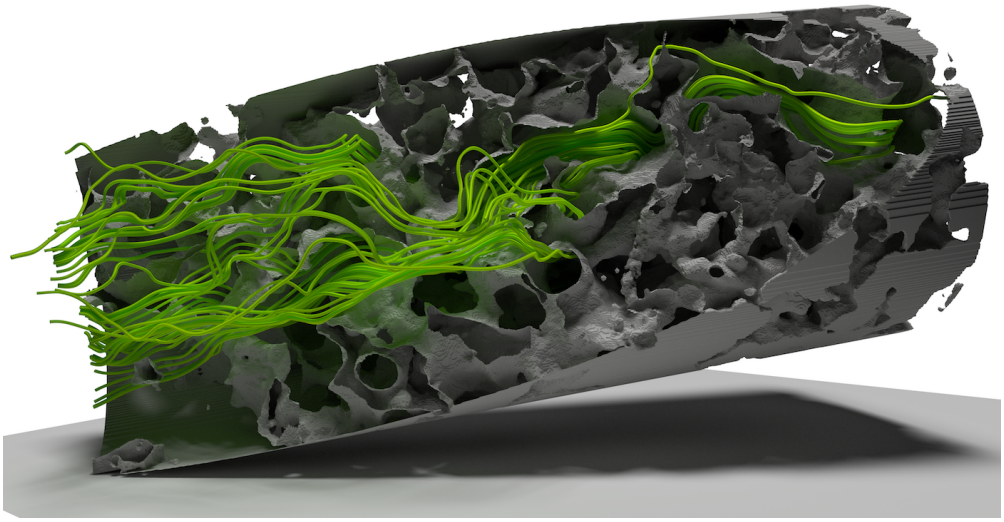
Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Towards Efficient Ray Casting-Based Visualization on Heterogeneous HPC Architectures

Tim Biedert



Dekan	Prof. Dr. Stefan Deßloch
Berichterstatter	Prof. Dr. Christoph Garth
Berichterstatter	Prof. Dr. Hank Childs
Datum der Aussprache	27. August 2019

D 386

Abstract

Visualization is vital to the scientific discovery process. An interactive high-fidelity rendering provides accelerated insight into complex structures, models and relationships. However, the efficient mapping of visualization tasks to high performance architectures is often difficult, being subject to a challenging mixture of hardware and software architectural complexities in combination with domain-specific hurdles. These difficulties are often exacerbated on heterogeneous architectures. In this thesis, a variety of ray casting-based techniques are developed and investigated with respect to a more efficient usage of heterogeneous HPC systems for distributed visualization, addressing challenges in mesh-free rendering, in-situ compression, task-based workload formulation, and remote visualization at large scale.

A novel direct raytracing scheme for on-the-fly free surface reconstruction of particle-based simulations using an extended anisoptropic kernel model is investigated on different state-of-the-art cluster setups. The versatile system renders up to 170 million particles on 32 distributed compute nodes at close to interactive frame rates at 4K resolution with ambient occlusion. To address the widening gap between high computational throughput and prohibitively slow I/O subsystems, in situ topological contour tree analysis is combined with a compact image-based data representation to provide an effective and easy-to-control trade-off between storage overhead and visualization fidelity. Experiments show significant reductions in storage requirements, while preserving flexibility for exploration and analysis. Driven by an increasingly heterogeneous system landscape, a flexible distributed direct volume rendering and hybrid compositing framework is presented. Based on a task-based dynamic runtime environment, it enables adaptable performance-oriented deployment on various platform configurations. Comprehensive benchmarks with respect to task granularity and scaling are conducted to verify the characteristics and potential of the novel task-based system design. A core challenge of HPC visualization is the physical separation of visualization resources and end-users. Using more tiles than previously thought reasonable, a distributed, low-latency multi-tile streaming system is demonstrated, being able to sustain a stable 80 Hz when streaming up to 256 synchronized 3840x2160 tiles and achieve 365 Hz at 3840x2160 for sort-first compositing over the internet, thereby enabling lightweight visualization clients and leaving all the heavy lifting to the remote supercomputer.

Kurzfassung

Visualisierung ist essentiell im wissenschaftlichen Entdeckungsprozess. Eine hochqualitative interaktive Darstellung bietet beschleunigten Erkenntnisgewinn in komplexe Strukturen und Zusammenhänge. Oft ist eine effiziente Abbildung von Visualisierungsverfahren auf Hochleistungsarchitekturen jedoch schwierig aufgrund von Komplexitäten sowohl in Hardware als auch Software, sowie domänenspezifischer Herausforderungen. Heterogene Architekturen verschärfen diese Schwierigkeiten. In dieser Arbeit werden verschiedene Raycasting-basierte Techniken im Hinblick auf die effiziente Nutzung heterogener HPC Systeme für die verteilte Visualisierung vorgestellt und untersucht. Dies umfasst Herausforderungen in gitterfreier Darstellung, in-situ Kompression, taskbasierter Algorithmusformulierung, sowie verteilter Remote-Visualisierung.

Ein neues direktes Raytracing-Schema zur Rekonstruktion freier Oberflächen partikelbasierter Simulationen basierend auf einem erweiterten Modell anisotropischer Kernels wird auf verschiedenen aktuellen Clustersystemen untersucht. Es können bis zu 170 Millionen Partikel auf 32 verteilten Rechenknoten nahezu interaktiv bei 4K-Auflösung mit Ambient Occlusion dargestellt werden. Hinsichtlicher der wachsenden Lücke zwischen Rechendurchsatz und langsamen I/O Subsystemen wird insitu topologische Kontourbaum-Analyse in Kombination mit kompakter bildbasierter Datenrepräsentation untersucht. So lässt sich auf effiziente und leicht kontrollierbare Weise der Schwerpunkt zwischen Speicherkosten und Visualisierungsflexibilität steuern, wobei Experimente signifikante Reduktionen der Datenmengen belegen. Motiviert durch eine zunehmend heterogene Systemlandschaft wird ein flexibles Task-basiertes Framework zum verteilten direkten Volume Rendering und Compositing vorgestellt. Auf Basis einer Task-basierten dynamischen Laufzeitumgebung werden Task-Granularität und Skalierungscharakteristika der neuartigen Algorithmusformulierung untersucht. Eine zentrale Herausforderung der HPC Visualisierung ist die räumliche Trennung zwischen Visualisierungsressourcen und Endanwender. Es wird ein verteiltes Multi-Tile Streaming-System vorgestellt, welches durch niedrige Latenzen und effiziente Hardware-Kompression bis zu 256 synchronisierte 4K-Streams bei stabilen 80 Hz verarbeitet, und bis zu 365 Hz für 4K Sort-First Compositing über das Internet bietet.

Acknowledgement

First and foremost I offer my sincerest gratitude to my supervisor, mentor and worst critic, Christoph Garth, who has supported me throughout this thesis with his knowledge, creativeness, and passion for nerdy technical details. Likewise, I thank my initial mentor, Hans Hagen, who has motivated and guided me during my early days in the department's PhD program.

A considerable amount of research in this thesis would not have been possible without the ample contributions from my research assistants Kilian Werner and Jan-Tobias Sohns, who have always striven for rock-solid and efficient implementations to strengthen our scientific endeavors.

At the same time, this work has profited enormously from valuable feedback and fruitful discussions with my co-authors Simon Schröder of Fraunhofer ITWM, Bernd Hentschel of RWTH Aachen University, Ingo Wald and Jefferson Amstutz of Intel Corporation, as well as Peter Messmer and Tom Fogal of NVIDIA Corporation, who are now my colleagues.

Regarding daily work I am grateful for enjoyable and fun times with my friends and office mates Tobias Gauweiler, Mathias Hummel and Jonas Lukasczyk. A (computer administration) sorrow shared is a sorrow halved. Also, I wonder who has eaten the largest amount of fries in the university's canteen?

Finally, I thank my parents and family for supporting me throughout all my studies and activities at university. Most importantly, I am eternally grateful for my loving and supporting wife, Elena, our wonderful baby son, Demian, and our remarkably crazy dog, Rocky, all of whom provide unending inspiration. I will never forget baby sling-carrying three-week-old Demian on my chest while finishing the fluid surfaces paper mere hours before the deadline.

Contents

1 Introduction

1.1 High Performance Computing

Microprocessors constitute the foundation of computational sciences and have seen a rapid development over the past decades, having given rise to diverse disciplines building upon the rich execution capabilities of increasingly powerful and complex processing units. Their performance is characterized by transistor speed, energy scaling and system integration density. Increases in transistor density by approximately 35% per year are complemented by additional growths in the overall die sizes of about 10% to 20% per year. These two factors combined result in annual transistor count increases of 40% to 55% [Chi14]. This trend was captured by Gordon Moore, the cofounder of Intel Corporation, who predicted that the number of transistors on an integrated chip would double roughly every two years [Ste07]. Typically referred to as *Moore's law*, this observation and prediction has proven uncannily accurate for the past five decades.

With increasing operational frequencies and architectural advances exploiting the growth in available transistors according to Moore's law, new microarchitectures have been empirically characterized by *Pollack's rule* [BC11], which broadly captures the area, power, and performance tradeoffs from several generations of microarchitecture. According to Pollack, if not limited by other parts of the system, performance increases as the square root of the number of transistors or area of a processor. Thus, each new generation doubling the number of transistors on a chip enables a new microarchitecture delivering a performance increase of 40%. An additional 40% performance improvement due to generally increased transistor frequencies results in an overall approximate two-fold performance increase within the same power envelope.

However, implementing a new microarchitecture every generation is difficult in practice and processor frequencies are close to hitting a solid wall due to heat dissipation constraints. Driven by an ever increasing demand for energy efficiency, the focus of chip designers has shifted from microarchitectural techniques towards alternate approaches. In contrast to optimizing a single large monolithic core, multicore microarchitectures have proven to be a lucrative design choice, where each

core delivers lower performance than a larger complex core, but the total compute throughput of all cores combined increases significantly. While multi-core systems can compute at increased throughput, their effective utilization at full potential is challenging. Many scientific codes and applications often provide opportunities for concurrent execution, but the overall performance gain through parallelization is ultimately limited by the sequential critical sections in a program. This relationship between potential parallel speed up and the fraction of code which benefits from a system's resource improvements is commonly known as *Amdahl's law* [Amd67].

High performance computing (HPC) takes the idea of increased computational throughput through hardware parallelism to the extreme by implementing concurrency and parallelism along several axes: A modern supercomputer consists of thousands of compute nodes, each with several multi- or many-core processors per node, which in turn might even utilize instruction-level parallelism and vectorized instructions. The latter is typically referred to as *single instruction multiple data (SIMD)* according to *Flynn's taxonomy*. There are fundamental similarities to a basic personal computer in terms of general hardware components (control hardware, computational units, internal memory, communication, mass storage, input/output channels) and software components (operating system, drivers, file systems, compilers, tools). However, the distinctiveness of an HPC system which distinguishes it from a conventional computer is the structured organization, interconnectivity and scale of the components, combined with the ability to manage the operation of such a system at large scale.

Today, the broader ecosystem around high performance computing is a vibrant multi-billion dollar market, projected to grow to more than $30 billion by 2020, with a compound annual growth rate of 8%. The fast-growing market is driven by end-user demand in a multitude of application domains, such as earth and life sciences, natural laboratories, oil and gas industry, manufacturing, financial services and government intelligence. The underlying mathematical models and techniques cover a wide sprectrum, including linear algebra (search engines, finite-element simulations, climate modeling), the solution of partial-differential equations (weather prediction, hurricane modeling, sea-ice modeling, oil reservoir modeling, compressible flow computation), large systems with pair-wise force interactions (cosmology, molecular dynamics simulations, medicine development, biomolecular development, plasma modeling), graph problems (machine learning, data analytics, fraud detection) and stochastic systems (radiation transport, particle physics, nuclear reaction design, risk analysis in finance, public health, disease spread modeling) [SAB17].

1.1.1 Heterogeneous Architectures

The vast amount of different application domains and underlying scientific models inherently entails a multitude of diverse requirements towards the computing system at hand. While standard CPUs can be considered a general-purpose architecture with high versatility, they are far from being a universally optimal architecture for all domains. In modern CPUs, a large portion of the overall die area consists of complex control flow units and caching hierarchies, leaving not much space for actual computational units, i.e., algorithmic logic units (ALUs).

The traditional approach in the everlasting pursuit of increasing an HPC system's total peak performance has been to simply scale the number of CPUs and combine them with a high-speed interconnect. However, in recent years the subject of power consumption has become a critical issue, with modern supercomputers consuming almost up to 20 MW [Top]. With globally rising energy prices, this results in excessive maintenance costs over a system's lifetime, which typically exceed its initial costs of acquisition. This development and the need for specific computational capabilities has given rise to heterogeneous supercomputer architectures, where a general-purpose CPU is combined with specialized accelerator hardware. While the former is particularly suited for low-latency processing of complex control flows seeking to optimize the execution of sequential programs, the latter typically enables high throughput through massive hardware parallelization. Thus, heterogeneous architectures are designed to maximize throughput for specific applications or parts of codes, thereby drastically improving the ratio of computational performance to overall power consumption.

Graphics processing units (GPUs) are inherently suited for massively parallel operations, e.g., the parallel processing of large vertex counts or parallel rasterization and processing of the resulting fragments. The term *embarassingly parallel* is often used in this context to describe workloads where little to no effort is required to separate the problem into many parallel tasks. Typically, these workloads are enabled by no or negligible amounts of interdependencies or communication needs. With the rise of general-purpose computing on graphics processing units (GPGPU), developers can now directly harness the parallel compute capabilities of such devices by implementing arbitrary parallel programs in high-level languages and frameworks such as NVIDIA's CUDA or the vendor-neutral OpenCL standard. NVIDIA has coined the phrase *single instruction multiple threads (SIMT)* for their many-core GPU architectures, which virtually contain several types of parallelism, i.e., multithreading, multiple instruction multiple data (MIMD), single instruction multiple data (SIMD), and instruction-level parallelism [CGM14]. From an architectural point of view,

CUDA is based on a scalable array of multithreaded Streaming Multiprocessors (SMs), where each SM contains a specific number of CUDA cores and CUDA programs are executed in so-called *warps* of 32 threads per SM. Several modern supercomputers make use of NVIDIA GPUs as accelerator cards [Top]. Parts of this thesis have been conducted on the Piz Daint supercomputer at Swiss National Supercomputing Centre (CSCS), which at the time of writing consists of 5320 compute nodes with a 12-core CPU and a NVIDIA Tesla P100 accelerator card, each providing 56 Streaming Multiprocessors and 3584 CUDA cores in total [NVI16].

Originally based on an earlier GPU design by Intel, the Xeon Phi series is another popular accelerator choice for parallel workloads in modern HPC systems. In contrast to modern GPUs, the Intel Xeon Phi is a manycore architecture based on the common x86 instruction set, thereby aiming for easier development and portability of existing codes. While originally designed as a PCI Express-connected coprocessor card, the second-generation Xeon Phi models codenamed Knights Landing (KNL) now constitute a standard standalone processor that can boot an off-the-shelf operating system. Furthermore, KNL is binary compatible with prior Intel Xeon server processors, enabling legacy software to run on the manycore platform without modifications. While getting functional code is rather straightforward as promoted, simple data structures and massive parallelism are critical for Xeon Phi to perform well. Without proper compiler-assisted parallelization and vectorization, efficient programming for Xeon Phi is still a challenging task [Fan+14]. Parts of this thesis have been conducted on the Stampede2 supercomputer at Texas Advanced Computing Center (TACC), which houses 4200 Intel Xeon Phi Knights Landing nodes, each with 68 cores on a single socket supporting 4 hardware threads per core and interconnnected by a mesh of rings [Sod+16].

1.1.2 Performance and Scaling

The fundamental defining property and ultimate value provided by a high performance system is large-scale performance. Without proper definition, performance as a quantitive measure can be ambiguous, as there are different objective and subjective meanings and interpretations to it. There are several concepts of performance which have in common that they relate time to some unit of work.

The *peak performance* of a system denotes the theoretical maximum rate at which operations can be performed by the hardware resources, i.e., a combination of clock rate and hardware parallelism. Peak performance is usually measured in floating-point operations per second (FLOPS), which in contrast to instructions

per second (IPS) provides a better bearing on the arithmetic capability of a machine in the context of scientific computing. With current systems in the lower petaFLOPS range, peak performance is anticipated to hit exaFLOPS by the end of this decade [SAB17].

In practice however, the actual or real performance that can be achieved on a supercomputer is often much less than theoretical peak performance. The so-called *sustained performance* is considered a more pragmatic indicator of an HPC system's true capabilities based on more realistic workloads. Sometimes also referred to as *wall clock time* or *time to solution*, the sustained performance represents the total average performance of program over its runtime, since the momentary performance of a program can vary throughout execution due to variable system circumstances. Similar to peak performance, sustained performance is typically expressed in floating-point operations per second, but can capture any unit of interest depending on the context, e.g., integer, load/store, or conditional operations per second. Clearly, measuring sustained performance for comparison purposes is only uses if it is based on standardized measurement approaches. The *LINPACK* benchmark and its portable implementation *HPL* [DLP02] are widely employed and referenced, serving as a baseline for the *Top 500* list, which tracks the world's fastest supercomputers on a semiannual basis [Top]. For this task, HPL generates a random dense linear system of equations of order n and solves it using LU decomposition with partial row pivoting. The Top 500 list demonstrates that a large number of HPC systems are in fact heterogeneous architectures using commercial off-the-shelf hardware.

The relationship between the performance and the size of a high performance system is denoted by *scaling*, an important measure often used to describe possible gains achievable through the utilization of larger hardware setups. A typical measure of system size in this setting is the total number of cores, independent of their specific architectural organization which can actually have significant impact on performance. Given a fixed problem size, *strong scaling* measures how the time to solution improves with increased system size. In this case, cutting the execution time in half by utilizing twice as many cores would be considered ideal strong scaling. In contrast to strong scaling, *weak scaling* assumes a fixed problem size per core, i.e., the overall data size grows proportionally with the size of the system. Ideal weak scaling describes using twice as many cores to solve a twice as large problem size in the same time frame. In practice, weak scaling is often more relevant as it can be used as a measure to describe how much money needs to be invested into a supercomputer to be able to solve a problem of a specific size. However, the total amount of main memory does not grow proportionally to system size due to costs. Thus, the amount of memory per core has been decreasing, limiting the

opportunities for weak scaling [SAB17]. In this thesis, both strong and weak scaling are studied for all benchmarks conducted.

There is a wide spectrum of possible reasons for the discrapency between theoretical peak performance and actually measured sustained performance. Mostly inherent to the hardware architectures and programming models, performance-limiting factors typically have in common the failure to effectively and efficiently exploit available resources in their entirety. In general, the negative impact of bottlenecks is more noticable when targeting larger machines, thus limiting scalability. Performance degradation is formally often referred to through the acronym *SLOW*, identifying its reasons as **s**tarvation, **l**atency, **o**verhead, and **w**aiting for contention [Ste+14].

Starvation refers to an insufficiency of available concurrent work to maintain a high utilization of all resources. Typical causes for starvation are either not enough parallelism exhibited by the application or load imbalance due to workloads not being distributed evenly. Latencies are imposed by the time-distance delay intrinsic to accessing remote resources and services, such as memory accesses, data transfers between nodes or the overall length of an execution pipeline. Possible approaches to solve latency-based performance degradation include reducing latency through locality or hiding latencies by not letting computational resources run idle. Common hardware-architectural designs to mitigate the effects of latency include cache hierarchies and multithreading hardware. Overhead refers to the additional work required for the management of parallel tasks and resources on the critical execution path, which would otherwise not be necessary in a purely sequential variant of the algorithm at hand. Examples for overhead include resource management, thread scheduling and synchronization. Waiting for contention resolution denotes delay due to lack of available of oversubscribed shared resources, such as bank conflicts in memory or insufficient communication bandwidth.

1.1.3 Challenges in HPC Visualization

Visualization is vital to the scientific discovery process, where the ideally interactive viewing of a high-fidelity rendering or animation can provide accelerated insight into complex structures, models and relationships. However, the efficient mapping of not only simulation codes but also visualization tasks to homogeneous and in particular heterogenerous high performance architectures is often difficult, as they are naturally also subject to performance degradation due to starvation, latency, overhead and waiting for contention. The growing field of *high performance visualization* addresses this interesting and challenging mixture of general hardware and software

architectural complexities in combination with domain-specific hurdles [BCH12]. This section provides a brief overview over common difficulties, trade-offs and challenges in visualization on large-scale HPC systems, which have given rise to the solutions developed and investigated in this thesis.

At the current scale of computational capability provided by large-scale parallel computer architectures such as commodity clusters and modern supercomputers, high-fidelity computational simulation models have assumed a significant role in scientific research and engineering applications. However, this increased amount of computation has incurred architectural trade-offs. While arithmetic capacity and in-core memory have grown at a tremendous rate, I/O subsystems have not been able to keep abreast in relative bandwidth [Chi+10]. As a consequence, numerical data produced during typical simulations cannot be persistently stored, e.g. to hard drives, in its entirety; a lack of available I/O bandwidth would make this prohibitively costly with respect to time. This limitation has given rise to so-called *in situ* visualization techniques, which aim to perform visualization directly on the compute system in tight connection with the underlying simulation. By storing only rendered images or other resulting artifacts [Ahr+14], in situ techniques trade smaller output size for reduced post-processing flexibility.

The handling of large-scale computational simulation models does not only pose a challenge to the efficient utilization of HPC hardware subject to its architectural trade-offs and limitations. An increasingly heterogeneous system landscape in modern high performance computing requires the efficient and portable adaption of performant algorithms to diverse architectures, which is often conflicting with traditional software frameworks and design practices. In recent years, parallel algorithms for concrete classes of visualization problems have been presented, such as direct volume rendering [HBC12] or integral curve computation [Pug+09]. Most large data approaches typically utilize a distributed memory model, where bulk-synchronous execution and communication using the Message Passing Interface (MPI) is standard. For improved scalability, hybrid approaches commonly resort to MPI for the coarse distribution of parallelly executable parts of an algorithm to a set of processes, where within each process a second concept - e.g. OpenMP, OpenCL, or CUDA - is used for additional finegrained parallelization of these steps. These practices require detailed knowledge of the different parallelization concepts and often result in specific optimizations for certain platform configurations or obligating the usage of distinct hardware components, which complicates or even hinders portability towards other architectures.

The parallelization of visualization concepts poses an additional difficulty by the fact that, in contrast to simulation computations, visualization tasks are frequently bandwidth-limited and inherently unbalanced. Thus, achieving scalable parallel execution demands not only an efficient utilization of the available memory bandwidth, where memory accesses ideally are overlapping with computational tasks, but also dynamic load balancing. A classic example for a visualization workload highly sensitive to load imbalance is distributed integral curve computation, where the path of each seeded streamline is dependent on the vector field at hand to be visualized and interpreted, prohibiting an a priori computation or estimate of a well-balanced seed or data block distribution.

