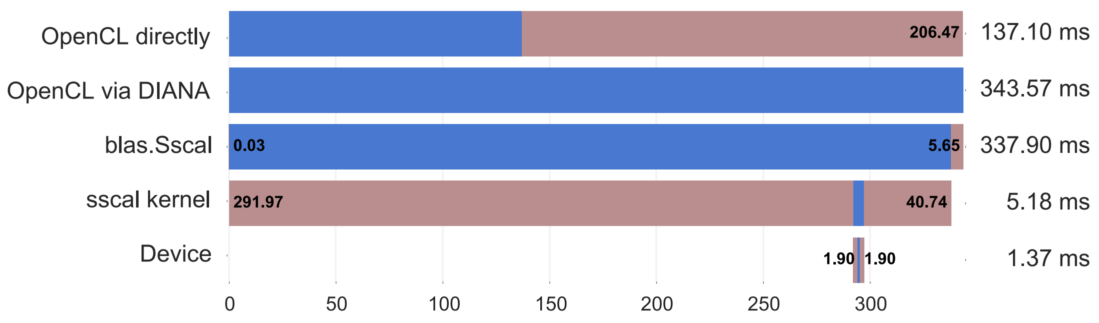
Figure B.1 — Worst case run times for the overhead benchmark (see Section 3.2.2) when calling CUDA via DIANA with command and buffer tracing enabled.



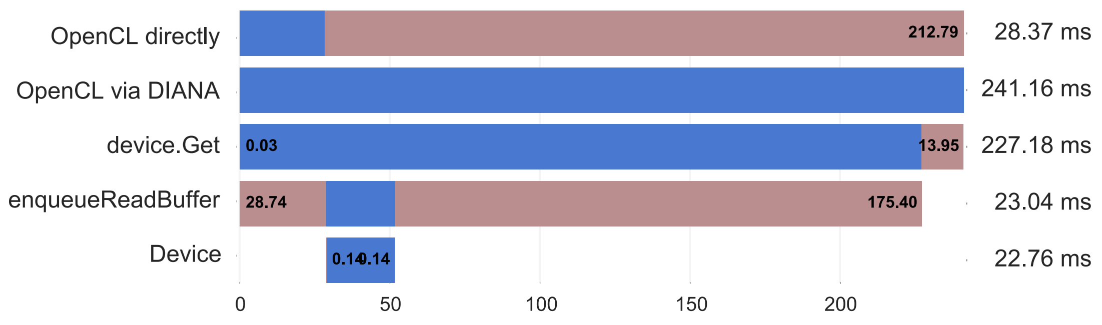
a — device.Alloc (OpenCL)



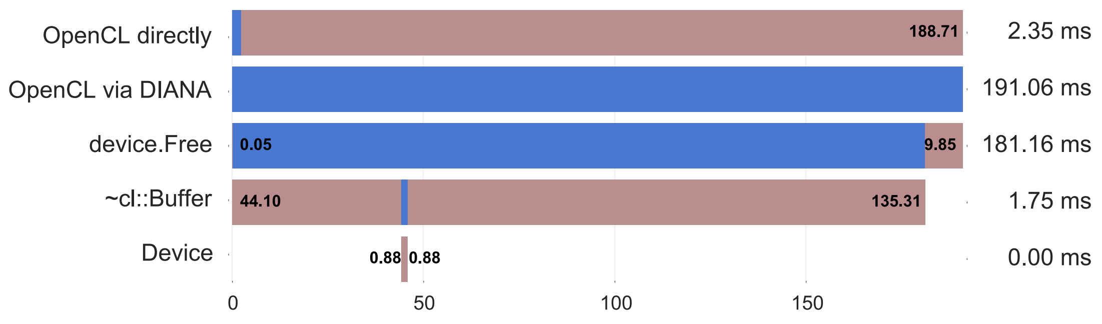
b — device.Put (OpenCL)



c — blas.Sscal (OpenCL)



d — device.Get (OpenCL)



e — device.Free (OpenCL)

Figure B.2 — Worst case run times for the overhead benchmark (see Section 3.2.2) when calling OpenCL via DIANA with command and buffer tracing enabled.