Another core challenge of HPC visualization is the physical separation of visualization resources and end-users, which is becoming increasingly relevant given the rise and popularity of streaming-based solutions over the internet, such as video on demand, live video and game streaming, or even interactive remote gaming. With increasing dataset sizes, in situ scenarios, and complex visualization algorithms, transfer to a separate visualization system becomes impractical. Not only are transfers of large-scale simulation results prohibitively costly with respect to time, but dedicated visualization systems are typically not able to scale with the size of the compute system. Modest demand for interactivity, low screen resolutions and user bases on relatively high-speed connections made frame based compression sufficient to provide a workable remote visualization experience in previous settings. However, with novel interactive workflows, commodity high-resolution monitors, complex rendering algorithms, latency sensitive display technologies and globally distributed user bases, new approaches to solve the remoting challenge are required.

1.2 Contributions

Ray casting-based techniques represent a specific subset of visualization methodology, where rays are intersected typically with a model or data set to sample, accumulate or compute per-pixel values such as output colors. In this thesis, several approaches based on ray casting are developed and investigated with respect to a more efficient usage of heterogeneous high performance computers for visualization workloads, addressing the aforementioned challenges in HPC visualization.

Particle-based simulation models have assumed a significant role in the numerical computation of complex and highly dynamic transient flow and continuum mechanical problems. However, direct visualization of surfaces from particle data without

intermediary discrete triangulation remains a challenging task. In this thesis, a novel direct raytracing scheme for on-the-fly free surface reconstruction is presented, building upon the rich anisotropic kernel approach, which is adapted and tuned to the surface definition of FPM-based fluid simulations. The improved anisotropic kernel-based surface definition incorporates automatic kernel scaling for variable smoothing lengths, provides intuitive visuals for isolated particles, and is easily parallelized. For this surface definition, a novel direct ray tracing scheme is described. This on-demand two-pass iterative sampling algorithm intelligently reduces intersection candidates for both opaque and transparent surface rendering, provides optimization opportunities for secondary rays, and allows the dynamic mapping of particle attribute values on to the surface using arbitrary transfer functions. By reducing the number of candidate kernels evaluated to converge to the surface threshold, the approach runs in image space rather than object space complexity. Based on an implementation within the OSPRay raytracing framework, comprehensive benchmarks are conducted and analyzed in order to quantify preprocessing and rendering times on different state-of-the-art hardware setups, including workstation, standard cluster and Xeon Phi accelerator systems. Furthermore, the applicability of the approach to a variety of medium and large scale FPM data sets is demonstrated. The system is suitable for both high quality and interactive desktop rendering, scales reasonably well even with trivial parallelization and renders up to 170 million particles on 32 distributed compute nodes at close to interactive frame rates at 4K resolution with ambient occlusion.

While high-fidelity simulation models on large-scale parallel computer systems can produce data at high computational throughput, modern architectural trade-offs make full persistent storage to the slow I/O subsystem prohibitively costly with respect to time. In this thesis, the feasibility and potential of combining in situ topological contour tree analysis and compact image-based data representation is demonstrated to address this problem. Based on in situ contour tree analysis and simplification, a segmented representation of the scalar fields contained in the simulation data at every time step is obtained. A rendering of this segmentation is then generated describing all components visible in every pixel (similar to an A-buffer), and stored together with the simplified contour tree. These ingredients can then be used in post-analysis to flexibly select specific subsets of the segmentation, after further simplification if required. Several experiments are conducted to quantify the I/O savings prossible from such an approach, showing significant reductions in storage requirements using topology-guided layered depth imaging, while preserving flexibility for explorative visualization and analysis. Intended as a baseline demonstration, the presented technique highlights the feasibility and potential of the

combination of topological analysis and image-based representation in large-scale in situ scenarios, and represents an effective and easy-to-control trade-off between storage overhead and visualization fidelity for large data visualization.

An increasingly heterogeneous system landscape in modern high performance computing requires the efficient and portable adaption of performant algorithms to diverse architectures. However, classic hybrid shared-memory/distributed systems are designed and tuned towards specific platforms, thus impeding development, usage and optimization of these approaches with respect to portability. In this thesis, a flexible parallel framework for distributed direct volume rendering is demonstrated. Built upon a task-based dynamic runtime environment, it enables adaptable performance-oriented deployment on various platform configurations. The novel task-based definition aims to provide a flexibly tunable task granularity by subdividing in both image and data space, thus yielding a hybrid scheme between sort-first and sort-last compositing. Based on an asynchronous binary tree compositing scheme including optimizations such as empty space skipping and early ray termination, the technique enables good scalability in combination with inherent dynamic load balancing. Internally, the approach is based on the High Performance ParalleX (HPX) framework, an aspiring task-based runtime environment supporting asynchronous communication across nodes. Each block is represented as an individual component in the active global address space (AGAS), allowing blocks to directly communicate in the compositing pattern. A custom priority queue scheduler is implemented on top of HPX's standard FIFO scheduler by manually keeping track of the number of rendering and compositing tasks being executed by HPX. Per-block rendering is performed using the OSPRay framework, whereas manual AVX2 instrinics are used for accelerated vectorized blend-over image compositing. Comprehensive benchmarks with respect to task granularity and scaling are conducted to verify the characteristics and potential of the novel task-based system design for high performance visualization.

The growing use of distributed computing in computational sciences has put increased pressure on visualization and analysis techniques. In this context, a core challenge of HPC visualization is the physical separation of visualization resources and end-users. While GPUs are routinely used in remote rendering on GPU-accelerated heterogeneous supercomputers, a heretofore unexplored aspect is these GPUs' special purpose video encoding/decoding hardware that can be used to solve the large-scale remoting challenge. The high performance and substantial bandwidth savings offered by such hardware enable a novel approach to the problems inherent in remote rendering, with impact on the workflows and visualization scenarios available. Using more tiles than previously thought reasonable, in this thesis, a distributed,

low-latency multi-tile streaming system is demonstrated, being able to sustain a stable 80 Hz when streaming up to 256 synchronized 3840x2160 tiles and achieve 365 Hz at 3840x2160 for sort-first compositing over the internet. Intended as a comprehensive case study, the impact of video compression and multi-tile streaming based on the H.264/HEVC codec family is investigated in order to address a multitude of novel use cases, such as directly streaming content from a cluster to remote large-scale tiled displays at sufficiently high frame rates, or how to strong-scale the rendering and delivery task by using video hardware to accelerate direct-send sort-first compositing. Using hardware-accelerated multi-tile streaming, traditional dedicated visualization clusters or workstations can be reduced to mere thin clients that leave the heavy lifting to the remote supercomputer.

1.3 Structure

The structure of this thesis is as follows. Intended for audiences with a general background in computer science, Chapter 2 covers the necessary fundamental concepts and relevant prior work in the field of scientific visualization necessary to understand the research presented in this thesis. This includes introductions to ray casting, volume rendering and transfer functions, image compositing techniques, topology-based data analysis, parallelization techniques in visualization, and a brief outline of lossy video compression.

Chapters 3 through 6 contain the major contributions of this thesis, i.e., direct raytracing of particle-based fluid surfaces, combining in situ topological data analysis with image-based storage as contour tree depth images, a novel task-based formulation of distributed volume rendering, and a comprehensive case study on hardware-accelerated multi-tile streaming. Each chapter is introduced with a brief motivation and overview of the state of the art in the respective field, followed by the theoretical and methodological concepts of the approach presented. After a section about implementation details providing helpful insights for accurate technical reproduction, a comprehensive results section covers the performance and scaling characteristics of the topic at hand. Each chapter is concluded with a short summarizing discussion and potential future work building upon the research outlined.

The overall conclusion of this thesis as well as directions for future research are presented in Chapter 7.

2 Background

This chapter serves as a brief introduction to the fundamental concepts in scientific visualization and related fields on which the contributions presented in this thesis build on. While this is intended as a general overview to the overall field, each main chapter features a dedicated section about important related work and the state of the art in the respective specialized research area.

2.1 Ray Casting

The central goal of computer graphics is simulating the distribution of light in a typically three-dimensional scene. In modern applications, there are only a few fundamentally different algorithms, which can loosly by classified into projective and image-space algorithms [Suf07]. Implemented on all modern graphics cards, the so-called *rasterization* approach belongs to the former category by projecting geometric primitives such as triangles onto the image plane and applying local shading models to the resulting pixel fragments, thus being amenable to pipeline processing and highly efficient hardware implementations. In contrast, image-space algorithms compute pixel colors by approximating the light transport which contributes to each pixel. The name of the popular *ray tracing* image synthesis algorithm reflects the core operations which it is based on, i.e., determining the nearest objects long lines of sight and following light along rays through the scene. Since the 1980s, ray tracing has been considered an eye opener due to its capability to produce high quality images with naturally included global light paths such as specular reflection and transmission, which in particular are difficult to compute using projective algorithms. However, due to its enormous computational complexity, ray tracing has long been restricted to high-fidelity offline rendering scenarios such as movie production. More than 35 years after its introduction, ray tracing has eventually found applicability in interactive and real-time rendering, enabled by modern high-performance hardware, parallel processing, sophisticated acceleration structures and state-of-the-art algorithms in combination with low-level engineering and optimization [Suf07].

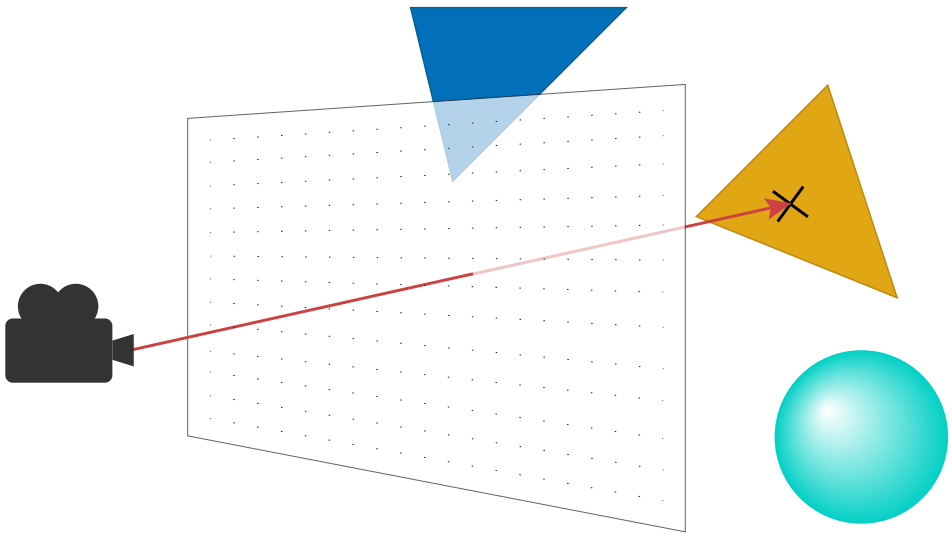


Fig. 2.1: In ray casting-based algorithms, e.g., ray tracing or direct volume rendering, rays are sent from the virtual camera through each pixel of the screen and intersected with the scene.

The fundamental concept of ray tracing is called *ray casting*, a term which was originally introduced by Scott Roth in 1982 in the context of rendering constructive solid geometry models [Rot82]. While often used interchangeably in computer graphics literature, ray casting and ray tracing can and should be distinguished. Essentially, ray tracing builds upon ray casting by recursively tracing secondary rays such as shadow rays or reflections, refractions and transmissions to approximate global light transfer in a scene. Thus, ray casting can be considered performing only the first iteration of primary ray casting in the ray tracing mindset. However, in general, the term ray casting is used to refer to a variety of problems in computer graphics and computational geometry. Besides the non-recursive ray tracing algorithm using primary rays only, this includes the general problem of determining the nearest object intersected by a ray, performing hidden surface removal based on finding the closest intersection of a ray cast from the eye through each pixel of an image, or direct volume rendering using volume ray rasting, where a three-dimensional scalar data is sampled along rays to accumulate color and opacity values using so-called *transfer functions*.

The basic structure of ray casting for image synthesis is illustrated in Figure 2.1. Depending on the context and problem at hand, the scene to be rendered typically consists of an arbitrary number of objects made of primitives such as triangles or implicit surfaces, in combination with materials specified for each object, and light sources to illuminate the scene. Given a virtual image plane whose surface is

covered with pixels, for each pixel a ray is shot (or cast) towards the objects in the scene through the center of the pixel. The nearest hit point is computed, which is often assisted with an acceleration data structure to speed up intersection candidate traversal by quickly discarding irrelevant objects. If a closest hit exists, the respective object's material is used in conjunction with all light sources of influence to compute the pixel's output color. Otherwise the pixel is set to a predefined background or miss color.

Note that the rays used in ray casting (and ray tracing) differ from real light rays or photons as they travel in the oppositive direction. The reason for this becomes clear when considering a perspective pinhole camera, where rays originate at the pinhole. Starting rays from the camera is the only practical way to render images, as the vast majority of rays started from the light sources would never pass through the camera's infinitely small pinhole. [Suf07]

All research topics covered in this thesis are utilizing some form of ray casting. The on-the-fly surface reconstruction of particle-based fluids (Chapter 3) is used in conjunction with standard ray tracing for surface intersection and global lighting including shadowing and ambient occlusion. In contrast, in the subsequent combination of in situ topological data analysis with image-based storage (Chapter 4), a variant of volume ray casting is used to idenfity all entities in the topological data segmentation contributing to each image pixel. Similarly, the task-based formulation of distributed volume rendering (Chapter 5) is based on standard block-wise volume ray casting for each image tile. In the comprehensive hardware-accelerated multi-tile streaming case study (Chapter 6), an extension of ray tracing called *path tracing* is highlighted as a show case amenable to distributed sort-first compositing.

Path tracing is a Monte-Carlo rendering method approximating a numerical solution to the integral of the so-called *rendering equation* [Kaj86], which adheres to the three principles of optics: global illumination, equivalence (reflected light is equivalent to emitted light), and direction (reflected light and scattered light have a direction). Similar to ray tracing, rays are cast from the eye through the pixels of the image plane into the scene. At the closest hit point, the rendering equation is evaluated using Monte Carlo integration. In order to approximate the required incident radiance, rays are traced into random sample directions, where typically so-called *importance sampling* is used to guide the sampling towards faster convergence by considering the local material properties and light sources. As long as there is a significant amount of radiance transported along a ray, this scheme is applied recursively. An example of path tracing applied to a scientific flow visualization scenario is illustrated in Figure 2.2.

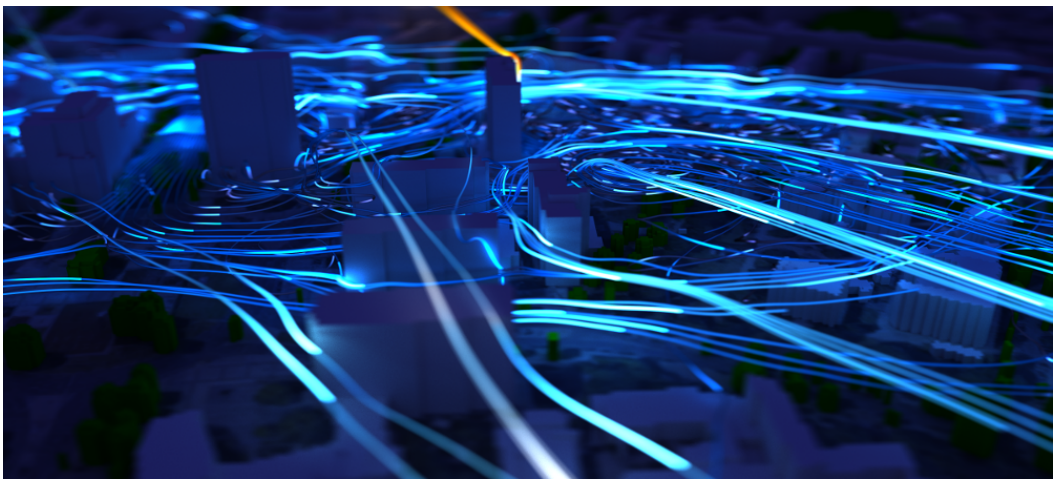


Fig. 2.2: Luminous streamlines rendered with path tracing, enabling advanced visual effects such as emissive materials, global illumination and depth of field. Data provided by Dr. Mohamed Salim from Humboldt University of Berlin, based on simulations with the urban mircroscale model PALM-4U. Rendered with NVIDIA's RTX path tracer for ParaView.

2.2 Volume Rendering

An increased amount of measured and simulated volume data in modern medical and computational sciences necessitates intelligible representations of a volume's interior. Common visualization applications of measured three-dimensional volume data include computed tomography (CT), magnetic resonsance imaging (MRI) and ultrasound scans, whereas simulated volume data can for instance originate from computational fluid dynamic applications and represent physical quantities such as velocity magnitude, pressure or porosity.

Several techniques have been proposed for the visualization of volume data, which can be differentiated based on their dimensionality. A simple but popular two-dimensional approach is *slicing*, where the volume is intersected by a possibly non-aligned planar slice, and the volume data is interpolated accordingly for direct slice display. In contrast, three-dimensional techniques do not simply eliminate one data dimension for rendering and can be classified into indirect and direct solutions. *Isosurface extraction* is called an indirect technique, as an intermediate triangle mesh is constructed for a given isovalue, for instance using the popular marching cubes algorithm [LC87]. In this thesis, variants of *direct volume rendering* are regularly used, which in contrast to isosurfacing is a direct technique without intermediate representations as the name suggests.

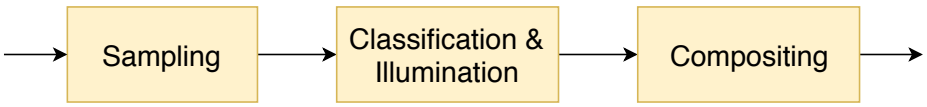


Fig. 2.3: Volume rendering pipeline as described by Levoy.

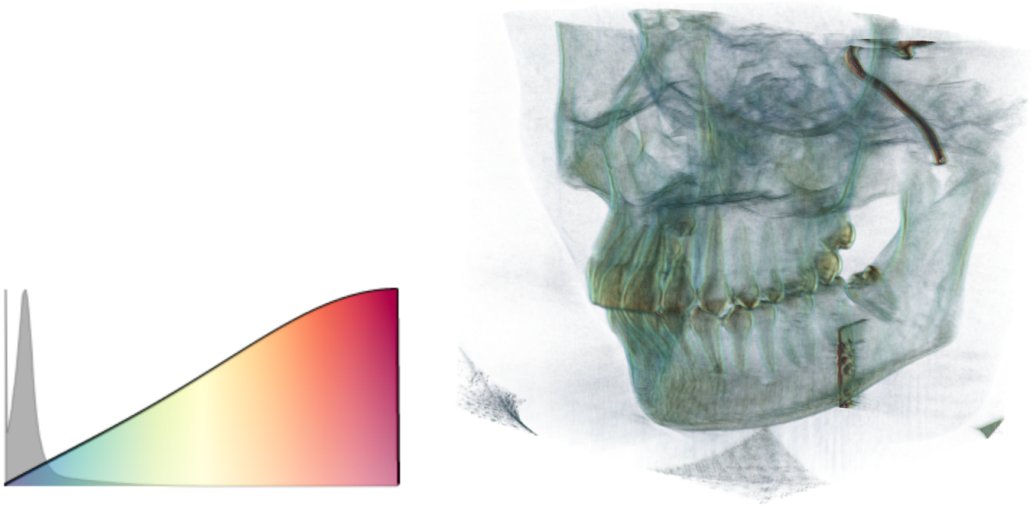
Direct volume rendering [DCH88] represents a crucial class of algorithms used in scientific scalar field visualization. Today, direct volume rendering typically follows the principle of ray casting as proposed by Levoy [Lev90], where primary rays are traced through a volumetric data set starting at a virtual camera and depending on the underlying sample locations optical properties are determined and accumulated along each ray. To systematically capture this, Levoy described a *volume rendering pipeline* consisting of three steps denoted by *sampling*, *classification and illumination*, and *compositing*, which is illustrated in Figure 2.3.

Levoy's approach is built around the goal of numerically approximating the so-called *volume rendering integral*, which is based on the *density-emitter* model introduced by Sabella [Sab88]. The density-emitter model is a simplified light transport model only considering emission and absorption, thereby neglecting more complicated radiative phenomena such as scattering, or wavelength-dependent effects such as diffraction. The volume rendering integral is defined as

$$I(s) = I(s_0)T(s_0, s) + \int_{s_0}^{s} Q(t)T(t, s)dt \tag{2.1}$$

where s denotes a location on a ray traversing the volume and entering it at s_0, $I(s)$ is the resulting intensity computed along the ray up to position s, $I(s_0)$ is the ambient intensity entering the volume, $T(x, y)$ is the accumulated absorption of along all points from x to y, and $Q(t)$ is the isotropic emission at t. Intuitively, in this model each particle is considered a tiny light source emitting light throughout the volume without interacting with it.

The above volume rendering integral describes the intensity along a ray traversing the volume to be visualized. In order to solve the volume rendering integral numerically, contributions have to be accumulated along the ray at discrete locations. This process is called sampling and denotes the first step in the volume rendering pipeline. The straightforward approach is to perform uniform sampling along the ray. However, care must be taken to use enough samples in order to sufficiently represent the resolution of the volume data at hand. The *sampling theorem* states that the data can be reconstructed exactly if the minimal wavelength (maximal frequency) is sampled more than twice. Thus, each cell should be sampled at least once for sufficient accuracy. A possible optimization over uniform sampling is to perform



(a) Transfer Function **(b)** Direct Volume Rendering

Fig. 2.4: Direct volume rendering applied to a computed tomography (CT) scan of a human skull. The transfer function is chosen such that the teeth and bone structures are highlighted, using a high opacity for the low intensity values of the data set (blue/green color). Rendered with ParaView Glance.

adaptive sampling based on the local volume properties, e.g., using empty-space skipping. Typically, for a given sample location, the discrete volume data has to be interpolated. For the regular grids used throughout this thesis, this can be achieved through simple trilinear interpolation.

In the subsequent classification and illumination step, the contribution of each sample to the rendering integral has to be computed. Using the potentially interpolated volume value at the given sample location, the sample value is first classified according to a so-called *transfer function*. Transfer functions allow the user to specify basic rules for color and opacity, thereby providing an intuitive way to define the appearance and transparency of a data region. The process is defined in feature space, i.e., transfer functions typically consist of two continuous one-dimensional functions describing color and opacity for each possible isovalue. The key idea in good transfer function design is that certain function ranges are expected to represent specific materials. Thus, these function ranges should be mapped to distinct visual properties. An example of this approach is illustrated in Figure 2.4, where the transfer function is chosen such that the teeth and bone structures within the CT scan are highlighted.

After classification, the resulting color and opacity values of the sample at hand are used by the lighting model to produce an illuminated and shaded output intensity for improved depth perception. Since the human visual system is trained to perceive

surfaces, Levoy has considered strong gradients in the data to represent object boundaries, thereby interpreting object boundaries as surfaces. Thus, at each sample location, the gradient of the volume data is used as local normal in the lighting model. In the case of regular data, gradients can be computed numerically via finite differences.

Once the sample values are computed, classified and illuminated, they are accumulated along the ray according to the underlying physical model. This process is called compositing and denotes the numerical approximation of the volume rendering integral. The standard approach is to use recursive alpha-blending derived from back-to-front compositing defined as

$$C_i^{'} = C_i + (1 - A_i)C_{i-1}^{'} \tag{2.2}$$

where $C_i^{'}$ denotes the radiant energy observed at position i, C_i is the radiant energy emitted at position i, $(1 - A_i)$ is the absorption at position i, and $C_{i-1}^{'}$ is the radiant energy observed at position $i - 1$. Note that $i = 0$ denotes the foremost sample in reverse viewing direction, i.e., the last sample along the ray. However, when using ray casting it is beneficial to compute the recursion via front-to-back compositing along the ray using

$$C_i^{'} = C_{i+1}^{'} + (1 - A_{i+1}^{'})C_i \tag{2.3}$$

with

$$A_i^{'} = A_{i+1}^{'} + (1 - A_{i+1}^{'})A_i \tag{2.4}$$

describing the accumulated opacity for decreasing values of i. Using front-to-back compositing enables to perform an optimization technique called *early ray termination*, as the ray's opacity is saturated when $A_i^{'} \approx 1$. Thus, based on Equation 2.3, once the accumulated opacity along a ray reaches saturation, the sampling process can be stopped for this ray.

A substantial body of research focuses on both hardware-accelerated and distributed volume rendering of large data, which will be covered in depth in Section 5.2. In this thesis, the combination of in situ topological data analysis with image-based storage (Chapter 4) uses a variant of volume ray casting to intersect the volume and determine sample locations, however does not compute an approximation of the volume rendering integral. Instead, at each sample location, the associated branch in the topological segmentation is determined in order to idenfity boundaries between segments for layered depth image construction. In contrast, the novel task-based formulation of distributed volume rendering (Chapter 5) performs standard direct

volume rendering based on ray casting as outlined above for each data block visible from each image tile.

2.3 Image Compositing

Image compositing is a crucial part of large-scale distributed visualization on high performance computers. Next to I/O, compositing is one of the most expensive operations in parallel rendering pipelines due to increased communication overhead for large node codes [BCH12]. Thus, crucial requirements to image compositing are concurrency and especially scalability.

Following the nomenclature of Molnar et al. [Mol+94], there are three approaches to the implementation of parallel rendering, which are classified based on what is distributed and what is replicated among the processes involved. In *sort-first* rendering, data is replicated across nodes, whereas image pixels are distributed, i.e., every node has the complete data, but renders only a partial tile of the final image. This scheme is often hard to apply in practice, as high-fidelity applications generate data possibly orders of magnitude larger than the memory capacity of a single node. Thus, sort-first compositing is not frequently encountered in HPC visualization. In contrast, *sort-last* rendering is suitable for data which is distributed across nodes, while pixels

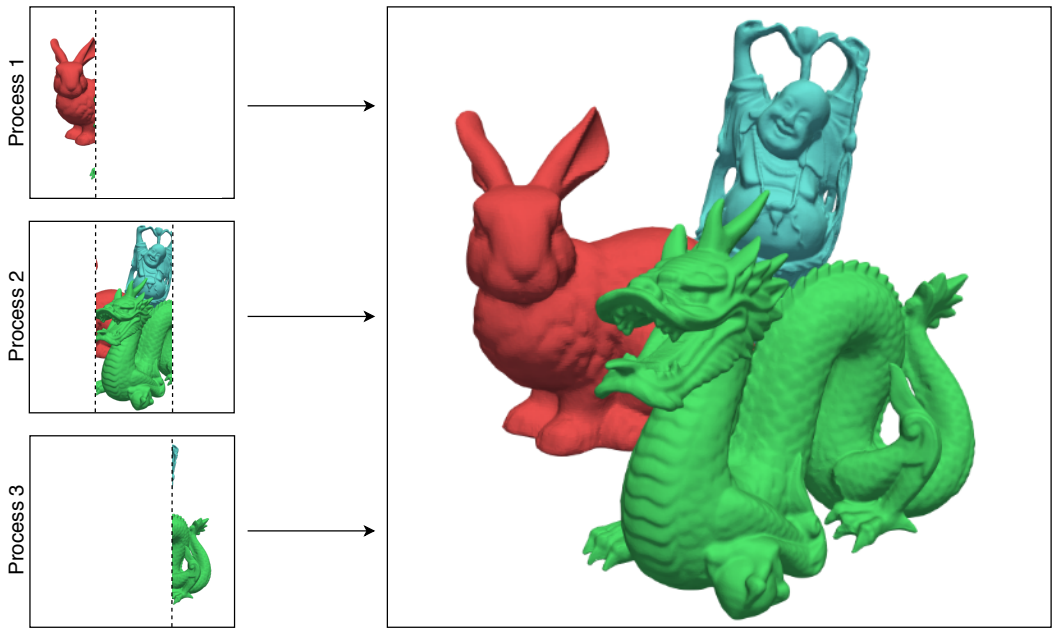


Fig. 2.5: Sort-first compositing. Every node has the complete data, but renders only a partial tile of the final image.

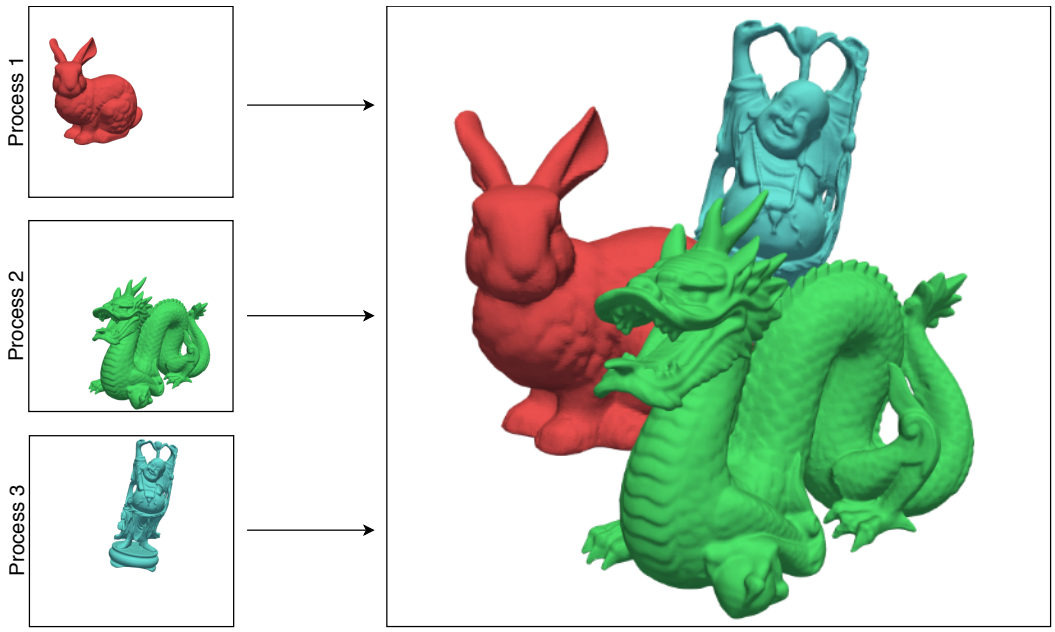


Fig. 2.6: Sort-last compositing. Every node renders a complete full-resolution image of a subset of the overall data.

are replicated, i.e., every node renders a complete full-resolution image of a subset of the overall data. Afterwards, the resulting images need to be combined into a single final output image. This process is called *compositing* and is typically based on each pixel's depth value similar to the conventional Z-buffer algorithm found in rasterization pipelines. *Sort-middle* approaches aim to represent a combination of sort-first and sort-last by distributing both data and image pixels across nodes. However, as the data per node depends on the actual viewpoint and thus requires a redistribution when the camera changes, this approach is considered difficult to implement and scale. Examples of sort-first and sort-last compositing are illustrated in Figures 2.5 and 2.6, respectively.

While both sort-first and sort-last compositing techniques have specific advantages and drawbacks [Mol+94], most scalable systems typically employ the sort-last approach, primarily due to the slower increase in image resolution compared to overall data size. A survey of methods for sort-last compositing can be found in [Sto+03], and an analysis of relative theoretical performance in [CMF05]. The performance and scalability of a multitude of common compositing implementations is demonstrated in the popular *IceT* framework [Mor+11]. Sort-last compositing algorithms can be loosely classified into direct-send and tree-based approaches. The essential concepts and commonly used techniques are outlined in the following.

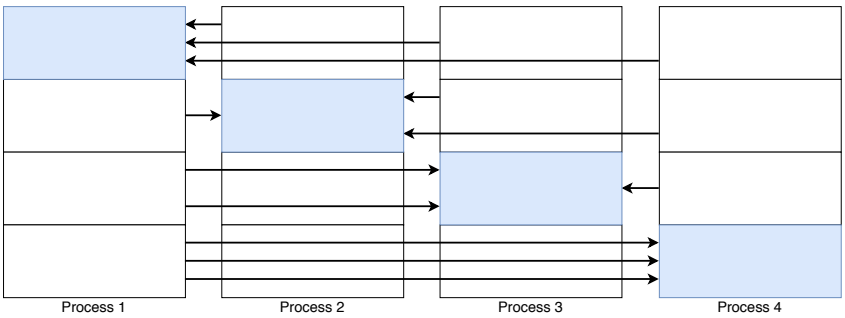
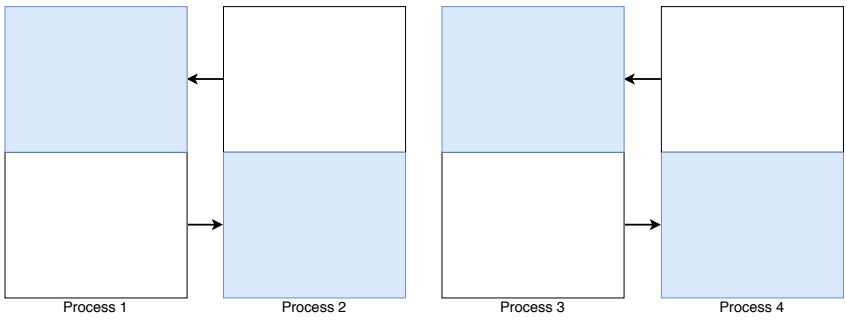


Fig. 2.7: Direct-send compositing using four processes. Each process is responsible for $1/4$ of the final image.

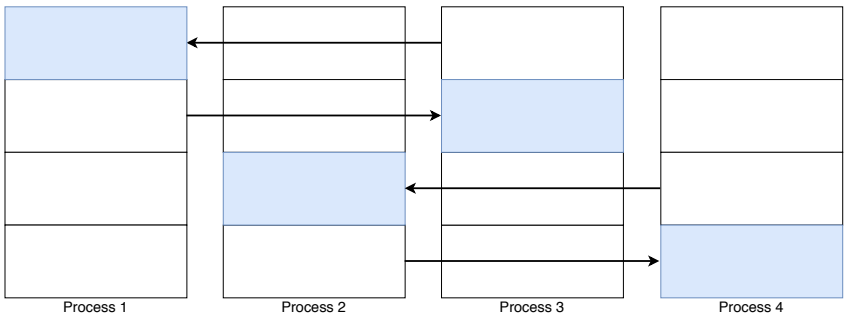
In *direct-send* compositing [Hsu93; Neu94; EP07], each of the p processes is responsible for $1/p$ of the final image. All other processes send this respective image tile to its owner, as illustrated in Figure 2.7. However, direct-send is typically not suitable for large processor counts due to increased network congestion caused by many simultaneous messages. An alternative approach represent tree-based communication patterns, where the levels in the tree structure represent rounds of the tree-based algorithm. In each round, partial images are exchanged between a manageable number of processes forming a group. In contrast to direct-send algorithms, which try to perform as much work as quickly as possible by generating the maximum number of messages in a single round, tree-based approaches generate fewer simultaneous messages over multiple rounds to balance the overall communication load [BCH12].

A well-known tree compositing algorithm is *binary-swap* [Ma+94], which tries to avoid the bottleneck of a growing subset of processes going idle in later rounds. Each process composes the incoming part of an image with the same part of the image it already has, and all processes remain active by continuous swapping. In each round of binary-swap compositing, the distance is doubled, i.e., neighbors are chosen twice as far apart, and the image size is halved. This scheme is illustrated in Figure 2.8 for four processes performing binary-swap over two rounds. Note that binary-swap is the same as direct-send for group sizes of two.

Several optimizations to the direct-send and binary-swap image compositing algorithms have been proposed. A common performance improvement is the size reduction of active image regions by exploiting the spatial locality and sparseness typically present in scientific visualization renderings. Other standard means to faster compositing times are load-balancing and performing improved scheduling in order to keep compute and communication resources busy. Reducing the active image size to minimize communication and compute costs can be achieved through



(a) Round 1

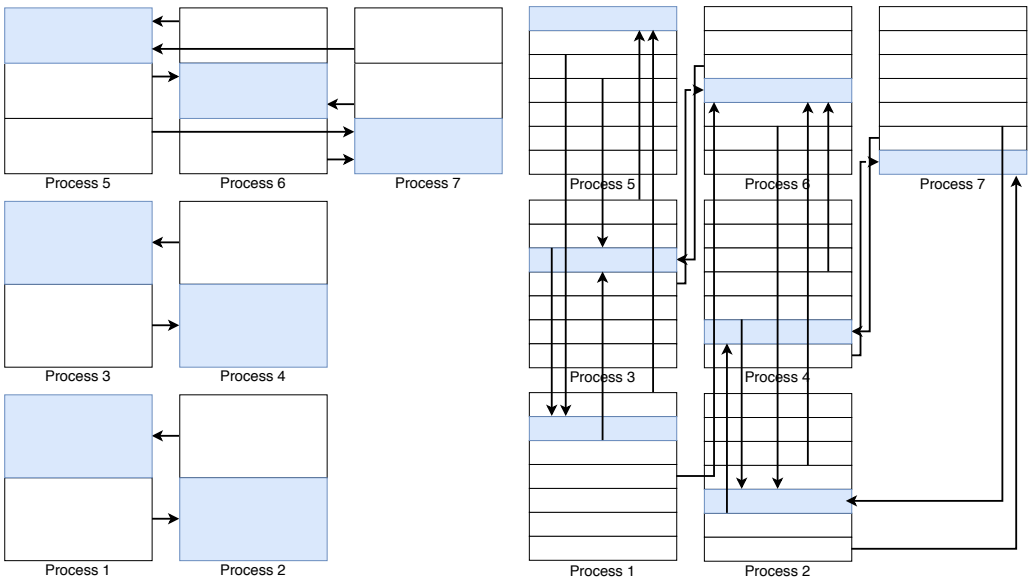


(b) Round 2

Fig. 2.8: Binary-swap compositing using four processes over two rounds. In each round the distance is doubled and the image size is halved.

lossless compression based on run-length encoding [AP98], or identifying bounding boxes of non-empty image regions [Ma+94]. Scan line interleaving [TIH03] is a form of load-balancing, where individual disjoint scan lines are assigned to different processes to distribute active pixels and balance workload. Note that this approach has the drawback that image scan lines must be rearranged for transmission and compositing. The so-called *SLIC* algorithm [Sto+03] is based on direct-send and uses the encoding of active pixels, scan line interleaving and scheduling of operations, where spans of compositing tasks are assigned to processes in an interleaved way.

An extension of binary-swap compositing to nonpower-of-two process counts was introduced by Yu et al. in an algorithm called *2-3-swap* [YWM08]. The goal is to combine the flexibility of direct-send with the scalability of binary-swap compositing. The 2-3 swap algorithm is based on the fact, that any natural number greater than one can be expressed as a sum of twos and threes. This observation is then used to assign the processes to the communication groups of the first round. By clever assignment of image subtiles, Yu et al. proved that, in each round, group sizes can be between two and five, where each round can have multiple different group sizes present within the same round. For p processes, the total number of rounds is

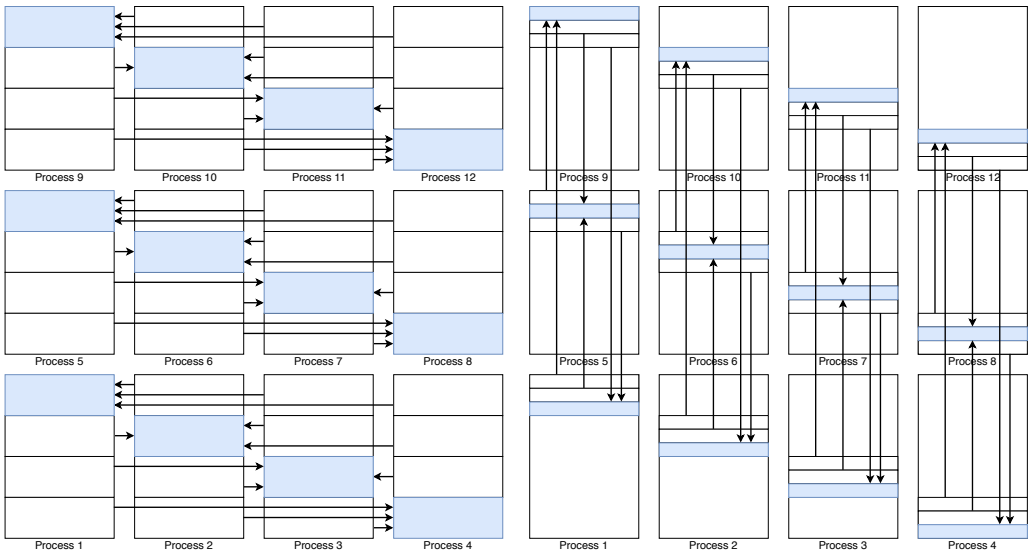


(a) Round 1 **(b)** Round 2

Fig. 2.9: 2-3-swap compositing using seven processes. In the first round (left), groups of twos and threes execute direct send compositing. In the second round (right), each process owns 1/7 of image, where the group size in the second round is between two and five.

equal to the base of $log(p)$. Figure 2.9 illustrates an example of 2-3-swap for seven processes.

The *radix-k* algorithm [Pet+09] improves on 2-3-swap by allowing more combinations of rounds and group sizes. Let k_i represent the communication group size in round i, where within each group direct-send is performed. Then, the k-values for all rounds can be expressed as a vector $\mathbf{k} = [k_1, k_2, ..., k_r]$, where r is the number of rounds. Let p denote the number of processes, then using above notation, direct-send can be expressed as $r = 1$ and $\mathbf{k} = [p]$. Similarly, binary-swap is defined as $r = log(p)$ and $\mathbf{k} = [2, 2, 2, ...]$. While 2-3-swap allows k-values between two and three, radix-k allows any factorization of $p = \prod_{i=1}^{r} k_i$, where all groups in round i are of equal size k_i. Figure 2.10 illustrates the radix-k scheme for twelve processes and $\mathbf{k} = [4, 3]$. Other possible factorizations for this example include $[12]$, $[6, 2]$, $[2, 6]$, $[3, 4]$ or $[2, 2, 3]$. The flexibility of radix-k allows to achieve higher compositing rates by selecting a factorization best suited for the underlying communication hardware, which might offer support for multiple simultaneous links or overlapping communication with compute. A number of optimizations to radix-k have been proposed, including active-pixel encoding and compression [Ken+10], scan line interleaving and improved compositing using telescoping for nonpower-of-two processor counts [Mor+11].



(a) Round 1 **(b)** Round 2

Fig. 2.10: Radix-k compositing using twelve processes over two rounds with $k = [4, 3]$.

In this thesis, several contributions with respect to accelerated compositing are presented. The novel task-based definition for distributed volume rendering (Chapter 5) provides a flexibly tunable task granularity by subdividing in both image and data space, thus yielding a hybrid scheme between sort-first and sort-last compositing. Based on an asynchronous binary tree compositing scheme including optimizations such as empty space skipping and early ray termination, the technique enables good scalability in combination with inherent dynamic load balancing. Additionally, a hardware-accelerated multi-tile streaming system is presented (Chapter 6) as a promising approach to make the distributed rendering capabilities of the GPUs within a remote supercomputer directly accessible to visualization systems. In this context, the feasibility and advantageousness of performing direct-send compositing on the client and displaying tiles at very high frame rates is demonstrated. This enables a system design in which render nodes directly send to the end-user client. With vastly improved hardware-accelerated compression and decompression based on progressive video codecs, even a large number of render nodes do not overwhelm the recipient of their images.

2.4 Topology-Based Data Analysis

Visualization of large-scale simulation output has to rely on a number of different strategies to facilitate meaningful analysis in reasonable time frames. Multi-

resolution schemes represent data on varying scales of resolutions and have a long standing tradition in this setting. They enable an essential compromise between fidelity and accuracy of visualization results on one hand, and computation and I/O bandwidth expended on the other. Among the large set of available methods, topological techniques such as the so-called *contour tree* stand out because they are able to provide meaningful simplification for scalar fields.

Topology-based data analysis and visualization use concepts and methods from mathematical and computational topology, such as homeomorphism, homotopy, connectivity, quotient spaces and cell complexes. By focusing on spatiotemporal relations between data subsets treated as functional units, an abstract and less redundant structural overview of the data can be provided. This overview can not only be used as a direct visual representation, but also for guided filtering and mapping in the visualization process. While topological methods are mathematically sound and typically feature robust algorithmic solutions, they require the joint work of domain and visualization scientists to define topological models that to not destroy the relevant structure in the data.

A particularly successful construct in the area of topology-based data analysis and visualization is the contour tree, which is used in this thesis as a tool for semi-automatic simplification in the context of in-situ data compression. This section will give a brief introduction to the concept of contour trees. A comprehensive overview of the state of the art in topology-based visualization and analysis is presented in Section 4.2.

Given a real-valued function $f : \Omega \rightarrow \mathbb{R}$ over a domain Ω, the so-called *level set* defines for each value in the function's range the set of points where the function value equals the given value v, i.e., the preimage of v with respect to f: $f^{-1}(v)$. Commonly known from topographic elevation maps, level sets can be depicted via contour maps, i.e., curves representing equal function value. Note that such a contour map typically only shows the curves for a few select values, while there is actually an infinite number of level sets resulting from an infinite number of values in the function's range. Level sets are not necessarily connected, and a connected component of a level set is formally called a contour. A key observation for visualization purposes is that contours can potentially serve as feature boundaries.