B.4 diana.commands.blas.Dgemm

The DGEMM interface is modeled after Netlib's reference implementation in *BLAS*[1]. Subwindows of the matrices are defined as extension (Lines 31 to 39 in Listing B.2) because they are not supported by all implementations and since buffers are not pointers in DIANA, i.e., the pointer arithmetic is performed in the command before calling a kernel or by the called library.

Listing B.2 — Protocol Buffer definition of DGEMM in DIANA.

```
 1 package diana.commands.blas;
 2 import "diana/diana.proto";
 3 option optimize_for = SPEED;
 4
 5 enum Trans
 6 {
 7   N = 0;
 8   T = 1;
 9   C = 2;
10 }
11
12 message Dgemm
13 {
14   optional Trans transa = 1 [default = N];
15   optional Trans transb = 2 [default = N];
16   optional uint64 m      = 3 [default = 0];
17   optional uint64 n      = 4 [default = 0];
18   optional uint64 k      = 5 [default = 0];
19   optional double alpha = 6 [default = 1.0];
20   optional double beta   = 7 [default = 0.0];
21   optional uint64 A      = 8
22     [default = 0, (diana_id) = BUFFER, (diana_access) = READ];
23   optional uint64 B      = 9
24     [default = 0, (diana_id) = BUFFER, (diana_access) = READ];
25   optional uint64 C      = 10
26     [default = 0, (diana_id) = BUFFER, (diana_access) = READ_WRITE];
27
28   extensions 100 to 999;
29 }
30
31 extend Dgemm
32 {
33   optional uint64 offsetA = 100 [default = 0];
34   optional uint64 lda     = 101 [default = 0];
35   optional uint64 offsetB = 102 [default = 0];
36   optional uint64 ldb     = 103 [default = 0];
37   optional uint64 offsetC = 104 [default = 0];
38   optional uint64 ldc     = 105 [default = 0];
39 }
```

[1]http://www.netlib.org/lapack/explore-html/d7/d2b/dgemm_8f.html

Bibliography

J. Ahrens, B. Geveci, and C. Law. ParaView: An End-User Tool for Large-Data Visualization. In *Visualization Handbook*, pages 717–731. Butterworth-Heinemann, 2005. 16, 134

J. Andersson, D. Baker, P.-L. Griffais, J. McDonald, T. Olson, A. Pranckevicius, and N. Smedberg. glNext: The Future of High Performance Graphics, 2015. [Online]. Available: `http://www.gdcvault.com/play/1022018/`. 4

D. Archambault, J. Abello, J. Kennedy, S. Kobourov, K.-L. Ma, S. Miksch, C. Muelder, and A. C. Telea. Temporal Multivariate Networks. In A. Kerren, H. C. Purchase, and M. O. Ward, editors, *Multivariate Network Visualization*, number 8380 in Lecture Notes in Computer Science, pages 151–174. Springer International Publishing, 2014. 93

J. Badger, J. Sauder, J. Adams, S. Antonysamy, K. Bain, M. Bergseid, S. Buchanan, M. Buchanan, Y. Batiyenko, J. Christopher, S. Emtage, A. Eroshkina, I. Feil, E. Furlong, K. Gajiwala, X. Gao, D. He, J. Hendle, A. Huber, K. Hoda, P. Kearins, C. Kissinger, B. Laubert, H. Lewis, J. Lin, K. Loomis, D. Lorimer, G. Louie, M. Maletic, C. Marsh, I. Miller, J. Molinari, H. Muller-Dieckmann, J. Newman, B. Noland, B. Pagarigan, F. Park, T. Peat, K. Post, S. Radojicic, A. Ramos, R. Romero, M. Rutter, W. Sanderson, K. Schwinn, J. Tresser, J. Winhoven, T. Wright, L. Wu, J. Xu, and T. Harris. Structural analysis of a set of proteins resulting from a bacterial genomics project. *Proteins: Structure, Function, and Bioinformatics*, 60(4):787–796, 2005. 55

A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *IEEE International Conference on Cluster Computing Workshops and Posters*, pages 1–7, 2010. 20

D. Bartz and C. Silva. Rendering and Visualization in Parallel Environments. 2001. [Online]. Available: `http://www.gris.uni-tuebingen.de/people/staff/bartz/tutorials/eg2001tutorial/`. 13