In order to reduce the infinite number of contours to a finite number of classes, a topological equivalence transformation is defined based on a continuous deformation of space that does not create new intersections or loops. Formally, this concept of properties of space that are invariant under transformation is called homeomorphism, and is based on the notion of Betti numbers from algebraic topology. Besides the

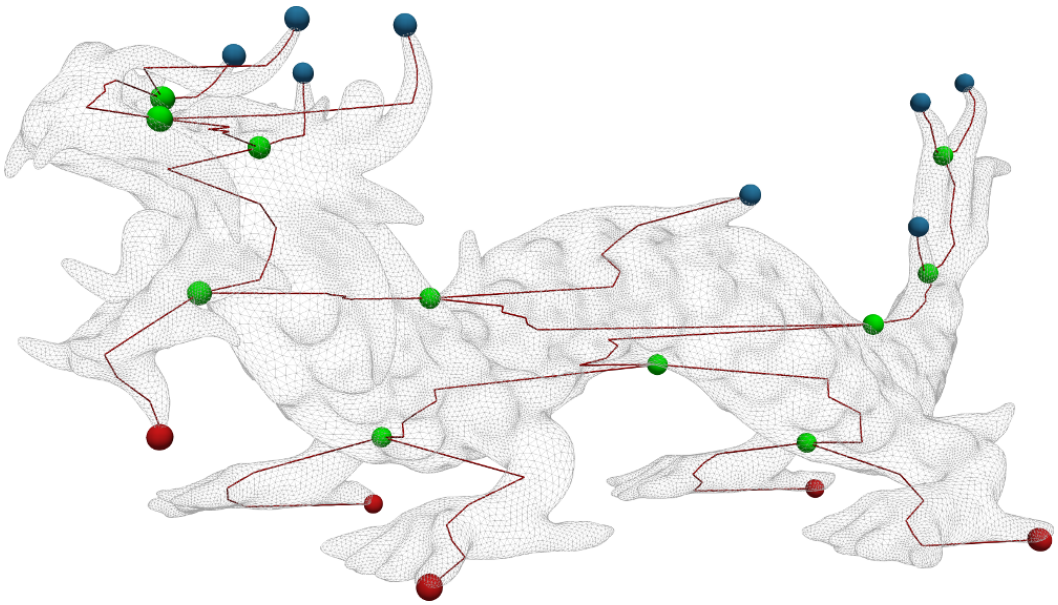


Fig. 2.11: Critical points and contour tree of a triangle mesh with height being interpreted as function value. The nodes of the contour tree are local minima (red), local maxima (blue) and saddles (green).

basic intuitive notion of connectivity Betti numbers also comprise higher-order connectivity in the form of loops and voids.

The evolution of level sets is reflected in so-called *Reeb graphs*, which are typically tought with the help of an imaginary flooded terrain. Assuming the function to be represented by a landscape, where each point's height is equal to its function value, the initial condition is a sufficiently high water level such that the landscape is fully submerged. By slowly draining the water, an intuition can be developed about how level set connectivity changes with changing function value. New contours become visible at peaks in the landscape, which correspond to local maxima of the function. Correspondingly, contours cease to be at local minima. Points where contours merge or split are called saddles. All of the above points are called critical points of f, and have in common that the gradient of f vanishes: $\nabla f(\mathbf{x}) = 0$.

The Reeb graph encodes a compact representation of this process by contracting each contour to a point. Contour adjacency is translated to point adjacency in a process formally known as quotient space in topology, where two points are equivalent if they belong to the same connected component of a level set. If Ω, the domain of f, is simply-connected, i.e., can be contracted to a point without destroying loops, the Reeb graph contains no loops and is then called the contour tree.

In a computational setting, where the function is the result of some numeric simulation, the domain is typically discretized into cells. In computational topology, this

discretization can be modeled using cell complexes known from algebraic topology. A frequently used model are simplicial complexes, where cells are convex combinations of affinely independent points. Assuming the function is modeled as a piecewise linear function on a simplicial complex, an efficient and elegant algorithm to compute the contour tree was presented by Carr et al. [CSA00]. This is illustrated in Figure 2.11 showing the critical points and contour tree of a three-dimensional mesh, where the height is intepreted as function value.

Topological concepts such as contour trees are powerful tools for noise reduction and data simplification. The idea of topological persistence gives a measure of relevance to parts of the contour tree. Based on the mathematical guarantee that a slightly perturbed function with simpler structure exists, topological simplification is a process reducing the contour tree in size and complexity. However, topological persistence is not only valuable in noisy settings, but can also be used to differentiate between large-scale and small-scale features.

In this thesis, in situ topological analysis and simplification based on the so-called *branch decomposition* of the contour tree is combined with compact image-based data representation to address the widening gap between increased computational throughout and prohibitive I/O overhead (Chapter 4).

2.5 Video Compression

Video compression is the process of converting raw digital video material into a format which takes significantly less space when stored or transmitted over network. Ubiquitously used in the entertainment industry, video compression is an essential tool enabling modern technologies such as digital television, DVD and Blu-ray disc, video conferencing, and internet streaming. In this thesis, hardware-accelerated video compression is applied to real-time rendered content in order to enable distributed remote rendering interactively at large scale (Chapter 6).

Both H.264 [Wie+03] and H.265 [Sul+12], also known as Advanced Video Coding (AVC) and High Efficiency Video Coding (HEVC), are industry standard video codecs, defining a syntax for storing compressed video and methods to decode this syntax in order to produce a displayable video sequence. Notably, the specifications do not define the details of the actual encoding process, which is left to the manufacturer of a video encoder. However, typically the encoder is designed such that it mirrors the procedure of the decoding process.

Both H.264 and H.265 essentially share the same encoding pipeline, which is illustrated in Figure 2.12. Conceptually, it consists of a lossless compression scheme based on spatial and temporal prediction followed by a binary encoding, as well as an optional lossy transformation and quantization routine. The corresponding decoder carries out the complementary decoding, inverse transformation and reconstruction steps to reproduce the original input as faithfully as possible.

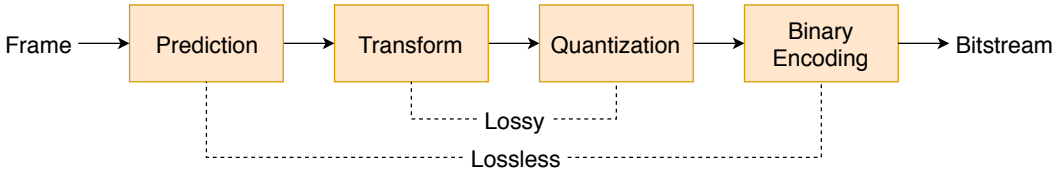
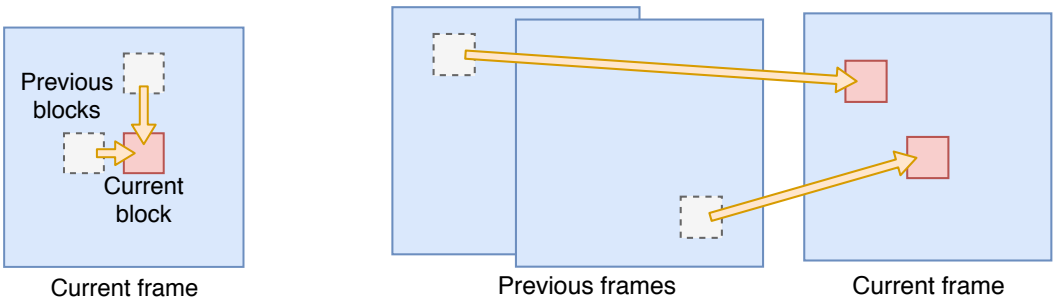


Fig. 2.12: Encoding pipeline for the H.264/H.265 video codec family consisting of both lossless and lossy compression steps.

In the prediction phase, video frames are processed in blocks of pixels, where for each block a prediction is formed based on previously-encoded data, either from within the current frame (intra prediction) or from other frames (inter prediction). This step leverages the redundancies between neighboring pixels, and determines what kind of compressed frame output is produced by the encoder. While so-called I-frames only perform intra prediction, P-frames and B-frames apply both intra and inter prediction. For P-frames, the search for redundancy is limited to prior frames in display order, whereas B-frames are able to use frames both before or after it. Note that while B-frames enable slightly better compression by providing a larger temporal search space for redundancy, they are unsuitable for low-latency streaming of interactively produced frames, which should be encoded and transmitted on the fly. The so-called *residual* is the difference between the current block and its prediction, and is preserved throughout the encoding pipeline, thus making the prediction phase lossless. In this context, the term *motion estimation* usually refers to the process of finding a suitable inter prediction within the given frame, whereas *motion compensation* denotes the subtraction of an inter prediction from the current block. Figure 2.13 illustrates the processes of both intra and inter prediction.

The H.264 standard processes frames in so-called *Macroblock* units of size 16x16 pixels, which can be further devided into smaller prediction blocks down to 4x4 pixels. H.265 improves on this by being able to adjust the block size into bigger or smaller blocks, called *coding tree units (CTU)*, ranging from 4x4 to 64x64 pixels. An improved CTU segmentation, as well as better motion compensation and spatial prediction are just a few of H.265's optimizations over H.264, which are out of the scope of this thesis.



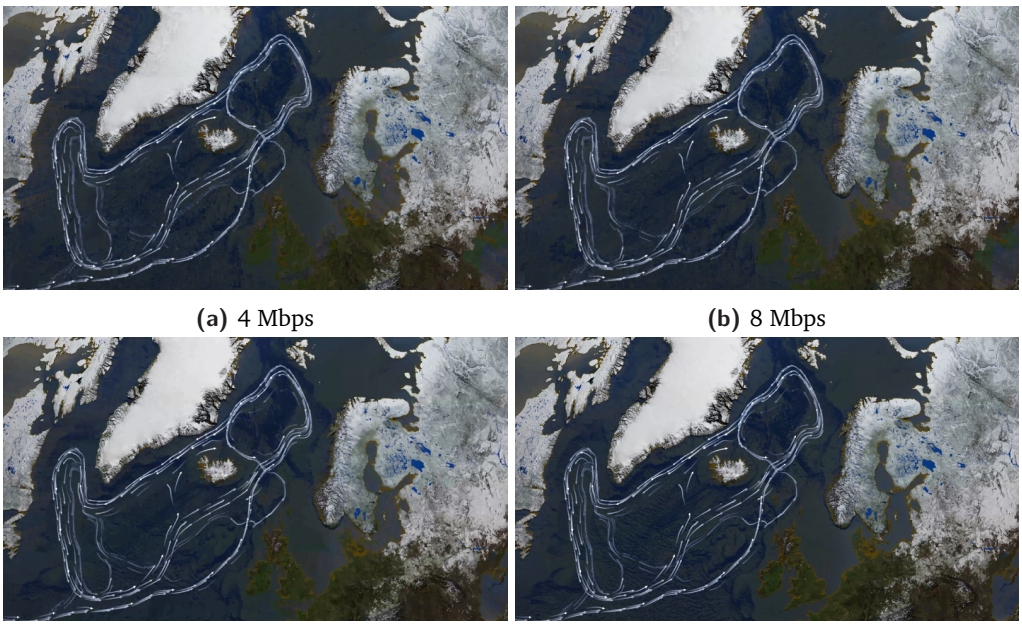
(a) Intra prediction **(b)** Inter prediction

Fig. 2.13: Prediction leverages the redundancies between spatially and temporally neighboring pixels. While intra prediction only considers previously encoded blocks from within the same frame, inter prediction also utilizes blocks from previously encoded frames.

The optional subsequent transform stage applies an approximate Discrete Cosine Transform (DCT) to a block of residual samples, producing a set of coefficients based on a standard basis pattern. Using these weighted basis patterns, the block of residual samples can be recreated. This transformation to frequency-space enables the prioritization of detail based on visual structure. The precision of the resulting transform coefficients is afterwards reduced by the quantization stage, where each coefficient is divided by an integer value. This process is guided by the so-called *quantization parameter (QP)*, an index used to derive a scaling matrix, where there is a logarithmic relationship between QP and the quantizer step size. The larger QP, the more coefficients are set to zero, providing larger opportunity for compression at the expense of decoded image quality.

While QP is essentially the primary control to trade quality for output size, in practice it is typically chosen automatically based on other user-defined metrics such as a specific constant target bitrate. Setting a fixed QP would result in a strongly varying bitrate depending on each scene's complexity, resulting in rather inefficient encodings. In contrast, for streaming scenarios as investigated in this thesis, it makes more sense to have encoders vary the QP automatically in order to meet a user-defined bitrate target. Figure 2.14 illustrates the impact of a given target bitrate on the reconstruction quality of an encoded scientific visualization rendering. Note the washed out look of the green landmass and the missing fine structures of the ocean floor at low bitrate. The impact of bitrate on encoding/decoding latency and streaming performance is further investigated in Chapter 6.

Ultimately, the produced quantized transform coefficients and associated parameters such as quantization range, prediction mode and other meta information on the structure of the compressed data must be encoded to form a single compressed



(a) 4 Mbps **(b)** 8 Mbps

(c) 16 Mbps **(d)** 32 Mbps

Fig. 2.14: Impact of bitrate on image reconstruction quality for different H.264 bitrates at 4K resolution based on a refresh rate of 90 Hz. Demo scenes from NASA's *Synthesis 4K* video, which was selected as a stand-in for high-resolution scientific visualization renderings. A low bitrate results in a washed out look with missing fine structures.

output bitstream. A lossless compression method such as variable-length coding or arithmetic codec is applied by the encoder. The resulting bitstream can be stored or streamed over network. For entertainment purposes, the raw video stream can be muxed into a container format such as MP4, where it is synchronized with other streams such as audio or subtitle tracks. In this thesis, the raw H.264 or H.265 streams produced by the rendering server are transmitted over the network and decoded at client-side for display.

3 Raytracing Particle-Based Fluid Surfaces

3.1 Motivation

The visualization of numerical simulation results based on mere point clouds is a challenging task. Such data sets emerge from Smoothed Particle Hydrodynamics (SPH) or Finite Pointset Method (FPM) simulations, two particle-based simulation techniques in the context of transient flow and continuum mechanical problems. In scenarios with free surfaces or moving geometry, classical grid-based numerical procedures, e.g., Finite Elements or Finite Volumes, fail due to their inherent necessity for remeshing.

However, currently there exist few well-elaborated standard visualization approaches tailored to grid-free methods. A multitude of contemporary rendering techniques are grid-based, and thus inappropriate for the evaluation and analysis of particle-based simulation results. Furthermore, for high density point clouds with great geometric complexity relative to the rastered image, it seems natural to stay within the context of point-based shape representation and directly use the surface points as display primitives.

In this context, numerous visualization approaches for particle-based surface reconstruction rely on scalar field visualization techniques. Using different kinds

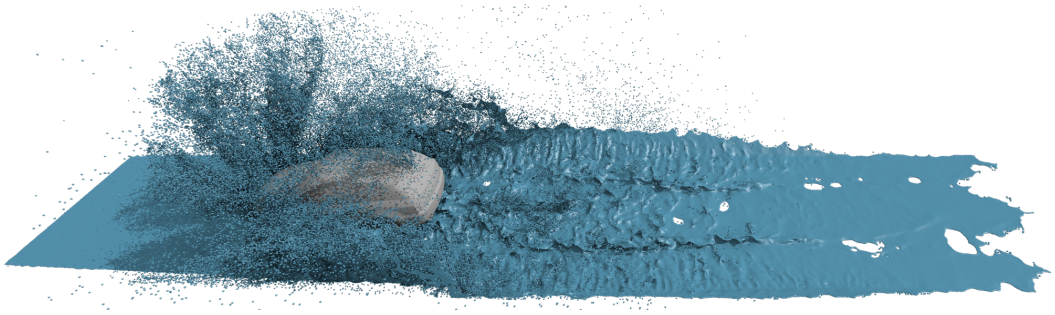


Fig. 3.1: Watercrossing simulation using 2.1 million particles. Rendered at 4K resolution with ambient occlusion and hard shadows in 2.4 seconds on 32 Xeon Phi KNL nodes of the Stampede2 supercomputer.

of overlapping basis functions, a scalar field representation of the fluid volume is computed, and subsequently used for direct volume rendering or isosurface extraction. A particularly interesting approach for SPH settings was presented by Yu and Turk [YT13], who have used each particle's neighborhood structure to compute anisotropic basis functions, thereby capturing local particle distributions more accurately and enabling smoother surfaces with distinct features. While the anisotropic surface definition itself is sound and promising, Yu and Turk have only used Marching Cubes for discrete triangulated surface extraction, thus preventing smooth visualizations which scale transparently with resolution. To achieve a pixel-accurate representation of the surface, the required resolution for the MC grid is prohibitively costly in terms of computational time for most relevant cases.

In this work, we build upon the rich anisotropic kernel approach, adapt and tune the surface definition to FPM-based fluid simulations, and present a novel direct ray tracing scheme for on-the-fly surface reconstruction. Specifically, after a brief review of relevant prior work (Section 3.2) and a compendiary introduction to the specifics of the FPM (Section 3.3), we make the following contributions:

- In Sections 3.4.1 and 3.4.2, we present an improved anisotropic kernel-based surface definition that specifically targets FPM simulations, incorporates automatic kernel scaling for variable smoothing lengths and intuitive visuals for isolated particles, and is easily parallelized.
- For this surface definition, we describe a novel direct ray tracing scheme definition (Section 3.4.3). This on-demand two-pass iterative sampling algorithm intelligently reduces intersection candidates for both opaque and transparent surface rendering, provides optimization opportunities for secondary rays, and allows the dynamic mapping of particle attribute values on to the surface using arbitrary transfer functions. Details of our implementation within the OSPRay raytracer are outlined in section 3.5.
- We conduct and analyze comprehensive benchmarks to quantify preprocessing and rendering times on different state-of-the-art hardware setups, including workstation, standard cluster and Xeon Phi accelerator systems, and demonstrate the applicability of our approach to a variety of medium and large scale FPM data sets (Section 3.6).

3.2 State of the Art

The reconstruction, tracking and visualization of fluid surfaces has been an object of research since the advance of fluid simulation and computational fluid dynamics. In the context of mesh-based simulation, various techniques for surface extraction have been developed such as level-set methods [OF03], particle level-set methods [ELF05], semi-Lagrangian contouring [Bar+06], volume of fluid methods [HN81] and explicit surface tracking [MÖ9].

However, especially for transient flow and continuum mechanical problems, particle-based simulation techniques such as SPH or FPM are more versatile than their grid-based counterparts. Different approaches have been investigated to reconstruct fluid surfaces directly from their point-based representations, such as splatting in combination with image-space curvature flow reduction [LGS09], collecting contributing particles using cylindrical rays [SJ00], globally fitting smooth interpolants based on radial basis functions [TO02], or point-set surfaces based on local moving least squares fits [Ale+01], which can also be used for adaptive advancing front surface triangulation [SFS05]. While point-set surfaces work well for densely sampled surface representations, e.g., from laser scans, they fail in turbulent and noisy scenarios with increasing counts of isolated particles. High quality volume rendering of particle data has been studied by Fraedrich et al. [FAW10] and Hochstetter et al. [HOK16]. Goswami et al. [Gos+10] present a voxel-based rendering pipeline on the GPU which constructs a partial distance field for subsequent ray casting. Reichl et al. [Rei+14] use binary voxel hashing to accelerate ray casting of point-based fluids on the GPU.

We follow the well elaborated approach of defining the fluid surface as an isosurface of a scalar field constructed from overlapping basis functions. The use of simple isotropic basis functions dates back to Blinn's metaballs [Bli82], which typically result in blobby surfaces. Zhu and Bridson [ZB05] extend this idea to compensate for local particle density variations to create considerably smoother surfaces. Adams et al. [Ada+07] track the particle-to-surface distance over time to create smooth surfaces for both fixed-radius and adaptively sized particles. Müeller et al. [MCG03] introduced the idea of creating a normalized scalar field based on the density as estimated by SPH, which we also follow. Solenthaler et al. [SSP07] propose a surface reconstruction technique based on considering the movement of the center of mass to reduce rendering errors in concave regions. Premžoe et al. [Pre+03] use isotropic kernels with interpolation weights stretched along the velocity field.

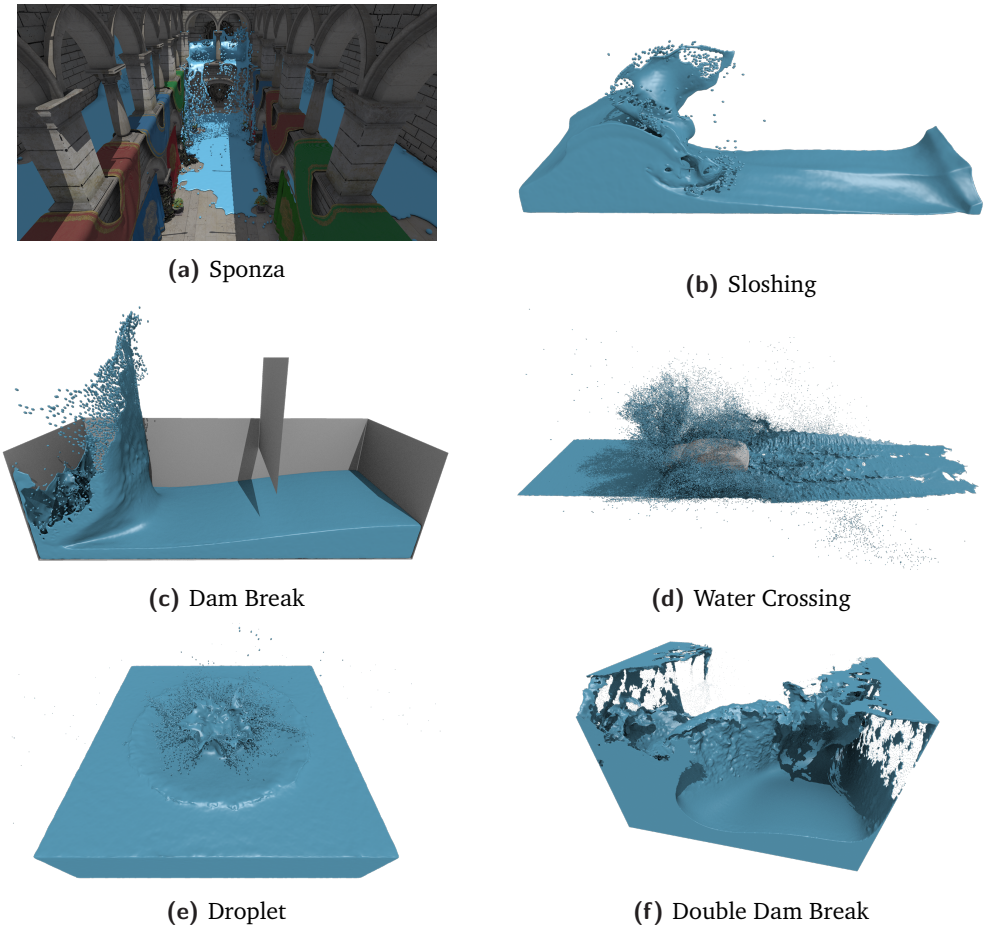
Owen et al. [Owe+98] inspired the use of anisotropic smoothing kernels, which were later combined by Ding et al. [DTS01] with variational implicit surfaces. Kalaiah and Varshney [KV03] have applied principal component analysis (PCA) to extract anisotropy from point clouds for point-based modeling, whereas Liu et al. [LLL06] have used anisotropic smoothing kernels for material deformation accuracy. Yu and Turk [YT13] have built upon these previous works and extracted a surface from the resulting normalized scalar field using the marching cubes algorithm [LC87]. Ando et al. [ATT12] employed this to determine and visualize thin fluid sheets in fluid simulations, while Macklin and Müller [MM13] combined it with the splatting approach of van der Laan et al. [LGS09] for their iterative density solver to achieve real-time fluid simulation. Akinci et al. [Aki+12] parallelized marching cubes-based surface extraction by considering only grid nodes in a narrow band around the surface. Yu et al. [Yu+12] used the anisotropic kernel method to construct an initial explicit triangle mesh, which is advected over time to track the air/fluid interface.

We improve on Yu and Turk's surface definition by presenting a novel direct raytracing scheme for anisotropic smoothing kernels in combination with a modified preprocessing procedure for FPM simulations. We also incorporate a variant of velocity-based stretching of isolated particles, which was shown by Bhatacharya et al. [BGB11] for level-set surface approximation minimizing thin-plate energy.

3.3 Finite Pointset Method

All simulations for this work have been performed using MESHFREE, a CFD software developed by Fraunhofer ITWM. MESHFREE uses the *Finite Pointset Method* (FPM, [Hie+05; Tiw+07]) to solve the Navier-Stokes equations on a point cloud. As the equations are solved in the Lagrangian formulation, particles move with the current local velocity in every timestep. In contrast to SPH techniques [LL03], particles are only numerical points and carry no mass, allowing to continuously adapt the point cloud by filling and deleting points. Furthermore, this also permits local refinement of the point cloud in areas of interest. Based on a generalized finite difference scheme, the FPM in contrast to SPH supports physical boundary and initial conditions, as well as many well known material models like Darcy, Johnson-Cook, and Drucker-Prager.

Typical real-world simulations discretize using less than 500 000 particles, as there is always a trade-off between computation time, accuracy and available resources. Lower particle counts are preferable as the time step size decreases with a finer



(a) Sponza **(b)** Sloshing **(c)** Dam Break **(d)** Water Crossing **(e)** Droplet **(f)** Double Dam Break

Fig. 3.2: Benchmark scenes rendered with ambient occlusion and shadows.

resolution due to numerical requirements. For this work we pushed the number of particles far beyond this to prove that our visualization method will be future-proof.

The visualization requires, per particle, information about position, velocity, smoothing length and kind of boundary. The smoothing length h controls the density of the point cloud; a distance of about $0.4 \cdot h$ among particles is ideal. The velocity information is used to deform isolated particles according to their direction of movement. Kind of boundary assigns each particle their type of boundary, which is divided into inner points, isolated points, wall points and free surface points. Isolated points are determined by the number of their neighbors within their smoothing length, wall points are permanently assigned by the FPM and the detection of free surface points is based on a local Delaunay tetrahedralization. Our visualization treats both wall and free surface points in the same way as boundary particles.

3.4 Surface Reconstruction

The proposed surface reconstruction pipeline consists of two steps: preprocessing and rendering. The former includes several computationally intensive steps such as smoothing of particle positions or computing anisotropy information based on local neighborhood structures. The resulting anisotropic kernel representation is used in the subsequent surface visualization via direct raytracing.

In Section 3.4.1 we will briefly recapitulate the mathematical foundation of the surface definition based on anisotropic kernels, with concrete algorithmic details and optimizations being presented in the following Sections 3.4.2 and 3.4.3.

3.4.1 Surface Definition

Our surface definition is based on the approach proposed by Yu and Turk [YT13], where a scalar field is constructed by overlapping anisotropic smoothing kernels representing the neighborhood structure of each particle. In contrast to previous isotropic approaches, these anisotropic kernels capture local particle distributions more accurately, enabling smooth surfaces, thin streams and sharp features in the reconstruction. We follow the original notation and mark contributed adaptions, optimizations and extensions accordingly.

The surface is defined as an isovalue of the normalized scalar field

$$\phi(\mathbf{x}) = \sum_i \frac{1}{\rho_i} W(\mathbf{x} - \bar{\mathbf{x}}_i, \mathbf{G}_i), \tag{3.1}$$

where ρ_i is the sum of the weighted contributions of nearby particles

$$\rho_i = \sum_j W(\bar{\mathbf{x}}_i - \bar{\mathbf{x}}_j, \mathbf{G}_j) \tag{3.2}$$

and $\mathbf{W}$ is an anisotropic smoothing kernel of the form

$$W(\mathbf{r}, \mathbf{G}) = det(\mathbf{G}) P(\|\mathbf{G}\mathbf{r}\|). \tag{3.3}$$

In the preceding equations, $\bar{\mathbf{x}}_\mathbf{i}$ is a smoothed particle position, $\mathbf{G}_i$ is a $3x3$ linear transformation matrix and P is a symmetric decaying spline with finite support. The linear transformation $\mathbf{G}$ rotates and stretches the radial vector $\mathbf{r}$ to normalized

isotropic kernel space, making $W(\mathbf{r}, \mathbf{G})$ an anisotropic kernel with iso-surfaces of ellipsoidal form.

The scalar field $\phi(\mathbf{x})$ is designed as a normalized density field smoothing out the scalar value of 1 at each particle's position over a continuous domain. Thus, an isosurface of $\phi(\mathbf{x})$ gives a surface representation encompassing the particles. We use a surface treshold of 0.2 for all of our use cases.

Note that the original surface definition by Yu and Turk [YT13] was designed for a SPH context and additionally included the mass of each particle. However, since FPM works with massless particles as transient nodes for computation these terms are not required in our definition.

3.4.2 Preprocessing

For each simulation frame, a dedicated preprocessing step is performed to compute the necessary data for the interactive surface renderer such as per-particle anisotropy information. The complete preprocessing procedure consists of several operations which will be discussed in the following.

Build search structure. We use hash grids for fixed-radius nearest neighbor searches, which are required in several pipeline steps. In this first step, we construct a hash grid over the complete set of fluid particles. Since nearest neighbor searches are the computationally most expensive operation during preprocessing, we try to cache and reuse previous search results as often as possible. Note that since FPM does allow a variable smoothing length (in contrast to SPH), we use the average smoothing length for the bucket size of the hash grid. Thus, the grid implementation needs to support search radii potentially larger than the grid size.

Determine thick boundary. In contrast to Yu and Turk [YT13], we do not use the complete particle set for kernel-based surface evaluation, but consider only particles in a given vicinity of the surface boundary, which we call the *thick* boundary. For this we use the MESHFREE-provided classification of free surface particles. However, if not already available, one could alternatively classify boundary particles based on their neighborhood count. For each boundary particle, we mark all particles within a radius r_b as thick boundary particles. r_b should be chosen as small as possible to reduce the number of candidate ellipsoids which are traversed during sampling, but at the same time large enough such that the surface is sufficiently represented and inner spheres do not intersect the outer surface. To this end, we empirically choose $r_b = 0.8 \cdot h_i$, where h_i is the smoothing length of the i-th particle.

The classification into thick boundary and inner particles is crucial to the subsequent preprocessing and the visualization. While inner particles are directly used by the renderer as a means for fast inner fluid traversal, only thick boundary particles are processed in the remaining preprocessing pipeline. After classification, particle data is regrouped such that inner and thick boundary particles reside in consecutive ranges, and search structure indexing is updated accordingly.

Update search structure. After data restructuring, the neighborhood hash grid is rebuilt to reflect the new particle indexing. In addition to the full particle set hash grid, we also construct a smaller additional hash grid only containing the thick boundary particles to speed up the subsequent computation of connected components.

Compute connected components. In order to alleviate attraction effects between approaching fluid components, a connected component analysis is performed on all particles in the thick boundary. Two particles i and j are defined as being connected if $\|\mathbf{x}_i - \mathbf{x}_j\| \leq r_{cc}$, where $r_{cc} = 0.45 \cdot h_i$. We identify this value since the dynamic point management algorithm of MESHFREE typically results in particles with distance $0.4 \cdot h_i$ to each other. We use a straightforward union-find algorithm to compute connected components and obtain a component id for each thick boundary particle. We only need to consider thick boundary particles, since components which are connected through inner particles necessarily are connected through boundary particles. Thus, when checking for connected component equality, inner particles are always considered valid.

Smooth particle positions. To improve the visual quality of flat surfaces, a single iteration of Laplacian smoothing is applied to the thick boundary particle positions. The updated particle positions $\bar{\mathbf{x}}_\mathbf{i}$ are computed via

$$\bar{\mathbf{x}}_\mathbf{i} = (1 - \lambda)\mathbf{x}_\mathbf{i} + \lambda \sum_j w_{ij}\mathbf{x}_j / \sum_j w_{ij}, \tag{3.4}$$

where $\lambda \in [0, 1]$ is constant describing the degree of smoothing and w_{ij} is a weighting function with finite support. Note that the updated smoothed particle positions are only used by the visualization pipeline and do not affect the underlying simulation. We typically use a value of 0.9 for λ to apply a strong smoothing effect. However, we find that conglomerates of isolated particles tend to be smoothed into a single particle, which can lead to visual artifacts. Thus, we set λ to 0.1 for particles with less than 20 neighbors.

The particle smoothing adheres to the previously established connected component labeling, i.e., for each particle only neighbors which belong to the same connected

component as the particle itself are considered. This is reflected in the following weighting function

$$w_{ij} = \begin{cases} 1 - ((\|\mathbf{x}_i - \mathbf{x}_j\|)/r_s)^3 & \text{if } \|\mathbf{x}_i - \mathbf{x}_j\| < r_s \text{ and } c_i = c_j \\ 0 & \text{otherwise} \end{cases}, \tag{3.5}$$

where c_i denotes the connected component of the i-th particle and r_s is the search radius used for nearest neighbor searches during particle smoothing; we use $r_s = r_b + r_{cc} = 1.25 \cdot h_i$, since inner particles are always considered part of each connected component. Thus, when a particle approaches a surface belonging to another connected component, the inner particles behind that respective thick boundary are not considered for smoothing until the two connected components are close enough to merge.

We cache neighborhood information determined during particle smoothing in order to reuse them in the subsequent computation of anisotropic smoothing kernels and normalization densities. Note that this results in an approximation since particle positions have been smoothed after neighborhood querying. However, our experiments show that reusing unsmoothed neighborhood information as an approximation has only negligible visual impact.

Determine kernel scaling factors. This step is a preliminary optimization for the subsequent computation of the actual anisotropic kernels, where a scaling factor is used to keep the volume of W (Equation 3.3) approximately constant for all particles with full neighborhood. Yu and Turk [YT13] have used an empirically chosen constant for all particles, which is dependent on the data set at hand and furthermore only makes sense for SPH which has a constant smoothing length. In contrast to this, we employ an automatic randomized sampling strategy to derive a polynomial relationship between local smoothing length and an optimal kernel scaling factor for a given particle.

To achieve this, we pick a random subset of inner particles and perform for each particle a simplified variant of the anisotropic kernel computation (the exact formulae will be outlined in the subsequent paragraph). For each particle, a covariance matrix C is constructed based on a local neighborhood query. Since inner particles are expected to have a full isotropic neighborhood, a convenient scaling factor for the particle at hand is computed based on the determinant of the covariance matrix as $k_s^i = \sqrt[3]{det^{-1}(C)}$.

All resulting (h_i, k_s^i) pairs are collected and averaged into a fixed number of buckets (20 in our experiments), and a least squares polynomial fit of degree 4 is computed,

which we denote by $k_s(h)$. This polynomial relationship is used in the following anisotropic kernel computation to pick a suitable kernel scaling factor for each particle based on its respective local smoothing length.

Compute anisotropic kernels. The foundation of the anisotropic kernel method is the determination of an anisotropy matrix $\mathbf{G}_i$ for each particle that describes the particle density distribution around it. By applying weighted Principal Component Analysis (WPCA) to the neighborhood of each particle in the thick boundary, the weighted covariance matrix $\mathbf{C}_i$ is constructed as

$$\mathbf{C}_i = \sum_j w_{ij}(\mathbf{x}_j - \mathbf{x}_i^w)(\mathbf{x}_j - \mathbf{x}_i^w)^T / \sum_j w_{ij}, \tag{3.6}$$

where $\mathbf{x}_i^w$ is the weighted mean defined as

$$\mathbf{x}_i^w = \sum_j w_{ij}\mathbf{x}_j / \sum_j w_{ij}. \tag{3.7}$$

Note that the weight function w_{ij} (Equation 3.5) now uses the updated particle positions originating from the preceeding smoothing operation and still considers connected components. We reuse the previously cached neighborhood information from the smoothing step, but only consider neighbors within the search radius $r_s = h_i$.

After the weighted covariance matrix has been constructed, a Singular Value Decomposition (SVD) is applied, yielding the principal vectors of deformation in the particle set considered. Assuming the SVD is denoted by $\mathbf{C}_i = \mathbf{R}\Sigma\mathbf{R}^T$, where $\mathbf{R}$ is a rotation matrix with principal axes as column vectors and $\Sigma = diag(\sigma_1, \sigma_2, \sigma_3)$ a diagonal matrix with eigenvalues $\sigma_1 \geq \sigma_2 \geq \sigma_3$, extreme deformations are prevented by restricting the proportions between largest and smaller eigenvalues, i.e., $\tilde{\sigma}_{2,3} = max(\sigma_{2,3}, \sigma_1/k_r)$, where k_r is a scaling factor denoting the maximum ratio between largest and smallest axis of the resulting ellipsoid. For our experiments we set $k_r = 4$.

Since the desired transformation matrix $\mathbf{G}_i$ is an inversion of the modified covariance matrix, its computation can be expressed as

$$\mathbf{G}_i = \frac{1}{h_i}\,\mathbf{R}\tilde{\Sigma}^{-1}\mathbf{R}^T, \tag{3.8}$$

where

$$\tilde{\Sigma}^{-1} = \frac{1}{k_s(h_i)}\,diag(\frac{1}{\sigma_1}, \frac{1}{\tilde{\sigma}_2}, \frac{1}{\tilde{\sigma}_3}). \tag{3.9}$$

In contrast to Yu and Turk [YT13], we do not represent isolated particles with insufficient neighbors as simple spherical kernels, but also transform those along each particle's velocity vector $\mathbf{v}_i$ as given by the simulation. From our experiments this approach leads to much more intuitive visuals for fast moving isolated particles such as splashing water droplets. Specifically, if a particle has less than 20 neighbors, the normalized velocity $v_n = \|\mathbf{v}_i\|/h_i$ is used to determine the major transformation strength as $m_a = 1 + d \cdot min(v_n, v_{max})/v_{max}$, where v_{max} is the maximum normalized velocity and d is the maximum degree of deformation. For our experiments we set $v_{max} = 50$ and $d = 0.3$. In order to preserve volume during transformation, the corresponding orthogonal minor transformation strength equals to $m_b = \sqrt{1/m_a}$. The final transformation matrix $\mathbf{G}_i$ is then

$$\mathbf{G}_i = \frac{1}{h_i}\, \mathbf{R}(\mathbf{e}_x, \mathbf{v}_i)\, \mathbf{S}^{-1}, \tag{3.10}$$

where $\mathbf{R}(\mathbf{e}_x, \mathbf{v}_i)$ is a matrix that rotates the x-axis $\mathbf{e}_x$ onto $\mathbf{v}_i$, $\mathbf{S} = k_n\, diag(m_a, m_b, m_b)$ is a scaling matrix and k_n is a size factor for isolated particles. We use $k_n = 0.35$ to prevent isolated particles from looking too bold.

Compute ellipsoid bounding boxes. Once all anisotropic transformations $\mathbf{G}_i$ have been computed, tight axis-aligned bounding boxes are constructed for each ellipsoid of influence as these are needed by the BVH acceleration structure used in the renderer for fast intersection candidate retrieval.

Using the standard derivation for tight bounding boxes around ellipsoids using projective geometry, the desired axis-aligned bounds can be computed as

$$\begin{aligned} x &= p_x \pm \sqrt{(\mathbf{G}^{-1}_{i,11})^2 + (\mathbf{G}^{-1}_{i,12})^2 + (\mathbf{G}^{-1}_{i,13})^2} \\ y &= p_y \pm \sqrt{(\mathbf{G}^{-1}_{i,21})^2 + (\mathbf{G}^{-1}_{i,22})^2 + (\mathbf{G}^{-1}_{i,23})^2}, \\ z &= p_z \pm \sqrt{(\mathbf{G}^{-1}_{i,31})^2 + (\mathbf{G}^{-1}_{i,32})^2 + (\mathbf{G}^{-1}_{i,33})^2} \end{aligned} \tag{3.11}$$

where $\mathbf{p}_i = (p_x, p_y, p_z)$ is the particle's center.

Compute weighted contributions per particle. As last preprocessing step, the sum ρ_i of weighted contributions of nearby particles as outlined in Equation 3.2 is computed for each particle based on the previously constructed anisotropic kernels. For the symmetric decaying spline P in Equation 3.3 we make use of the reversed *smootherstep* function defined as $P(x) = 1 - (6x^5 - 15x^4 + 10x^3)$ for $x \in [0, 1]$. Also, we precompute the combined coefficient value of $det(\mathbf{G}_i)/\rho_i$ for each particle which is needed in the renderer for surface sampling as illustrated in Equation 3.1.

After preprocessing has finished, only the data relevant for our direct ray-based rendering technique is kept in memory. For each particle in the thick boundary this boils down to position $\mathbf{p}_i$, transformation matrix $\mathbf{G}_i$, bounding box, coefficient $det(\mathbf{G}_i)/\rho_i$ and optionally a user-selected attribute value which is used for color mapping onto the surface in conjunction with a given transfer function. For inner particles only the position $\mathbf{p}_i$ and a radius $r_i = s \cdot h_i$ is stored, where s is a scaling factor that should be chosen sufficiently large such that the resulting inner spheres overlap completely with themselves and the thick boundary, i.e., there are no holes, however at the same time as small as possible to improve acceleration structure efficiency during traversal. We choose $s = 0.5$ in our approach.

3.4.3 Intersection

Contrary to previous work based on isosurface extraction via marching cubes, we perform a direct raycasting of the scalar field formed from the preprocessed anisotropic (ellipsoidal) smoothing kernels.

In order to determine ray-surface intersection position, it is necessary to sample and test the scalar field along the ray. Since the field is defined at any point as the sum of contributing kernel values, multiple overlapping kernels may be needed to reach the surface value. On the other hand, intersecting a single arbitrary kernel does not guarantee that the surface is hit. Figure 3.3 shows a typical scenario, where a fluid volume defined by multiple overlapping anisotropic kernels form in the thick boundary is partially occluded by an isolated particle in front. Additionally, we use spherical inner particles to overlap the complete fluid volume in order to reduce the number of sampling locations and perform fast traversal of inside segments.

While a ray may encounter an isolated particle as a candidates for intersection, the surface threshold is not surpassed during sampling. The first actual surface hit is encountered at the surface of the fluid bulk. If the surface is completely opaque, then rendering for this particular ray stops here. However, e.g., in the case of transparent rendering, a new ray may be started an epsilon behind the former hit point, which continues sampling through the set of contributing anisotropic kernels. It is crucial to keep the number of required samples to a minimum and efficiently skip ray intervals which are completely inside the fluid volume.

To collect contributing kernels for each ray, an all-hit intersection test is performed, computing entrance and exit positions for all candidate anisotropic kernels along the ray. Candidates are provided by the underlying acceleration structure based on their

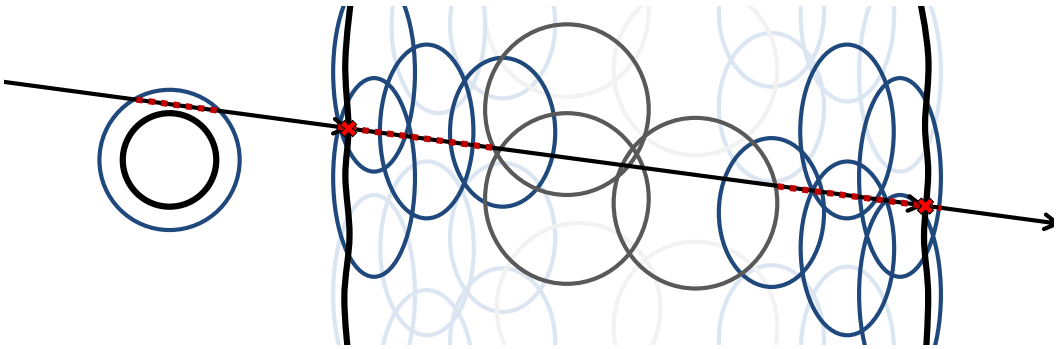
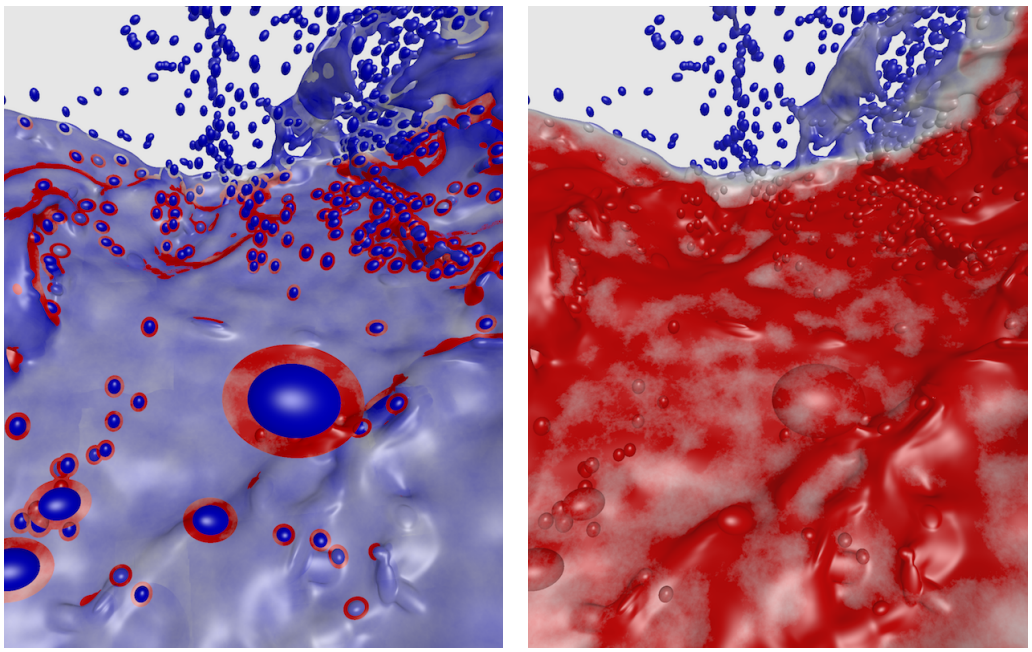


Fig. 3.3: Intersection scheme: anisotropic smoothing kernels (blue), inner spheres (gray), sampling locations (red dots) and actual surface (black). The smoothing kernel of a candidate isolated particle in front is intersected and sampled, but the surface threshold is not reached until the surface of the fluid bulk is intersected. In transparent rendering, a secondary ray is started, which utilizes the inner spheres to reduce the number of required samples until the exit intersection is reached.

axis-aligned bounding boxes. The resulting events are inserted in an ordered list, which in our case has proven to be faster than saving them unordered and sorting them afterwards, as studied in [Ams+15]. If no kernels were hit, there can be no surface intersection.

However, gathering all intersected ellipsoids along the complete ray can be quite costly and is often not even necessary, e.g., for opaque surfaces, since the essential contributing kernels are located in close vicinity to the frontmost ellipsoid hit. Motivated by this observation, we perform a two-pass approach for opaque surface rendering, where in a first step only kernels close to the first hit are gathered and checked for intersection. Since typical acceleration structures do not guarantee strict sorted ordering of query results for performance reasons, we guide the acceleration structure to converge to the frontmost candidates as fast as possible.