M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the Third International AAAI Conference on Weblogs and Social Media*, pages 361–362, 2009. 107

G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998. 93

L. Bavoil and E. Enderton. Constant-Memory Order-Independent Transparency Techniques, 2011. [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/ConstantMemoryOIT.pdf. 50

L. Bavoil and K. Myers. Order Independent Transparency with Dual Depth Peeling, 2008. [Online]. Available: http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf. 50

L. Bavoil, S. P. Callahan, A. Lefohn, J. L. D. Comba, and C. T. Silva. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 97–104. ACM, 2007. 50, 64

C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The Aesthetics of Graph Visualization. In D. W. Cunningham, G. Meyer, and L. Neumann, editors, *Proceedings of the Third Workshop on Computational Aesthetics*, pages 57–64. The Eurographics Association, 2007. 109

H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000. 49

M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester. Image inpainting. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 417–424. ACM, 2000. 77, 79

E. W. Bethel, H. Childs, and C. Hansen, editors. *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, 2012. 13

J. Beyer, M. Hadwiger, and H. Pfister. A Survey of GPU-Based Large-Scale Volume Visualization. In R. Borgo, R. Maciejewski, and I. Viola, editors, *Eurographics/IEEE Conference on Visualization - State of the Art Reports*. The Eurographics Association, 2014. 120

E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 73–80. ACM, 1993. 96

A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984. 18

Bitcoin Wiki. Mining hardware comparison, 2015a. [Online]. Available: https://en.bitcoin.it/w/index.php?title=Mining_hardware_comparison&oldid=56965. 17

Bitcoin Wiki. Non-specialized hardware comparison, 2015b. [Online]. Available: https://en.bitcoin.it/w/index.php?title=Non-specialized_hardware_comparison&oldid=56942. 17

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002. 36, 148

D. Blythe. Rise of the Graphics Processor. *Proceedings of the IEEE*, 96(5):761–778, 2008. 19

B. Bode, M. Butler, T. Dunning, T. Hoefler, W. Kramer, W. Gropp, and W.-m. Hwu. The Blue Water Super-System for Super-Science. In *Contemporary High Performance Computing*, pages 339–366. Chapman and Hall/CRC, 2013. 1

J. Breitbart. CuPP - A framework for easy CUDA integration. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009. 20

K. Brodlie, R. Allendes Osorio, and A. Lopes. A Review of Uncertainty in Data Visualization. In J. Dill, R. Earnshaw, D. Kasik, J. Vince, and P. C. Wong, editors, *Expanding the Frontiers of Visual Analytics and Visualization*, pages 81–109. Springer London, 2012. 77, 83

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in Heterogeneous Computing. *Scientific Programming*, 18(1): 1–33, 2010. 17

A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013. 19

M. Burch, C. Vehlow, F. Beck, S. Diehl, and D. Weiskopf. Parallel Edge Splatting for Scalable Dynamic Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2344–2353, 2011. 94

M. Burch, C. Vehlow, N. Konevtsova, and D. Weiskopf. Evaluating Partially Drawn Links for Directed Graph Edges. In M. v. Kreveld and B. Speckmann, editors, *Graph Drawing*, number 7034 in Lecture Notes in Computer Science, pages 226–237. Springer Berlin Heidelberg, 2012. 94

M. Burch, H. Schmauder, A. Panagiotidis, and D. Weiskopf. Partial Link Drawings for Nodes, Links, and Regions of Interest. In *Proceedings of the 18th International Conference on Information Visualisation (IV)*, pages 53–58. IEEE, 2014. 6

F. Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009. 3, 72, 73

L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, pages 103–108. ACM, 1984. 15, 50, 137

B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 17

D. Cohen and Z. Sheffer. Proximity clouds — an acceleration technique for 3d grid traversal. *The Visual Computer*, 11(1):27–38, 1994. 121

G. Csárdi and T. Nepusz. The igraph software package for complex network research. *InterJournal Complex Systems*, 1695, 2006. 114

W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li. Geometry-Based Edge Clustering for Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, 2008. 94

R. de Freitas Zampolo and R. Seara. A comparison of image quality metric performances under practical conditions. In *IEEE International Conference on Image Processing*, volume 3, pages III–1192–5, 2005. 125

M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng. Confluent Drawings: Visualizing Non-planar Diagrams in a Planar Way. In G. Liotta, editor, *Graph Drawing*, number 2912 in Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, 2004. 94

J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. In V. Kindratenko, editor, *Numerical Computations with GPUs*, pages 3–28. Springer International Publishing, 2014. 20

J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15 (9):803–820, 2003. 1, 36

J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *International Conference on High Performance Computing and Simulation*, pages 224–231. IEEE, 2010. 20

I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013. 72, 75, 76

S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, 2009. 16

G. Ellis and A. Dix. A Taxonomy of Clutter Reduction for Information Visualisation. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1216–1223, 2007. 94

C. Everitt. Interactive Order-Independent Transparency, 2001. [Online]. Available: https://developer.nvidia.com/system/files/akamai/gamedev/docs/order_independent_transparency.pdf. 50

C. Everitt, G. Sellers, J. McDonald, and T. Foley. Approaching Zero Driver Overhead in OpenGL, 2014. [Online]. Available: http://www.slideshare.net/CassEveritt/approaching-zero-driver-overhead. 4

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 1990. 14, 50

S. Frey and T. Ertl. PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 131–140, 2010. 20

S. Frey, G. Reina, and T. Ertl. SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 399–406, 2012. 124

S. Frey, F. Sadlo, and T. Ertl. Explorable Volumetric Depth Images from Raycasting. In *Conference on Graphics, Patterns and Images*, pages 123–130, 2013. 82, 135

S. Frey, F. Sadlo, K.-L. Ma, and T. Ertl. Interactive Progressive Visualization with Space-Time Error Control. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2397–2406, 2014. 121

T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. 93, 111

G. W. Furnas. Generalized fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 16–23. ACM, 1986. 96

E. R. Gansner, Y. Koren, and S. North. Graph Drawing by Stress Majorization. In J. Pach, editor, *Graph Drawing*, number 3383 in Lecture Notes in Computer Science, pages 239–250. Springer Berlin Heidelberg, 2005. 93

J. Gao, H. Liu, J. Huang, M. Beck, Q. Wu, T. Moore, and J. A. Kohl. Time-Critical Distributed Visualization with Fault Tolerance. In K.-L. Ma and J. M. Favre, editors, *Proceedings of the 8th Eurographics Symposium on Parallel Graphics and Visualization*, pages 65–72. The Eurographics Association, 2008. 74

D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*. PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, 2010. 19

D. Göddeke, R. Strzodka, and S. Turek. Accelerating Double Precision FEM Simulations with GPUs. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *18th Symposium Simulationstechnique*, Frontiers in Simulation, pages 139–144, 2005. 20

I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 161–172. ACM, 2013. 20

J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE Computer Society, 1986. 135

S. Grottel, M. Krone, C. Müller, G. Reina, and T. Ertl. MegaMol - A Prototyping Framework for Particle-Based Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):201–214, 2015. 16, 42, 57, 134

P. Guo, H. Xiao, Z. Wang, and X. Yuan. Interactive local clustering operations for high dimensional data in parallel coordinates. In *IEEE Pacific Visualization Symposium*, pages 97–104, 2010. 106

C. Gutwin. Improving focus targeting in interactive fisheye views. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, pages 267–274. ACM, 2002. 99

R. B. Haber and D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*, pages 74–93. IEEE Computer Society Press, 1990. 9

M. Hadwiger, J. M. Kniss, C. Rezk-Salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006. 120

M. Hadwiger, P. Ljung, C. R. Salama, and T. Ropinski. Advanced Illumination Techniques for GPU-based Volume Raycasting. In *ACM SIGGRAPH 2009 Courses*, pages 2:1–2:166. ACM, 2009. 123

G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D: Nonlinear Phenomena*, 149(4):248–277, 2001. 41, 85

I. Haque and V. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 691–696, 2010. 69

M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991. 120

N. Henry, J.-D. Fekete, and M. J. McGuffin. NodeTrix: a Hybrid Visualization of Social Networks. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1302 –1309, 2007. 94

D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12 (5):741–748, 2006. 94, 95, 105

D. Holten and J. J. van Wijk. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum*, 28(3):983–990, 2009. 94

D. Holten and J. J. v. Wijk. A user study on visualizing directed edges in graphs. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 2299–2308. ACM, 2009. 94