To achieve this, we introduced an optimization we call *offset culling*: whenever a candidate ellipsoid is evaluated, the end of the ray is clamped to the respective entry intersection point plus a predefined offset. The offset must be chosen large enough, such that in any case all kernels contributing to the surface are returned by the search structure, even when the first candidate encountered is the frontmost ellipsoid. We use $0.5 \cdot h_{avg}$, where h_{avg} denotes the average smoothing length of the whole data set. Offset culling yields a significant performance improvement over a naïve all-hit query, since the number of candidate kernels potentially contributing to the surface is dramatically reduced, as can be seen in Figure 3.4. This leads to another improvement as we do not need to sort ellipsoids during gathering, but rather simply store them in any order. If the surface was not hit during the first pass, the ray is cast again without offset culling to gather all kernels as described



(a) Offset culling **(b)** Naïve all-hit

Fig. 3.4: Total number of ellipsoids collected per ray, from zero (blue) to 200 (red). Using offset culling dramatically reduces the number of smoothing kernels considered for surface sampling. Red halos in Figure 3.4a indicate areas where the frontmost kernels are hit, but the surface threshold is not reached and offset culling terminates. Thus, an all-hit intersection is performed to reach the real surface intersection.

above. Offset culling is only performed for opaque surfaces, where rays always start outside of the volume and stop at the first surface intersection. When rendering with transparency, large numbers of rays start after the first surface intersection and pass through the inner fluid, where next surface interaction does not lie in vicinity of the frontmost ellipsoids encountered. Listing 3.1 shows the handling of each candidate ellipsoid during the all-hit phase.

```
1  compute intersections of ray with ellipsoid
2  if (no intersection)
3     return
4  if (offset culling)
5     if (ellipsoid behind end of ray)
6        return
7     set end of ray to entry + offset
8     store ellipsoid in unsorted array
9  else
10    construct events for entry and exit
```

```
11      insert events in sorted array
```

Listing 3.1: Per-ellipsoid callback for all-hit intersection.

In addition to the anisotropic kernels for particles in the thick boundary, we make use of the remaining inner particles of spherical form for fast traversal of inner ray segments. Each sphere along the ray marks an interval that is always inside the volume. Therefore, there can be no intersection with the surface during this interval and it is safe to skip sampling on this segment of the ray. Similiar to the gathering of candidate ellipsoids, we perform an all-hit intersection in the local spheres scene. To reduce the number of intervals to be checked during sampling, intervals are merged with overlapping ones in a sorted list of intervals as they are detected, as outlined in Listing 3.2.

```
1  compute intersections of ray with sphere
2  if (no intersection)
3     return
4  check existing intervals for overlap
5  if (no overlap)
6     insert new interval in sorted list
7  else
8     merge overlapping intervals with new interval
```

Listing 3.2: Per-sphere callback for all-hit intersection.

Having collected all potentially contributing anisotropic smoothing kernels and having constructed the minimal list of inner intervals, uniform sampling is performed along the ray between all recorded events. For our experiments we employ $0.1 \cdot h_{avg}$ as sampling step size. At each sampling position, we first check if an inside segment can be skipped. Then, the list of currently contributing anisotropic kernels is updated. If no ellipsoids are actually contributing at the sample position, we jump to the next ellipsoid's entry event. Otherwise, the surface's scalar field value is evaluated over the sum of contributing kernels. If the computed value passes the defined surface value, the surface was hit between the current and the last sample position. We then perform recursive sub-sampling using value-weighted bisection to refine the intersection point on the given interval. With only a few iterations this approximates the surface position up to single floating point precision in our cases. The surface normal is interpolated from the weighted contributions of the individual kernel normals. The complete surface intersection scheme is outlined in Listing 3.3.

```
// 1st pass: offset culling
if (offsetCulling)
   all-hit intersect ellipsoids (with culling)
   if (no intersection)
      return
   sample from first hit to offset:
      sum up scalar field over contributing kernels
      if (surface value passed)
         determine exact hit (value-weighted bisection)
         interpolate normal
         if (color mapping)
            interpolate attribute
         return
      go to next sampling position

// 2nd pass: full traversal
all-hit intersect ellipsoids (no culling)
if (no intersection)
   return
all-hit intersect spheres (construct inner intervals)
sample from first event to last event:
   if (sample point inside inner interval)
      jump to end of interval
   update list of contributing ellipsoids
   if (no contributing ellipsoids)
      jump to next event
      continue
   sum up scalar field over contributing kernels
   if (surface value passed)
      determine exact hit (value-weighted bisection)
      interpolate normal
      if (color mapping)
        interpolate attribute
      return
   go to next sampling position
```

Listing 3.3: Two-pass surface intersection scheme.

For secondary rays such as shadow rays or ambient occlusion there is no need for precise hit point sampling. In order to accelerate occlusion tests, we also perform a two-pass approach here. First, only the inner spheres are intersected for occlusion, terminating upon any sphere hit. If there is no occlusion from spheres, the actual surface intersection algorithm is performed as outlined above in a simplified form. In this case, we do not perform recursive subsampling, and we do not compute interpolated normals and attributes.

3.5 Implementation

We extensively use local OpenMP parallelization in conjunction with MPI distribution across nodes throughout our preprocessing pipeline, in which nearest neighbor searches are the dominant computational effort. Our experiments have shown that in general dynamic scheduling for local parallelization is superior to static scheduling due to the slightly imbalanced workload depending on each particle's neighborhood. For the multi-threaded construction of the hash grids we use the concurrent *libcuckoo* hash map [Li+14] as back-end. We also investigated search structures based on kD-trees, which exhibited slightly slower performance than the hash grid-based approach for our use cases.

Since the input data sets are relatively small in size even for practically large point clouds, we begin with the complete particle set on each node. All operations on the thick boundary are distributed across nodes, i.e., each node performs processing on only a subset of the particles. After each preproccesing step, state is exchanged through MPI messages. The partial thick boundary markings are all-reduced using a logical or-operator. Partial connected component labelings are exchanged and merged to a global labeling on each node using the same union-find approach as locally. Smoothed particle positions and anisotropic kernels are simply all-gathered on each node. Since the kernel scaling factors are based on random sampling with low performance impact, the scaling curve is computed on a single node only and broadcasted to the others. Eventually, at the end of preprocessing, the partially computed densities are gathered at the master node which then has the fully preprocessed data and sets up the visualization.

The intersection algorithm was implemented as a user geometry in the OSPRay framework [Wal+17]. Internally, the all-hit kernels to collect thick boundary ellipsoids and inner spheres make use of two additional manual Embree scenes, which can be queried for intersection and occlusion independently and are based on the all-hit kernel studies in [Ams+15]. The complete visualization code is automatically vectorized by the ISPC compiler, which compiles a C-based SPMD programming language to run on the SIMD units of CPUs and the Intel Xeon Phi architecture, without the need for tedious writing of manual SSE/AVX intrinsics.

We use OSPRay's scientific visualization renderer and for distributed rendering the MPI offloading device, which replicates the scene model on all worker nodes and performs sort-first compositing at the master node.

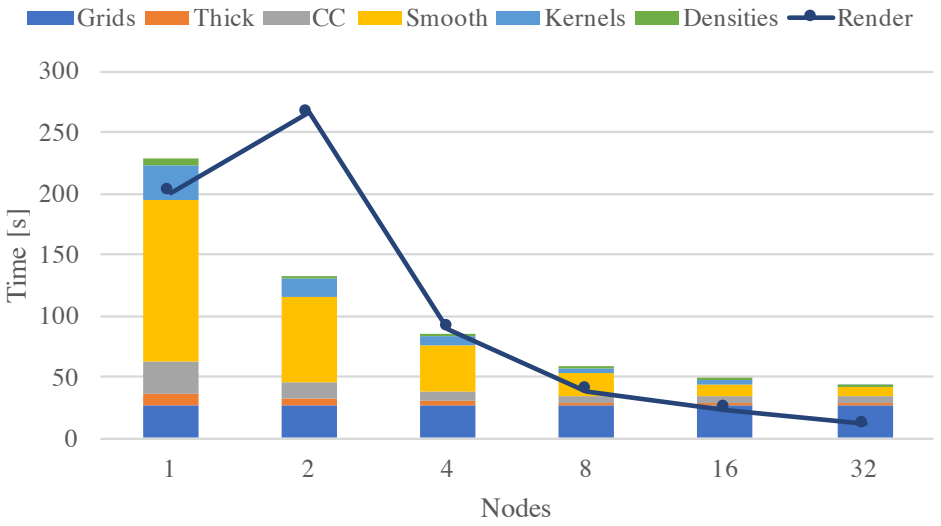
3.6 Results

To investigate the characteristics and potential of our direct raytracing scheme, we conduct comprehensive benchmarks with respect to data set size, scene complexity, visual fidelity and hardware setup. We consider three hardware setups: a desktop workstation with an Intel i7-6700K CPU (4x 4.0 GHz), the *Elwetritsch* cluster using one Intel Xeon E5-2640 v3 (8x 2.6 Ghz) per node with InfiniBand QDR interconnect, and the *Stampede2* supercomputer providing Intel Xeon Phi 7250 Knights Landing accelerator cards (68x 1.4 GHz, 272 hardware threads) with Intel Omni-Path interconnect. We choose three base cases for rendering: opaque rendering, opaque rendering with ambient occlusion (16 samples per hit) and hard shadows, and transparent rendering. All renderings have been performed at standard 4K resolution (3840x2160), except for the low resolution desktop scenario which was performed at 960x540 to represent a downsampled rendering mode for interactive use cases.

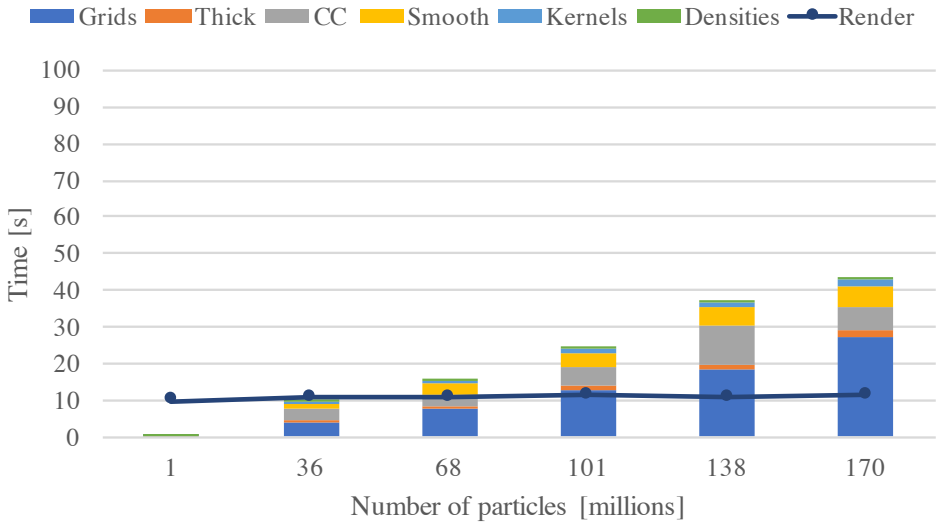
Figure 3.2 shows the different test scenes for our benchmarks, covering a wide spectrum of particle counts and rendered with ambient occlusion and hard shadows. As expected from the anisotropic kernels method, surfaces are smooth while preserving sharp features such as thin shields and isolated particles. The visualization can be considered faithful to the simulation, even if this exposes the dynamic particle management in the form of occasionally popping particles across time steps.

Complete timings for preprocessing and rendering are presented in Table 3.1. Preprocessing scales roughly linearly with the number of thick boundary particles. Basic opaque rendering times can be considered interactive for reduced resolutions on single nodes, whereas distributed rendering enables interactive frame rates at full 4K resolution. Depending on scene complexity, we observe highly increased rendering workload for ambient occlusion, which is not due to our method per se but generally expected in raytracing contexts. From a hardware perspective, the Stampede2 accelerator cards outperform standard CPU-based machines in rendering due to ample vectorization opportunities in packed raytracing. On the other hand, preprocessing provides less room for vectorization, for instance during hash grid construction or connected component analysis. Distributed rendering requires enough workload in relation to communication overhead in order to scale reasonably well.

To better understand our method's capability to handle large data, we perform an extensive scaling study using the Double Dam Break data set, which was iteratively refined from 1 million particles up to 170 million particles. Thus, we can observe approximately the same scene complexity at different resolutions. Figures 3.5 and 3.6 show two fundamentally different scaling studies on Elwetritsch and Stampede2,



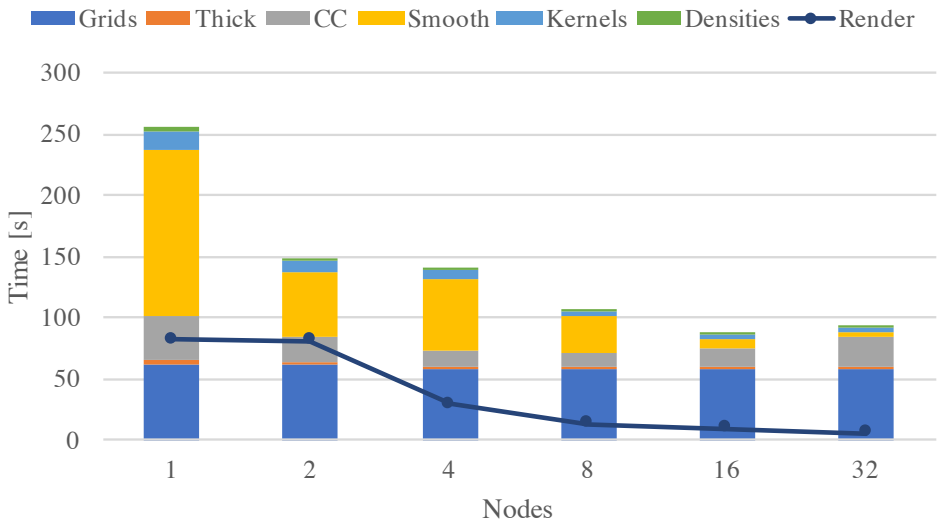
(a) Elwetrisch: 170 million particles on 1 - 32 nodes.



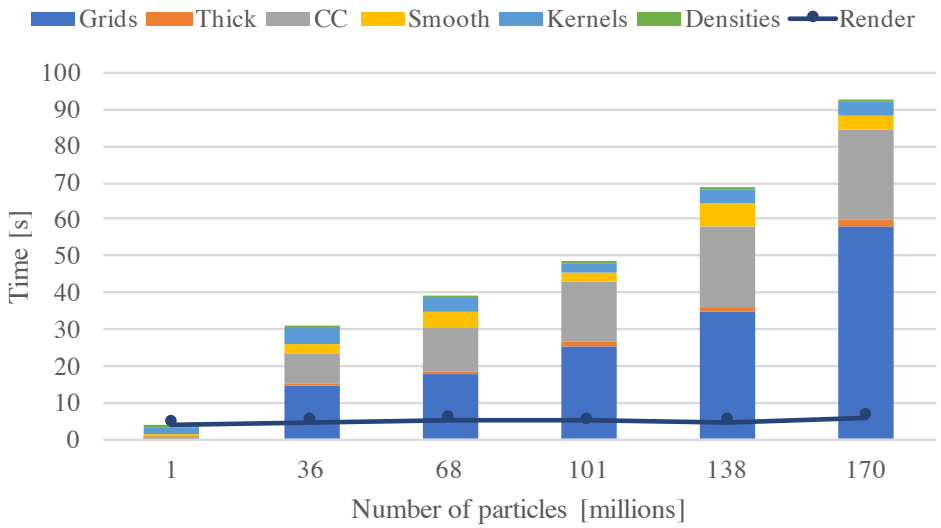
(b) Elwetrisch: 1 - 170 million particles on 32 nodes.

Fig. 3.5: Scaling on Elwetritsch using the Double Dam Break data set, rendered at 4K resolution with ambient occlusion and shadows.

respectively. Figures 3.5a and 3.6a show the highest resolution data set preprocessed and rendered on varying node counts, whereas Figures 3.5b and 3.6b present timings for different particle counts on 32 nodes of each cluster. In all cases, rendering was performed with ambient occlusion and hard shadows to provide enough opportunity for workload distribution during rendering. We highlight that while our core aim was not perfect scalability but rather a feasibility study, our method also performs well in these computationally involved scenarios.



(a) Stampede2: 170 million particles on 1 - 32 nodes.



(b) Stampede2: 1 - 170 million particles on 32 nodes.

Fig. 3.6: Scaling on Stampede2 using the Double Dam Break data set, rendered at 4K resolution with ambient occlusion and shadows.

When considering the same full resolution data set on increasing node counts, it becomes obvious that hash grid construction is a purely local operation on each node which does not scale well. Connected component construction scales to some degree, but it still limited by the eventual merge of partial connected components on each node. The increase in rendering time for two nodes is explained by the way we use OSPRay's MPI offloading device, where the master rank has only organizational duties and performs sort-first compositing, while the remaining nodes

do the actual rendering. Thus, only one of two ranks is rendering, with additional communication overhead. In contrast to Elwetritsch, the overhead seems to be negligible on Stampede2.

While preprocessing times increase approximately linearly with particle count, rendering appears rather unaffected in these cases. This is expected, as the offset culling optimization in opaque rendering is based on the local smoothing length, resulting in approximately the same magnitude of anisotropic kernels contributing to the surface intersection for each ray. The same behavior can be observed on both Elwetritsch and Stampede2, while the former is in general the better preprocessor and the latter the faster renderer.

We conclude that our direct rendering technique is versatile and suitable for both high fidelity and interactive rendering scenarios. It scales reasonably well even using trivial parallelization, and is thus an option for in-situ use cases by easily enabling preprocessing and rendering on multiple nodes. An additional design aspect of our method is that it runs in image space rather than object space complexity, which is a desirable feature for large scale data applications and is common for raytracing based approaches.

3.7 Discussion

We presented a novel approach to the direct visualization of particle-based fluids and have demonstrated its applicability to a wide spectrum of FPM simulations. Our technique is based on an anisotropic kernel model, in which the neighborhood of each particle is used to construct a locally deformed smoothing kernel, allowing smoother surfaces with sharp features. In our raytracing-based rendering scheme we perform optimizations such as offset culling to effectively reduce the number of contributing kernels per surface intersection. We conducted comprehensive benchmarks to study the performance and scaling characteristics of our parallelized and distributed implementation on various hardware configurations and have demonstrated its general versatility.

We anticipate that many improvements to our approach are possible. Since FPM relies on local moving least squares interpolation, it by necessity incorporates nearest neighbor search structures with managed ghost particle information in distributed cluster mode, which could be reused in the preprocessing state of our pipeline. Furthermore, we would like to investigate using our system for production visualization, i.e., full path tracing, which is an important use case, e.g., in the automobile

industry. Finally, comparing performance of our implementation against a GPU implementation would be interesting to shed light on the relative strength of the differing architectures for our use case. We intend to investigate all these aspects in future work.

Data Set Particles (Thick) Preprocessed Size	Time	Desktop (4K)	Desktop (Low Res)	Elwetritsch (1 Node)	Elwetritsch (32 Nodes)	Stampede2 (1 Node)	Stampede2 (32 Nodes)
Sponza 161 000 (97%) 12 MB	Preprocess	1.0	1.0	1.2	0.3	4.3	2.2
	Opaque	1.6	0.1	1.9	0.1	0.4	0.6
	AO + Shadows	41.9	2.9	26.4	1.8	5.3	0.6
	Transparent	7.1	0.7	7.8	0.6	2.1	0.6
Sloshing 457 000 (49%) 20 MB	Preprocess	1.9	1.9	1.8	0.5	4.8	1.7
	Opaque	4.9	0.4	5.5	0.4	1.6	0.5
	AO + Shadows	56.2	3.7	56.0	4.3	32.9	2.8
	Transparent	50.5	4.8	51.4	3.3	17.9	1.6
Dam Break 586 000 (49%) 26 MB	Preprocess	2.0	2.0	2.0	0.5	4.8	2.3
	Opaque	3.1	0.3	3.6	0.2	0.9	0.5
	AO + Shadows	166.4	10.5	164.5	10.2	81.2	5.9
	Transparent	28.2	2.7	30.0	1.7	8.7	0.6
Water Crossing 2 153 000 (79%) 136 MB	Preprocess	10.3	10.3	11.9	1.9	11.4	7.1
	Opaque	3.5	0.3	3.8	0.2	1.3	0.6
	AO + Shadows	55.9	3.7	57.4	3.0	29.9	2.4
	Transparent	27.1	3.4	27.9	1.6	11.9	1.5
Droplet 3 765 000 (19%) 103 MB	Preprocess	5.2	5.2	5.0	1.1	6.8	4.6
	Opaque	6.0	0.6	6.7	0.4	1.8	0.7
	AO + Shadows	34.5	2.4	35.4	2.0	17.7	2.1
	Transparent	48.2	4.9	51.7	2.5	13.9	0.7
Double Dam Break 170 000 000 (14%) 4.1 GB	Preprocess	329.9	329.9	230.6	44.9	205.6	102.8
	Opaque	15.0	6.8	13.5	1.0	8.2	2.3
	AO + Shadows	202.9	20.1	199.6	11.3	80.7	5.8
	Transparent	150.2	24.8	147.2	9.4	77.7	14.4

Tab. 3.1: Preprocessing and rendering timings in seconds.

4 Contour Tree Depth Images

4.1 Motivation

At the current scale of computational capability provided by large-scale parallel computer architectures such as commodity clusters and modern supercomputers, high-fidelity computational simulation models have assumed a significant role in scientific research and engineering applications. However, this increased amount of computation has incurred architectural trade-offs. While arithmetic capacity and in-core memory have grown at a tremendous rate, I/O subsystems have not been able to keep abreast in relative bandwidth [Chi+10]. As a consequence, numerical data produced during typical simulations cannot be persistently stored, e.g. to hard drives, in its entirety; a lack of available I/O bandwidth would make this prohibitively costly with respect to time.

Visualization of large-scale simulation output thus has to rely on a number of different strategies to facilitate meaningful analysis in reasonable time frames. Multi-resolution schemes represent data on varying scales of resolutions and have a long standing tradition in this setting. They enable an essential compromise between fidelity and accuracy of visualization results on one hand, and computation and

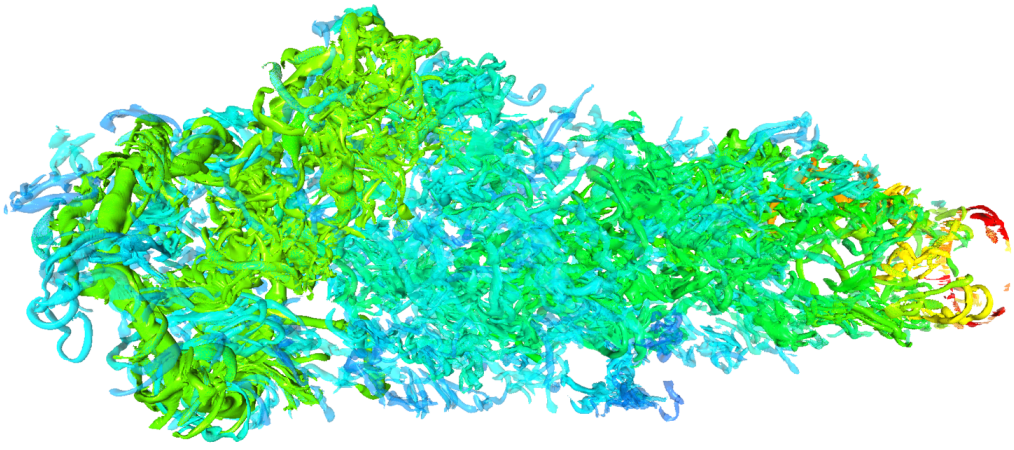


Fig. 4.1: Automatic segmentation of the turbulent jet5 λ_2 data set into 1024 branches based on branch persistence.

I/O bandwidth expended on the other. Among the large set of available methods, topological techniques such as the *contour tree* stand out because they are able to provide meaningful simplification for scalar fields.

A further smart approach to reduce the volume of data generated from simulations while preserving a significant amount of analysis flexibility was recently introduced by Ahrens et al. [Ahr+14]. The underlying idea is to generate in situ an image database that stores multiple layers of predefined visualization renderings. The layers can then be composited in post processing depending on specific demands by the scientist conducting analysis.

In this context, the intent of this work is to study the combination of in situ topological analysis with the image-based approach of Ahrens et al. Based on in situ contour tree analysis and simplification, we obtain a segmented representation of scalar fields contained in the simulation data at every time step. A rendering of this segmentation is then generated describing all components visible in every pixel (similar to an A-buffer), and stored together with the simplified contour tree. These ingredients can then be used in post-analysis to flexibly select specific subsets of the segmentation, after further simplification if required.

The overall intent of this work is to investigate possible advantages of such an approach for the visualization of large-scale data. Specifically, after a brief review of relevant prior work (Section 4.2), we make the following contributions:

- In Sections 4.3 through 4.7, we describe a system to combine in situ contour tree analysis, simplification, and image-based representation to facilitate reduced I/O requirements while preserving flexibility in visualization.
- We conduct several experiments to quantify the I/O savings possible from such an approach, and describe results and analysis in Section 4.8.
- We anticipate that many enhancements and improvements are possible, and discuss a number of such opportunities in Section 4.9.

Our contribution is intended as a baseline demonstration of the feasibility and potential of the combination of topological analysis and image-based representation in large-scale in situ scenarios.

4.2 State of the Art

A classic use case of topology in scalar field visualization is isosurface extraction, where typically several topological properties such as the number of connected components or the genus of the isosurface, i.e., the number of independent tunnels, are of central interest. Based on Morse theory, showing that topological changes in scalar fields defined on manifolds happen only at distinct critical points, Reeb graphs capture the topological evolution of individual contours using these critical points and their relationships. The efficient construction of Reeb graphs in general is still an active field of study [DN09]. However, for simply connected domains, the Reeb graph is always a tree structure [BR63], called *contour tree*, which is algorithmically computable for tetrahedral [CSA00] and hexahedral [PCM04] meshes. Since contour trees can become hard to understand due to high complexity, Weber et al. [WBP07] introduced *topological landscapes*, a visual metaphor for contour trees by creating a representative terrain with the same topological structure as a given contour tree, which can be further extended to reflect the geometric proximity of the features represented therein [Bek+12] or hierarchically used for topology exploration [Dem+12].

Topological techniques have proven highly valuable for the analysis, visualization and exploration of scientific data. Bajaj et al. [BPS97] introduced the *contour spectrum*, an interface providing the contour tree and additional properties such as area and enclosed volume alongside isosurface visualization. Fujishiro et al. detected significant isovalues automatically using the contour tree for transfer function design [Fuj+00]. Weber et al. based scalar field exploration on the detection of critical points and critical regions [WSH03]. Van Kreveld et al. [Kre+97] performed seeded isosurface extraction based on the contour tree, which was extended by Carr and Snoeyink to use the contour tree as a visual index for a volume data set and identify all contours for a given isovalue [CS03]. Takahashi et al. [TFT05] employed interval volumes to visualize regions of uniform topology, providing means to examine internal structures by peeling away top layers. Takashima et al. [Tak+05] further investigated the idea of peeling off layers by using topological information such as isosurface inclusion level in multi-dimensional transfer function design.

Noise in data sets can lead to large numbers of irrelevant critical points, complicating feature-driven exploration based on topology. Topological simplification eliminates insignificant features by *cancellation*, i.e., removing irrelevant pairs of critical points. The *Volume skeleton tree* [TTF04] and the *Morse-Smale complex* [Ede+03; Bre+04; Gyu+05] are two topological structures widely used for scalar topological simpli-

fication besides the contour tree. However, our work relies on the simplification approaches introduced by Carr et al. [CSP04] and Pascucci et al. [PCMS04]. Carr et al. [CSP04] used leaf pruning and node collapse operations for contour tree simplification. Removing a leaf and the arc incident to the leaf from the contour tree discards the corresponding contours from further consideration when using seed-based contour extraction. Pascucci et al. [PCMS04] introduced the *branch decomposition* of a contour tree, which can be interpreted as a hierarchy of contour tree simplifications. Since a branch is defined by a monotone path connecting a saddle and an extremum vertex, discarding a branch is equivalent to the topological cancellation of the respective two critical points. Our work is inspired by Weber et al. [Web+07], who used the branch decomposition for the segmentation of a volume into regions of equivalent contour topology and applying separate transfer functions to each region. Carr et al. [CSP10] also used the simplified contour tree as an interface for exploratory visualization.

Bremer et al. studied the application of topological methods to large data scenarios, defining feature identification by thresholding isosurfaces in terms of the Morse complex and representing the complete evolution of all features over time in tracking graphs [Bre+10]. Also, Bremer et al. used hierarchical merge trees as a compact feature representation reducing data storage [Bre+11]. Thompson et al. [Tho+11] introduced *hixels* as a new compact representation of large scalar data, storing a histogram of values for each sample point of the domain, thereby trading off data size and complexity for scalar value uncertainty. Landge et al. used segmented merge trees to encode a wide range of threshold based features to obtain a reduced data representation while maintaining post-processing flexibilities [Lan+14]. More recently, also image-based approaches to large data storage and visualization emerged. Tikhonova et al. [TCM10] presented the idea of using proxy images for interactive exploration without accessing the original 3D data. View changes, transfer function exploration, and relighting are handled in proxy image space only. Ahrens et al. [Ahr+14] generated an in situ image database, storing multiple layers of predefined visualization renderings, which can then be composited in post processing. The central idea, which is also key to our work, is to achieve a massive data reduction when storing the simulation output of large-scale numerical simulations, while preserving visualization fidelity and flexibility for future post-processing. Frey et al. presented an interesting novel scheme for progressive rendering which helps to achieve a fluent interactive visualization of large data at high frame rates [Fre+14].

Building on this rich methodological foundation, we investigate the feasibility and potential of combining in situ topological analysis and image-based representation

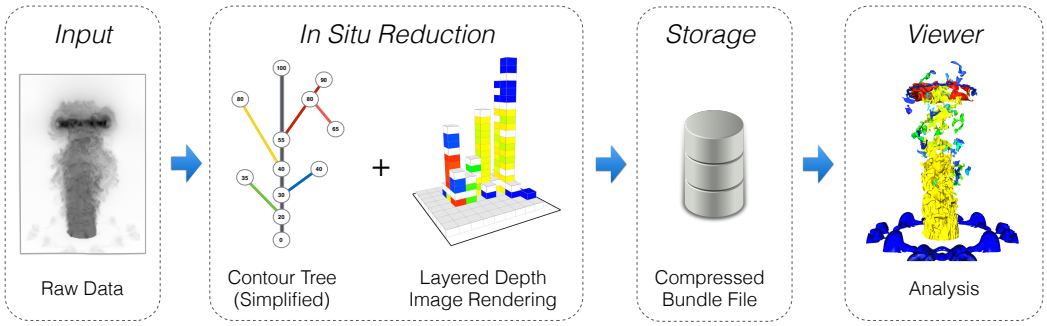


Fig. 4.2: Conceptual architectural overview of our contribution, providing a framework for the flexible exploration of in-situ generated compressed renderings. Volume data is segmented into topological regions based on its contour tree. After automatic filtration, the segments imposed by the simplified contour tree are intersected by a ray casting rendering approach. All resulting fragments form a layered depth image, which is compactly stored combined with the contour tree and of significantly smaller size than the original data.

to tackle the problems in permanent data storage of high-fidelity simulation results, caused by the modern architectural trade-offs in large-scale parallel computer systems.

4.3 Method Overview

We aim to achieve substantial reductions in data size by leveraging topological simplification and compact image-based storage. Our approach essentially consists of two components: an image *rendering library* and an interactive *viewer application*. A conceptual overview of the central steps in the processing workflow is depicted in Figure 4.2.

The rendering library is directly includable into simulation code and targets high-performance cluster environments or workstations. Simulation output data is segmented into topological regions based on its contour tree, which in turn are intersected by a ray casting rendering approach. However, instead of only determining the fragment closest to the camera, we store each intersection together with a set of local properties in a linked fragment list, similar to an A-buffer [Car84]. These properties include the element in the contour tree corresponding to the intersected segment and further additional attributes used for rendering such as the normal or ray depth at the intersection. The fragment lists combined with the contour tree are stored in a proprietary binary file format, which is considerably smaller in size

than the original simulation data, yet provides enough flexibility for subsequent data exploration.

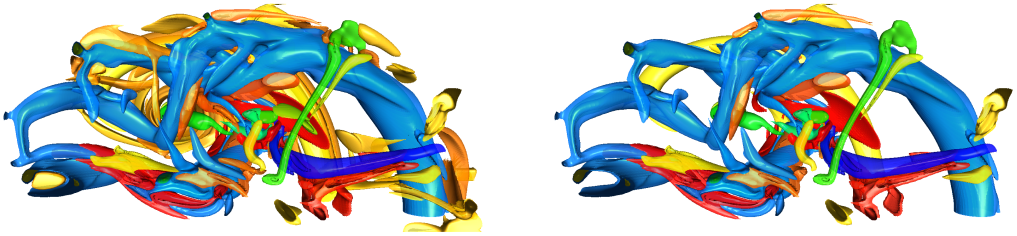
Once compressed layered depth image data has been generated, it represents a significant size reduction of the original input data and can be explored in the interactive viewer application running on the user's desktop computer. By interactively modifying visual properties of the regions imposed by the contour tree or applying further filtering schemes, the user can control which topological regions are displayed.

4.4 Segmentation and Filtering

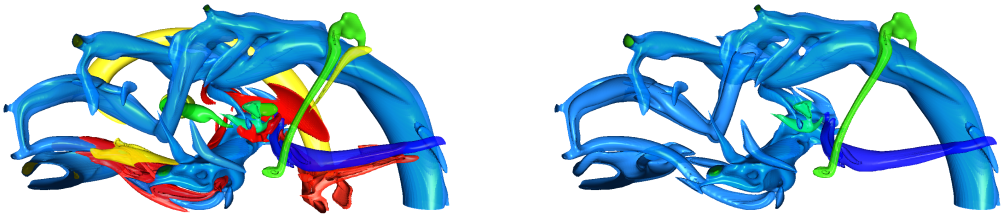
Given a scalar field defined on a regular grid as input data, we construct the contour tree using the sweep and merge algorithm by [CSA00], where the split and join sweep phases are executed in parallel. Afterwards, the so-called *branch decomposition* is computed using a variant of [PCMS04]. The branch decomposition can be interpreted as a hierarchy of contour tree simplifications. Since a branch is defined by a monotone path connecting a saddle and an extremum vertex, discarding a branch is equivalent to the topological cancellation of the respective two critical points. The output of the above algorithm is a tree structure representing the branch decomposition of the contour tree and a mapping of vertices to their corresponding branch in the decomposition. Notably, the latter is used for rendering only, whereas the branch decomposition is stored in the final image, with each branch being characterized by index, volume and critical value pair.

Filtering is a central concept inherent to the hierarchical branch decomposition structure. In our context, if a branch is to be discarded, all vertices belonging to this branch are reassigned to the closest unfiltered parent branch. We employ this technique in two ways. First, the initial branch decomposition used for rendering can be controlled by a user-provided maximum number of branches. This is a crucial step to achieve significant data reduction at in situ time, as the initial unfiltered branch decomposition consists of numerous branches which are irrelevant for the later visualization, either due to negligible importance or being caused by noise. Second, in the interactive viewer application, the user can apply several consecutive filtering steps in order to simplify the layered depth image visualization.

Besides manual branch selection, the library provides means to automatically identify branches of interest. Automatic branch selection can be either done by range, i.e., select all branches whose critical values intersect a given input range, or by sorting,



(a) 90 branches **(b)** 50 branches



(c) 30 branches **(d)** 10 branches

Fig. 4.3: Incremental simplification of the central turbulence in the *plate* λ_2 data set by sorted branch persistence, enabling a flexible and topologically-guided exploration of vortex core structures.

i.e., sort all branches either by persistence or volume and pick the first k branches which fulfill a given minimum persistence or volume threshold. Notably, if the sorting criterion of two branches is equal, they are sorted by their depth in the branch decomposition tree, ensuring child branches are discarded before their parent. Once expendable branches are identified, filtering is performed recursively. If a branch is flagged for discard, also all of its children are discarded. Otherwise they are individually checked. However, other filtering criteria are certainly possible.

The consequences of performing a filtering step depend on whether it is done for in situ data reduction or in the viewer application. Automatic filtering after the construction of the initial branch decomposition prior to rendering only requires an update to the vertex-to-branch mapping. However, when applying additional user-controlled filtering in the viewer application, the branch indices of all fragments in all depth images linked to this contour tree need to be updated. Furthermore, since branch filtering can geometrically lead to the merging of neighboring topological regions, replacing branch indices in the linked fragment lists can produce duplicate intersections which need to be eliminated. In this case, we only keep the one closest to the camera.

4.5 Depth Image Rendering

After the contour tree for a single simulation time step has been constructed, multiple layered depth image renderings can be computed based on its branch decomposition. Thus, given a scalar field defined on a regular grid, the vertex-to-branch mapping, camera position and orientation, resolution and an optional maximum number of depth layers, we perform a concurrent ray casting procedure for image generation.

4.5.1 Segment Intersection

The goal is to record all intersections of rays with the boundaries of the topological regions defined by the contour tree. Assuming a sequence of sample positions along the ray, we principally need to determine the branch index at each location and detect an intersection if the branch index at the current candidate location is different from the branch index at the previous accepted intersection. However, while we want to achieve high intersection precision, performing naïve uniform sampling with sufficiently small step size along the complete ray obviously suffers from bad performance in large homogeneous segments.

Rather, our sampling algorithm traverses data voxels following a three-dimensional Bresenham approach. For each cell encountered, we check if all eight corner vertices belong to the same branch, in which case we can trivially use this respective branch index and return the entering intersection of the ray with the voxel boundary as intersection candidate. Otherwise, there are potential intersections with topological segment boundaries within the cell, which we approximate by a local uniform sampling scheme restricted to the cell volume.

At each sample position, we need to determine the topological segment containing the given location, i.e., determine the respective branch index. We follow the approach presented in [Web+07], which is based on the relation that monotone paths in the scalar field always map to monotone paths in the corresponding contour tree and vice versa. First, the unique monotone path going through the given sample position and two cell corners is determined by exploiting the linearity of the trilinear interpolant within the cell along axis-oriented lines. We then follow the monotone path in the contour tree until the branch containing the data value at the sample position is found. Eventually, for each detected fragment, the associated branch index and additional optional parameters such as gradient of the scalar field or depth value are stored.

Notably, the central idea of this proof-of-concept work is independent of the employed data sampling scheme, providing opportunities for performance optimization. Also, together with an appropriate intersection detection scheme, our approach is easily applicable to non-regular data.

4.5.2 GPU Acceleration

We have implemented the above layered depth image rendering algorithm both as a CPU- and GPU-version. While the multi-threaded CPU version of our library targets rendering of large data sets in traditional cluster environments without accelerator cards, we have additionally implemented GPU hardware acceleration as a proof of concept of our highly parallelizable algorithm.

As described above, rendering by ray casting creates a list of fragments for each ray, representing the visual properties for all intersections with topological segments it encounters. Our implementation is conceptionally similar to per-pixel linked lists [Yan+10], a recent technique in computer graphics for hardware-accelerated order-independent transparency [Mau+12] based on the classic A-buffer approach [Car84]. For each pixel, a linked list of fragments keeps track of the intersections encountered during traversal. The respective fragment properties themselves are stored in a global shared fragment pool.

Based on OpenGL 4.3 [SA13], our implementation relies on shader storage buffer objects and image load/store extensions. Input volume data and the vertex-to-branch mapping originating from the branch decomposition of the contour tree are stored in uniform volume samplers. Per-pixel information, i.e., the head node of the linked fragment list and the number of nodes in the list, is stored in two uimage2D samplers in r32ui layout, sized at the requested rendering resolution. A shader storage buffer object holds the global fragment pool, where each fragment stores its properties and a link to the next fragment. An atomic counter buffer object is used to keep track of the next free fragment in the global fragment pool. Whenever a newly generated fragment is to be stored in the pool, the counter is incremented, the head node is exchanged and the number of fragments in the list is increased, using the respective atomic buffer and image operations. In order to maintain linked list consistency, the next node of the new fragment is set to the previous head node of the pixel's linked fragment list.

Fast parallel ray generation is achieved by first rendering only the back faces of the data volume cube to an offscreen framebuffer and storing the generated interpolated

fragment position, i.e., ray exit point of this pixel, in the respective framebuffer's color attachment. In the next pass, the same procedure is applied to the front faces only, yielding the ray starting positions.

In summary, our GPU-based rendering procedure consists of four phases, all of which are executed highly parallel by hardware:

1. Draw back faces of data cube to offscreen framebuffer to obtain ray exit locations.
2. Draw fullscreen quad clearing linked fragment lists, i.e., reset head nodes and node counts in per-pixel samplers.
3. Draw front faces of data cube to obtain ray entry locations.
4. Perform ray casting and store intersections in linked fragment lists.

Step 3 and 4 are executed in a single shader pass. After rendering, the contents of the global fragment pool and the per-pixel samplers are read back to host memory.

4.6 Storage

Reducing data storage requirements while maintaining flexibility for interactive exploration of results is central to our contribution. Thus, compactly storing the output of our rendering algorithm is crucial. For a given (filtered) contour tree, multiple depth renderings from different perspectives and resolutions can be generated. These bundles are stored in a single file in HDF5 format with zlib deflate compression enabled at level 6, which we found to be a good compromise between size and speed.

The branch decomposition of the contour tree is serialized recursively in depth-first manner, where for each branch we store index, saddle value, extremum value, and volume. However, with only a few kilobytes, the contribution of the branch decomposition to the final file size is negligible. Clearly, the majority of storage is consumed by the depth images, which thus require more optimization w.r.t. memory consumption. We need to store for each pixel a list of fragments, where a single fragment consists of its associated branch index and additional attributes such as normal. While during rendering fragments are stored and manipulated as linked lists distributed across memory and potentially shared by rendering threads, this data can be compactly reorganized for permanent storage. We remove the next-pointers of the linked lists by rearranging fragment lists as contiguous blocks in a single large

layout, where each pixel of the final layered depth image only stores the starting offset and the number of fragments.

Since the largest offset, the maximum number of fragments per pixel and the maximum branch index (i.e. total number of branches in the branch decomposition) are known, primitive data types used for storing the respective values can be intelligently chosen. As an example, if we can restrict the total number of branches to 256, this allows branch indices to be stored in single bytes, yet provides sufficient segmentation detail in many scenarios.

While storing the branch index of each fragment is mandatory, additional attributes are optional and depend on the intended visualization. In our studies, we additionally compactly store the normal at each intersection in two bytes using a spheremap transformation [Pra10]. Also, we store the intrinsic and extrensic camera parameters used for image generation, serving as reference for lighting and shading in the viewer.

4.7 Interactive Viewer

Compressed layered depth image bundle files can be loaded and interactively explored on the user's desktop machines in the interactive viewer application, the counterpart of the parallel rendering library in our framework.

The interface is split in two parts: the visualization of the selected depth image and a tree widget used for the display and modification of the branch decomposition tree, including color, persistence, value range and volume of each branch. Users can manually or automatically select multiple branches by range or sorted minimum persistence/volume criterion as described in section 4.4. Additionally, picking branches in the visualization using the mouse cursor is an efficient way of selecting regions of interest. Picking can either be restricted to automatic selection of the front-most fragments or further guided by presenting all intersected segments in a cross-section interface. Selected branches are visually highlighted by distinct coloring. Given a set of selected branches, the user can apply filtering in order to simplify the visualization. Filters can even be combined by applying them on top of previous filtering steps. Undo is supported by storing the history of filtering operations.

Depth images are interactively updated and displayed in real time, being powered by hardware-accelerated rendering similar to the techniques outlined in section

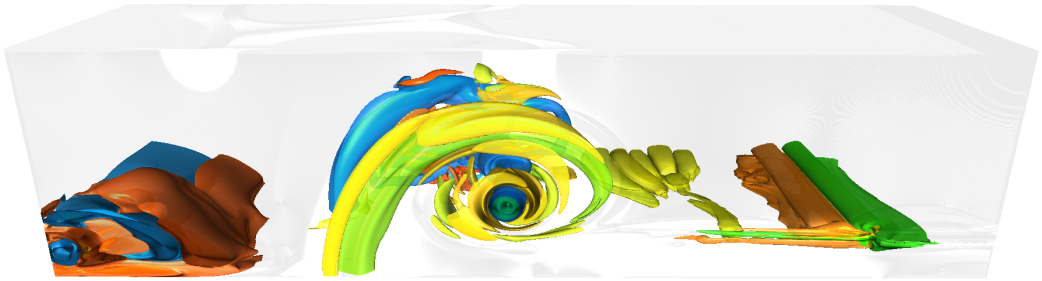


Fig. 4.4: Automatic segmentation of the *plate* λ_2 data set into 100 branches sorted by branch volume. An HSV color scale has been applied to all branches, where for each branch the average scalar branch value was approximated by the mean of saddle and extremum values stored with the branch.

4.5.2. Image data and branch properties as edited in the branch decomposition tree interface are transferred to shader storage buffer objects on the GPU and rendered to an offscreen framebuffer, which is afterwards displayed at flexible scale or exported to an image file. For each pixel, the fragments are back-to-front composited (they are already sorted by design), with shading being computed based on the stored fragment normal and branch coloring as defined for the respective branches. In addition to standard Phong illumination, we partially apply angle-based view-dependent transparency as presented in [Hum+10].

4.8 Results

We have tested our framework on two regular vector field data sets. The *jet5* data set (given on a 256×512×256 grid over 3000 timesteps) results from a direct numerical simulation of a jet of high-velocity fluid entering a medium at rest and exhibits progressively finer vortical structures in the velocity field. Similarly, the *plate* data set (with a resolution of 1024×256×256 over 285 timesteps) describes the mixing of fluid flowing past a plate at different speeds that undergo mixing due to viscous effects. Both data sets have been converted to regular scalar fields based on velocity, vorticity magnitude or the λ_2 vortex detection criterion [JH95]. Full examplary renderings of the data sets are depicted in Figures 4.7 and 4.4, respectively. An HSV color scale has been applied to all branches based on λ_2, where for each branch the average scalar branch value was approximated by the mean of saddle and extremum values stored with the branch.