W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996. 49

G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 129–140. ACM, 2001. 16

G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002. 16

C. Hurter, B. Tissoires, and S. Conversy. FromDaDy: Spreading Aircraft Trajectories Across Views to Support Iterative Queries. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1017–1024, 2009. 95

C. Hurter, O. Ersoy, and A. Telea. Graph Bundling by Kernel Density Estimation. *Computer Graphics Forum*, 31:865–874, 2012. 94

A. Inselberg and B. Dimsdale. Parallel coordinates: a tool for visualizing multi-dimensional geometry. In *Proceedings of the 1st conference on Visualization '90*, pages 361–378. IEEE Computer Society Press, 1990. 106

K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the Art of Performance Visualization. In R. Borgo, R. Maciejewski, and I. Viola, editors, *Eurographics Conference on Visualization - State of the Art Reports*. The Eurographics Association, 2014. 119, 120

T. J. Jankun-Kelly and K.-L. Ma. MoireGraphs: radial focus+context visualization and interaction for graphs with visual nodes. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 59–66, 2003. 107

T. Jansen. *GPU++ - An Embedded GPU Development System for General-Purpose Computations*. PhD thesis, Universitätsbibliothek der TU München, 2007. 20

C. Johnson and A. Sanderson. A Next Step: Visualizing Errors and Uncertainty. *IEEE Computer Graphics and Applications*, 23(5):6–10, 2003. 77, 83

H. Kang, C. Plaisant, B. Lee, and B. Bederson. NetLens: Iterative Exploration of Content-Actor Network Data. In *Proceedings of the IEEE Symposium on Visual Analytics Science And Technology*, pages 91–98, 2006. 107

M. Kaufmann and D. Wagner, editors. *Drawing Graphs–Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer, 2001. 93

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Evaluation of per-pixel linked lists for distributed rendering and comparative analysis. *Computing and Visualization in Science*, 15(3):111–121, 2013a. 7, 49, 53

D. Kauker, M. Krone, A. Panagiotidis, G. Reina, and T. Ertl. Rendering Molecular Surfaces using Order-Independent Transparency. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 33–40. The Eurographics Association, 2013b. 7, 49, 64

D. M. Kauker. *Distributed Computing and Transparency Rendering for Large Displays*. PhD thesis, University of Stuttgart, 2015. 7

D. A. Keim, F. Mansmann, J. Schneidewind, and H. Ziegler. Challenges in Visual Data Analysis. In *Proceedings of the 10th International Conference on Information Visualization (IV)*, pages 9 –16, 2006. 96

A. Kerren, H. Köstinger, and B. Zimmer. VINCENT–Visualization of Network Centralities. In *Proceedings of the International Conference on Information Visualization Theory and Applications*, pages 703–712, 2012. 107

T. Klein, M. Strengert, S. Stegmaier, and T. Ertl. Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *IEEE Visualization*, pages 223–230, 2005. 121

P. Knowles, G. Leach, and F. Zambetta. Efficient Layered Fragment Buffer Techniques. In P. Cozzi and C. Riccio, editors, *OpenGL Insights*, pages 279–292. CRC Press, 2012. 50, 53

S. Koch, H. Bosch, M. Giereth, and T. Ertl. Iterative integration of visual insights during patent search and analysis. In *IEEE Symposium on Visual Analytics Science and Technology*, pages 203–210, 2009. 8, 103, 104, 107

R. Koduri. On APIs and the future of Mantle, 2015. [Online]. Available: `https://community.amd.com/community/gaming/blog/2015/05/12/on-apis-and-the-future-of-mantle`. 4

K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, pages 149–160. ACM, 2013. 20

M. Krone, J. E. Stone, T. Ertl, and K. Schulten. Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *Eurographics Conference on Visualization - Short Papers*, volume 1, pages 67–71. The Eurographics Association, 2012. 49

J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg. Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In *Proceedings* *of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pages 327–335. ACM, 1990. 19

M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, 1990. 120

F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Efficient depth peeling via bucket sort. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 51–57. ACM, 2009. 50

J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967. 99

Mark J. Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, University of North Carolina, 2003. 19

M. Maule, J. L. D. Comba, R. P. Torchelsen, and R. Bastos. A survey of raster-based transparency techniques. *Computers & Graphics*, 35(6):1023–1034, 2011. 14, 15, 50

A. Meligy. Parallel and Distributed Visualization: The State of the Art. In *Fifth International Conference on Computer Graphics, Imaging and Visualisation*, pages 329–336, 2008. 13

P. Messmer, P. Mullowney, and B. Granger. GPULib: GPU Computing in High-Level Languages. *Computing in Science & Engineering*, 10(5):70–73, 2008. 20

S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994. 13

K. Moreland. Oh, $#*@! Exascale! The Effect of Emerging Architectures on Scientific Discovery. In *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis*, pages 224–231, 2012. 2, 4, 138

K. Moreland, W. Kendall, T. Peterka, and J. Huang. An Image Compositing Solution at Scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 25:1–25:10. ACM, 2011. 16

J. Mulder, F. Groen, and J. van Wijk. Pixel masks for screen-door transparency. In *Visualization '98. Proceedings*, pages 351–358, 1998. 15

J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, 1999. 10

C. Müller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):605–617, 2009. 20

C. Müller, G. Reina, and T. Ertl. The VVand: A Two-Tier System Design for High-Resolution Stereo Rendering. In *Extended Abstracts on Human Factors in Computing Systems*. ACM, 2013. 21

T. Ni, G. Schmidt, O. Staadt, M. Livingston, R. Ball, and R. May. A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. In *Virtual Reality Conference, 2006*, pages 223–236, 2006. 15

Nicole Hemsoth. The Tiny Chip That Could Disrupt Exascale Computing, 2015. [Online]. Available: http://www.theplatform.net/2015/03/12/the-little-chip-that-could-disrupt-exascale-computing/. 2

J. Novák, V. Havran, and C. Dachsbacher. Path Regeneration for Interactive Path Tracing. In *Eurographics Conference on Visualization - Short Papers*, pages 61–64. The Eurographics Association, 2010. 124

M. A. O'Neil and M. Burtscher. Floating-point data compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, pages 7:1–7:7. ACM, 2011. 138

A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin. Diffusion curves: a vector representation for smooth-shaded images. *ACM Transactions on Graphics*, 27(3):92:1–92:8, 2008. 77

R. S. A. Osorio and K. W. Brodlie. Uncertain Flow Visualization using LIC. In W. Tang and J. Collomosse, editors, *Theory and Practice of Computer Graphics*, pages 215–222. The Eurographics Association, 2009. 83

J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. 19

A. Panagiotidis. *Entwicklung einer Interaktionsmetapher für die Exploration und Manipulation von gebündelten Kanten*. Diploma Thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2009, Published: Diploma Thesis: University of Stuttgart, Institute of Visualisation and Interactive Systems, Visualisation and Interactive Systems. 8

A. Panagiotidis, H. Bosch, S. Koch, and T. Ertl. EdgeAnalyzer: Exploratory Analysis through Advanced Edge Interaction. In *44th Hawaii International Conference on System Sciences*, pages 1–10. IEEE Computer Society, 2011a. 8, 94, 96

A. Panagiotidis, D. Kauker, S. Frey, and T. Ertl. DIANA: A Device Abstraction Framework for Parallel Computations. In P. Iványi and B. H. V. Topping, editors, *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Civil-Comp Press, 2011b, paper 20. 7, 29, 30

A. Panagiotidis, D. Kauker, F. Sadlo, and T. Ertl. Distributed Computation and Large-Scale Visualization in Heterogeneous Compute Environments. In *11th International Symposium on Parallel and Distributed Computing*, pages 87–94, 2012. 7, 29, 44

A. Panagiotidis, M. Burch, O. Deussen, D. Weiskopf, and T. Ertl. Graph Exploration by Multiple Linked Metric Views. In *Proceedings of the 18th International Conference on Information Visualisation (IV)*, pages 19–26. IEEE, 2014a. 8, 94, 107

A. Panagiotidis, G. Reina, and T. Ertl. Strategies for Fault-Tolerant Distributed Visualization. In *2014 IEEE Pacific Visualization Symposium*, pages 286–290, 2014b. 7, 71

A. Panagiotidis, S. Frey, and T. Ertl. Exploratory Performance Analysis and Tuning of Parallel Interactive Volume Visualization on Large Displays. In *Eurographics Conference on Visualization - Short Papers*, pages 13–17. The Eurographics Association, 2015a. 8, 119

A. Panagiotidis, G. Reina, M. Burch, T. Pfannkuch, and T. Ertl. Consistently GPU-Accelerated Graph Visualization. In *International Symposium on Visual Information Communication and Interaction*. ACM, 2015b. 6, 138

C. Papadopoulos, K. Petkov, A. Kaufman, and K. Mueller. The Reality Deck–an Immersive Gigapixel Display. *IEEE Computer Graphics and Applications*, 35(1): 33–45, 2015. 15

P. Papadopoulos, J. Kohl, and B. Semeraro. CUMULVS: extending a generic steering and visualization middleware for application fault-tolerance. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, pages 127–136. IEEE, 1998. 74

M. Peercy, M. Segal, and D. Gerstmann. A Performance-oriented Data Parallel Virtual Machine for GPUs. In *ACM SIGGRAPH 2006 Sketches*. ACM, 2006. 19

P. Pérez, M. Gangnet, and A. Blake. Poisson image editing. In *ACM Transactions on Graphics*, pages 313–318. ACM, 2003. 77, 79, 82

T. A. Phelps and R. Wilensky. The Multivalent Browser: A Platform for New Ideas. In *Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 58–67. ACM, 2001. 96

T. Porter and T. Duff. Compositing digital images. In *Proceedings of the 11th annual conference on Computer Graphics and Interactive Techniques*, pages 253–259. ACM, 1984. 12

K. Potter, P. Rosen, and C. R. Johnson. From Quantification to Visualization: A Taxonomy of Uncertainty Visualization Approaches. In A. M. Dienstfrey and R. F. Boisvert, editors, *Uncertainty Quantification in Scientific Computing*, number 377 in IFIP Advances in Information and Communication Technology, pages 226–249. Springer Berlin Heidelberg, 2012. 77

H. C. Purchase, R. F. Cohen, and M. James. Validating Graph Drawing Aesthetics. In *Proceedings of the Symposium on Graph Drawing*, pages 435–446. Springer, 1996. 93

J. Roberts. State of the Art: Coordinated Multiple Views in Exploratory Visualization. In *Proceedings of the Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization*, pages 61–71, 2007. 107

R. Rosenholtz, Y. Li, J. Mansfield, and Z. Jin. Feature congestion: a measure of display clutter. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 761–770. ACM, 2005. 93

M. Rotard, M. Giereth, and T. Ertl. Semantic Lenses: Seamless Augmentation of Web Pages with Context Information from Implicit Queries. *Computers and Graphics*, 31(3):361–369, 2007. 96

F. Sadlo, M. Üffinger, T. Ertl, and D. Weiskopf. On the Finite-Time Scope for Computing Lagrangian Coherent Structures from Lyapunov Exponents. In *Topological Methods in Data Analysis and Visualization II*, Mathematics and Visualization, pages 269–281. Springer Berlin Heidelberg, 2012. 43

R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with K-way replication. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 75–53. IEEE, 2001. 74

N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009. 20

B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *International Conference on Dependable Systems and Networks*, pages 249–258, 2006. 69

B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007. 3, 69

W. Schroeder, K. Martin, and B. Lorensen. *Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., 4 edition, 2006. 134

M. Schulz, J. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci. Interpreting Performance Data across Intuitive Domains. In *2011 International Conference on Parallel Processing*, pages 206–215, 2011. 120

G. Sellers, R. S. Wright, and N. Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 6 edition, 2013. 11

J. Shade, S. Gortler, L.-W. He, and R. Szeliski. Layered Depth Images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 231–242. ACM, 1998. 82

A. Shan. Heterogeneous Processing: a Strategy for Augmenting Moore's Law. *Linux Journal*, (142), 2006. 16

P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research*, 13 (11):2498–2504, 2003. 107

B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, 1996. 109

B. Silverman. *Density Estimation for Statistics and Data Analysis*, volume 26 of *Monographs on Statistics & Applied Probability*. Chapman and Hall, 1986. 109

A. Smith. Gnomes per second in Vulkan and OpenGL ES, 2015. [Online]. Available: `http://blog.imgtec.com/powervr/gnomes-per-second-in-vulkan-and-opengl-es`. 4

J. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *IEEE Symposium on Information Visualization*, pages 57–65, 2000. 102

M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 49–56, 2008. 20

E. Strohmaier. TOP500 Supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, 2006. 1, 2

N. Sundaram, A. Raghunathan, and S. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009. 20

C. Tominski, J. Abello, F. van Ham, and H. Schumann. Fisheye Tree Views and Lenses for Graph Visualization. In *Proceedings of the 10th International Conference on Information Visualization (IV)*, pages 17–24. IEEE, 2006. 95

TOP500 Supercomputer Sites. June 2015 List, 2015. [Online]. Available: `http://top500.org/lists/2015/06/`. 2

J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977. 22

M. Üffinger, F. Sadlo, M. Kirby, C. Hansen, and T. Ertl. FTLE Computation Beyond First-Order Approximation. In *Eurographics 2012 - Short Papers*, pages 61–64. The Eurographics Association, 2012. 42

D. van Dyk, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke, and C. Gutwenger. HONEI: A collection of libraries for numerical computations targeting multiple processor architectures. *Computer Physics Communications*, 180(12):2534–2543, 2009. 20

R. van Liere and W. de Leeuw. GraphSplatting: Visualizing Graphs As Continuous Fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2): 206–212, 2003. 94

A. A. Vasilakis and I. Fudos. Z-fighting aware depth peeling. In *ACM SIGGRAPH 2011 Posters*, pages 77:1–77:1. ACM, 2011. 50

Z. Wang, E. Simoncelli, and A. Bovik. Multiscale structural similarity for image quality assessment. In *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1398–1402 Vol.2, 2003. 84, 122

C. Ware, H. Purchase, L. Colpoys, and M. McGill. Cognitive Measurements of Graph Aesthetics. *Information Visualization*, 1(2):103–110, 2002. 93

D. Weiskopf. *GPU-Based Interactive Visualization Techniques*. Springer Berlin Heidelberg, 2007. 9

H. Wickham and L. Stryjewski. 40 Years of Boxplots, 2012. [Online]. Available: http://vita.had.co.nz/papers/boxplots.html. 22

Wikipedia. List of AMD graphics processing units, 2015a. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_AMD_graphics_processing_units&oldid=672834010. 36

Wikipedia. List of Nvidia graphics processing units, 2015b. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=672620028. 37

Wikipedia. Memoization, 2015c. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Memoization&oldid=673479170. 46

Wikipedia. OpenGL, 2015d. [Online]. Available: https://en.wikipedia.org/w/index.php?title=OpenGL&oldid=675625638. 3

Wikipedia. Serialization, 2015e. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Serialization&oldid=675153441. 18

C. M. Wittenbrink. R-buffer: A Pointerless A-buffer Hardware Architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 73–80. ACM, 2001. 50

N. Wong and S. Carpendale. Supporting Interactive Graph Exploration Using Edge Plucking. In *Proceedings of IS&T/SPIE 19th Annual Symposium on Electronic Imaging: Visualization and Data Analysis*, volume 6495, pages 08–12. SPIE and IS&T, 2007. 95

N. Wong, S. Carpendale, and S. Greenberg. EdgeLens: an interactive method for managing edge congestion in graphs. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 51–58. IEEE, 2003. 95, 107

B. J. N. Wylie and M. Geimer. Large-scale performance analysis of PFLOTRAN with Scalasca. In *Proceedings of the 53rd Cray User Group meeting*. Cray User Group Inc., 2011. 120

Z. Xue, X. Dong, S. Ma, and W. Dong. A Survey on Failure Prediction of Large-Scale Server Clusters. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 733–738. IEEE, 2007. 69

J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum*, 29(4):1297–1304, 2010. 15, 50, 53, 64, 137

Y. Yu, K. Zhou, D. Xu, X. Shi, H. Bao, B. Guo, and H.-Y. Shum. Mesh editing with poisson-based gradient field manipulation. *ACM Transactions on Graphics*, 23(3):644–651, 2004. 77

J. Zhao, K. Trivedi, Y. Wang, and X. Chen. Evaluation of software performance affected by aging. In *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*, pages 1–6. IEEE, 2010. 72

M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive Level-of-Detail Rendering of Large Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012. 94