All images depicted in this chapter have been rendered at 1920x1080 resolution on a standard desktop workstation using an Intel Core i7-4770k quad-core CPU and a

NVIDIA GeForce GTX 770 GPU. The complete topological segmentation of a single time step took less than one minute using the above hardware for both data sets. Rendering times varied depending on the chosen perspective, but typically ranged from 20 to 60 seconds. The final compression and storage of the resulting depth images was performed in less than one second.

4.8.1 Compression

The key goal of our contribution is to provide a flexible trade-off between storage memory consumption and interactive data exploration, essentially controllable by the branch decomposition simplification level used for layered depth image rendering.

Figure 4.5 depicts the output bundle file sizes of our technique applied to the *jet5* data set in relation to several user-provided maximum branch numbers, using the same camera perspective as in Figure 4.1. For each number of branches, the graph shows

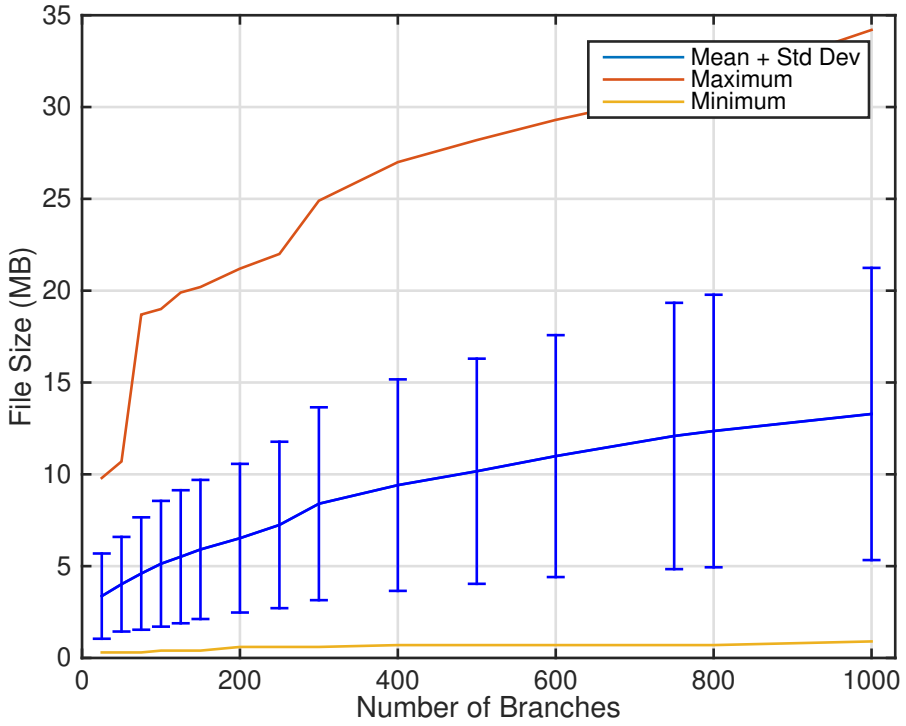


Fig. 4.5: Impact of limiting the maximum number of branches in the automatic simplification of the *jet5* λ_2 data set on the compressed layered depth image file size, using the same camera perspective as in Figure 4.1. Depth image file size can vary greatly depending on the complexity of the scene captured in the rendering, however in general stays significantly smaller than the input data size of 134 MB.

the mean, standard deviation, minimum and maximum file sizes measured across the complete time range of the data set. Clearly, the mean file size is monotonously increasing with the number of branches. Notably, there is a small jump visible at 256 branches, when fragments are required to use shorts instead of bytes for branch index storage. However, in our studies, a maximum number of 256 branches has emerged as a very good compromise between file size and segmentation detail, providing a mean output size of approximately 7 MB, compared to the input data size of 134 MB for each scalar *jet5* time step, i.e., a size reduction of 95%.

The size reduction is even more prominent for full-view renderings of the larger *plate* data set as in Figure 4.4. Figure 4.6 illustrates the raw output size and compressed output size of our algorithm applied to each time step in relation to the constant input data size. In contrast to the great variation in file size due to the continuously increasing complexity of the *jet5* data set over time, the graph reflects the rather uniform complexity of the *plate* data set. Also, the graphs clearly highlight the fundamental reduction in data size compared to the input data even

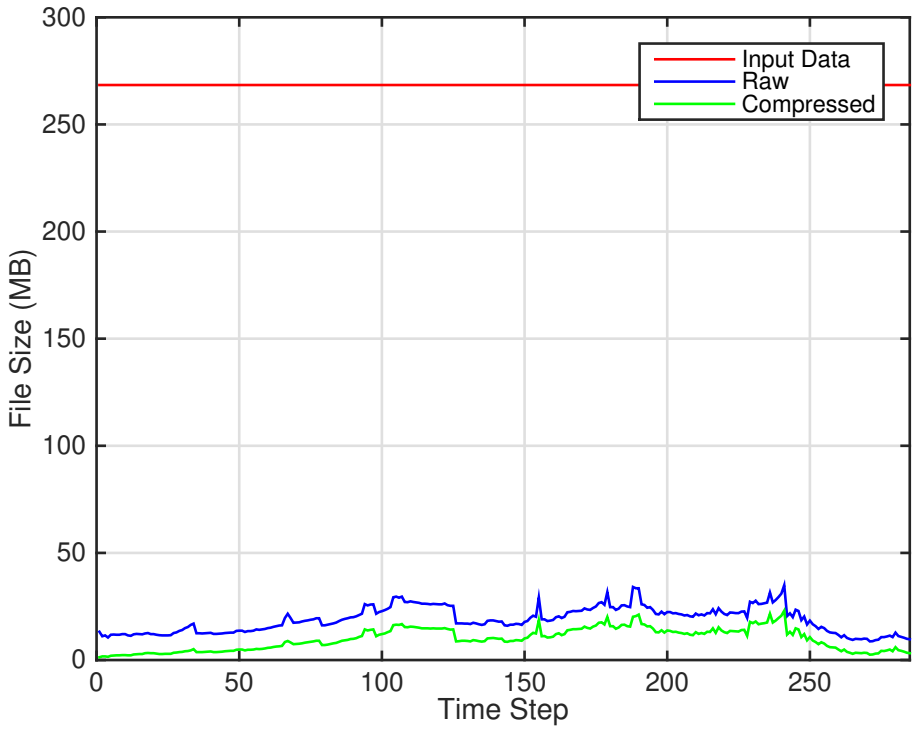


Fig. 4.6: Comparison of raw and compressed layered depth image sizes in relation to the input data size of the *plate* λ_2 data set across the full time range, using 256 branches and the same camera perspective as in Figure 4.4. The reduction in file size is even more prominent for larger input data sets.

when storing the layered depth images in raw format, i.e., without additional zlib deflate compression, which usually achieves further compression ratios of about 50-60%.

4.8.2 Analysis

By design, our approach features a powerful framework for hierachical simplification and automatic segmentation based on the branch decomposition of the contour tree, which is interactively controllable in the viewer application.

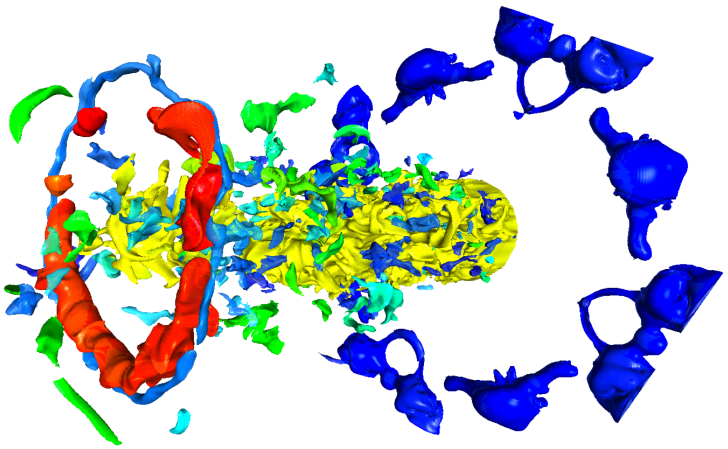
Figure 4.3a depicts a close-up on the central turbulences of the *plate* data set, which is located behind the main whirl visible in Figure 4.4. After an initial simplification of the whole data set to 256 branches prior to layered depth image rendering, the surrounding branches have been manually pruned in the viewer application. The remaining 90 branches visible in the depicted scene have been colored using a HSV color scale based on the branches' average λ_2 values. In the following Figures 4.3b, 4.3c and 4.3d, the branch decomposition subsequently has been further simplified to 50, 30 and 10 branches, respectively. Simplification was performed automatically based on sorted persistence as described in section 4.4, while maintaining the color scheme for non-discarded branches. One can clearly see the incremental reduction in complexity, which has been achieved with minimal user interaction and immediate visual feedback.

A comparison of applying automatic persistence-based simplification to the *jet5* data set is shown in Figure 4.7. In each image, the same scene is depicted consisting of 256 branches in total, with data having been constructed using either velocity (4.7a), vorticity magnitude (4.7b) or λ_2 (4.7c).

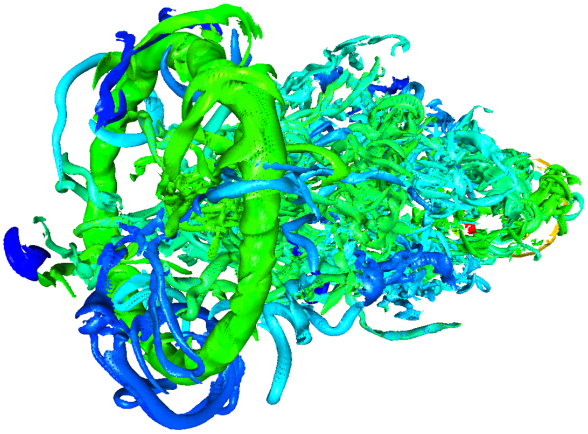
In our studies, the topological segmentation based on the branch decomposition of the contour tree has proven itself useful as a flexible representation of the major structures of interest occurring in the data sets, which furthermore provides an intuitive approach to simplification and filtering in both pre- and postprocessing.

4.8.3 Scaling

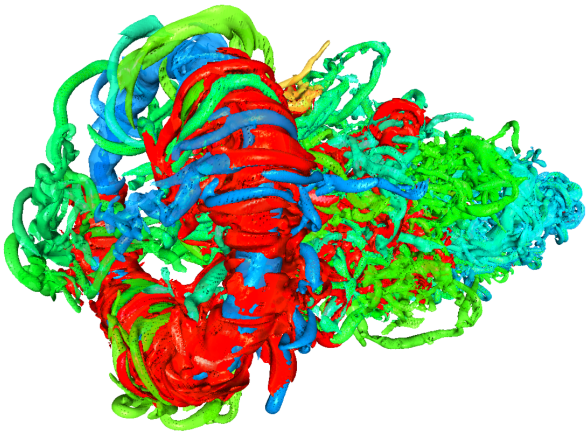
We have studied the scaling characteristics of the task-parallel CPU-based layered depth image renderer on the *Elwetritsch* cluster at TU Kaiserslautern, which is depicted in Figure 4.8, using the example of the *jet5 velocity* data set.



(a) velocity



(b) vorticity magnitude



(c) λ_2

Fig. 4.7: Comparison of automatic persistence-guided simplification of the *jet5* data set into 256 branches and applying an HSV color scale based on the average branch value.

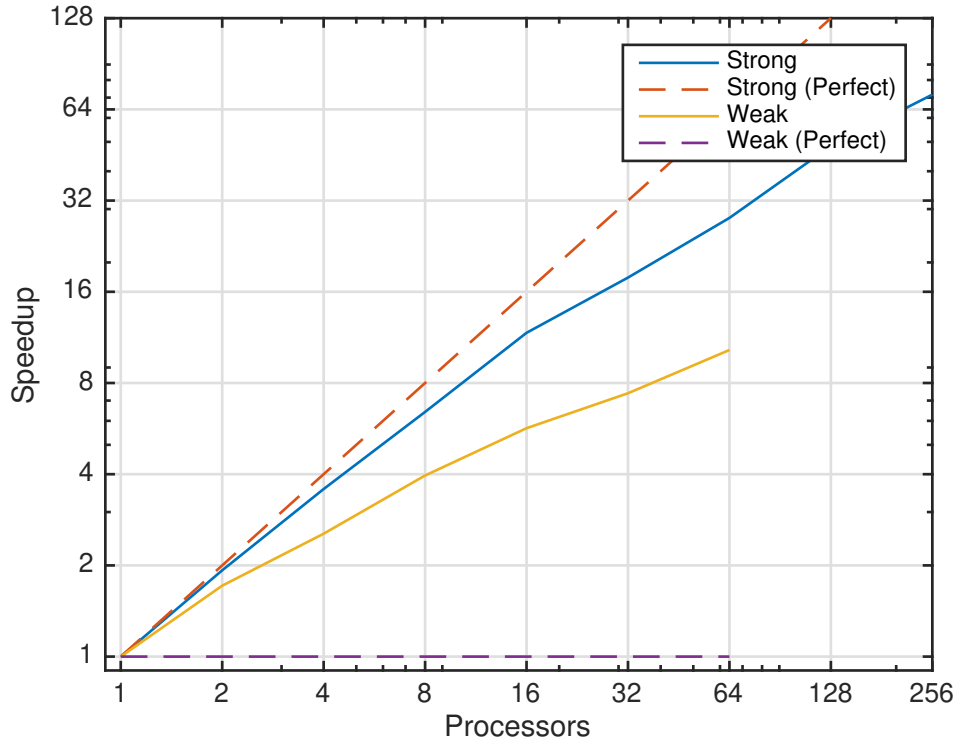


Fig. 4.8: Strong and weak scaling behavior of the distributed layered depth image rendering approach using HPX on the *Elwetritsch* cluster. Subblocks of the *jet5 velocity* data set are distributed across nodes and rendered in parallel. The resulting partial images are composited to a single layered depth image afterwards. We achieve overall good strong scaling and excellent weak scaling characteristics.

Considering strong scaling, we achieve an overall good efficiency. As expected, there is a minor decrease when using more than 16 processors due to the required network communication between nodes for image composition after parallel rendering.

Moreover, the algorithm exposes excellent weak scaling behavior, which is of greater significance to practical considerations than strong scaling. To measure weak scaling, the *jet5 velocity* data set was resampled for each run proportionally to the respective increase in processors used. Interestingly, the results show that bigger volumes can be rendered using proportionally more processors in shorter time.

4.9 Discussion

We have demonstrated the feasibility and potential of combining in situ topological analysis and compact image-based data representation. Our approach significantly

reduces the amount of I/O bandwidth required to store the numerical results of high-fidelity numerical simulations running on large-scale parallel computer systems, while preserving flexibility in visualization.

Based on in situ contour tree analysis and simplification, we obtain a segmented representation of scalar fields contained in the simulation data at every time step. Together with the simplified contour tree, we store a rendering of this segmentation that describes all components visible in every pixel. Rendering can leverage hardware acceleration, as we have demonstrated by the GPU-based implementation of our rendering algorithm. The resulting compressed layered depth images can then be used in post-analysis to flexibly select specific subsets of the segmentation, and perform further topological simplification if required.

While our results already show substantial reductions in output file size, especially for larger data sets, our contribution is intended as a baseline demonstration investigating possible advantages of such an approach for the visualization of large-scale data. We anticipate that many enhancements and improvements of our approach are possible:

- Similar to Ahrens et al. [Ahr+14], our technique could be easily extended to generate a complete in situ image data base from multiple perspectives, which can be combined in the viewer application to enable a flexible 3D data exploration, or even be used for reconstruction purposes.
- Multivariate topological methods such as *Joint Contour Nets* [CD14] might be investigated to obtain improved segmentations.
- A critical shortcoming of our current implementation is frame-to-frame temporal consistency. Since contour trees are computed and decomposed independently at each time step, the resulting contours can vary noticeably between time steps depending on the chosen automatic simplification criteria, potentially undermining analysis due to the lack of frame-to-frame coherence. This crucial problem could be addressed by incorporating feature tracking techniques into the branch selection and simplification process.
- The simplified contour tree stored with the compressed image files could be further annotated to compactly contain relevant properties of the original input data set, thereby improving the power and flexibility of the resulting visualization. However, not only the contour tree, but also the fragments can be used to compactly store local information of the intersected segment useful for later visualization.

- Notably, the general idea of our concept is not restricted to regular scalar data and is easily applicable to different kinds of potentially more complex data structures, providing means for topological segmentation and intersection.
- More sophisticated compression schemes might be used to further increase the compactness of the layered depth images generated.

We will investigate these possibilities in future work.

5 Task-Based Distributed Volume Rendering

5.1 Motivation

High-fidelity computational simulation models have assumed a significant role in scientific research and engineering applications, thereby necessitating efficient visualization techniques at large scale. In recent years, parallel algorithms for concrete classes of visualization problems have been presented, such as direct volume rendering [HBC12] or integral curve computation [Pug+09]. Most large data approaches typically utilize a distributed memory model, where bulk-synchronous execution and communication using the Message Passing Interface (MPI) is standard. For improved scalability, hybrid approaches commonly resort to MPI for the coarse distribution of parallelly executable parts of an algorithm to a set of processes, where within each process a second concept - e.g. OpenMP, OpenCL, or CUDA - is used for additional finegrained parallelization of these steps.

These practices require detailed knowledge of the different parallelization concepts and often result in specific optimizations for certain platform configurations or obligating the usage of distinct hardware components, which complicates or even hinders portability towards other architectures. An additional challenge in the parallelization of visualization concepts is posed by the fact that, in contrast to simulation computations, visualization tasks are frequently bandwidth-limited and inherently unbalanced. Thus, achieving scalable parallel execution demands not only an efficient utilization of the available memory bandwidth, where memory accesses ideally are overlapping with computational tasks, but also dynamic load balancing.

Considering the general parallelization of complex algorithms against this background, in recent years the paradigm of task-based parallelization has been established [DG15]. Here, an algorithm is formulated as a set of tasks which can be carried out concurrently, where a single task represents an atomically executable subsequence of the algorithm. Interdependencies between tasks can be modelled explictly by the developer. These relationships are used by the underlying runtime

environment to coordinate the parallel execution. Thus, with the help of the task graph the developer specifies *what* should be executed, whereas the *how* of the execution is left to the runtime environment [Kai+14].

The task-based paradigm entails several crucial advantages. Conceptionally, tasks enable a more straightforwad formulation of massively parallel programs, where the maximum degree of parallelism is determined by the maximum width of the task graph for the given computation. Technically, the coordinated execution by the runtime environment ensures a flexible and transparent portability to diverse hardware platforms. Furthermore, task-based systems can inherently handle the parallel execution of dynamically changing computational loads by the principle of work stealing [Din+09].

In this context, the intent of this work is to study a task-based system design for distributed direct volume rendering. Our task definition is based on hybrid parallelization in both image and data space (see Figure 5.2), thus representing an effective and easy-to-control trade-off between sort-first and sort-last image compositing. The presented asynchronous binary tree compositing scheme enables good scalability in combination with inherent dynamic load balancing.

The overall intent of this work is to investigate possible advantages of such an approach for the design of large scale visualization systems. Specifically, after a brief review of relevant prior work (Section 5.2), we make the following contributions:

- In Sections 5.3 through 5.4, we describe a task-based formulation for a distributed direct volume rendering system.
- We conduct comprehensive benchmarks to verify the characteristics and potential of our novel task-based system design for high-performance visualization and describe results and analysis in Section 5.5.
- We anticipate that many enhancements and improvements are possible, and discuss a number of such opportunities in Section 5.6.

Our contribution is intended as a baseline demonstration of the applicability of the emerging task-based paradigm in large scale high performance computing to distributed algorithms and challenges in scientific visualization.

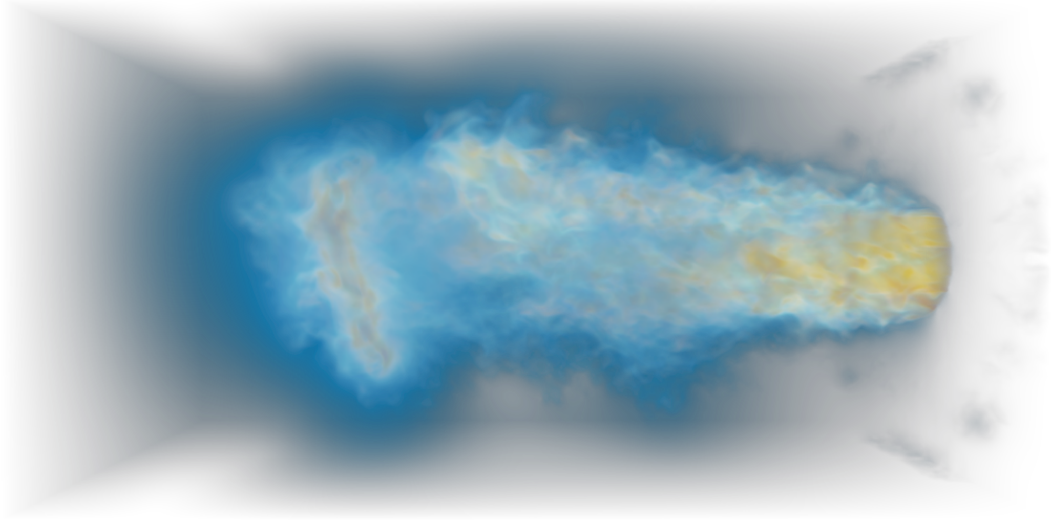


Fig. 5.1: The *jet* reference data set used for all benchmarks with 2048^3 voxels (32 GB) on a single node. For weak scaling, the number of voxels is increased proportionally to the number of cores, up to 6502^3 (275 gigavoxels) on 32 nodes. All renderings are performed at 3840x2160 resolution.

5.2 State of the Art

Direct volume rendering [DCH88] represents a crucial class of algorithms used in scientific scalar field visualization. Today, direct volume rendering typically follows the principle of ray casting [Lev90], where primary rays are traced through a volumetric data set starting at a virtual camera and depending on the underlying sample locations optical properties are determined and accumulated along each ray.

Following the nomenclature of [Mol+94], there are two fundamental approaches for the parallelization of volume ray casting: sort-first and sort-last volume rendering. In sort-first, the image plane is subdivided into rectangular tiles for which rendering is performed concurrently [CM93; Bet+03; Mol+11]. Sort-last algorithms perform parallelization based on a spatially disjunct partion of the input data, where each process computes a partial image of its assigned data. The resulting images are afterwards composed to the final output image. While both techniques have specific advantages and drawbacks [Mol+11], most scalable systems typically employ the sort-last approach, primarily due to the slower increase in image resolution compared to data size. Müller et al. present a hardware-accelerated sort-last approach, using block subdivision for fast empty-space skipping and performing dynamic load balancing by block redistribution based on previous computation times [MSE06]. Marchesin et al. achieve load balancing by dynamic restructuring of the underlying

k-d tree [MMD06]. Navratil et al. use queue-based dynamic scheduling in order to increase ray coherence and memory bandwidth utilization, leading to improved L2-cache access patterns [Nav+07]. Childs et al. present a hybrid scheme, using distributed parallelization both in input data and in image space [CDM06].

Dedicated graphics and accelerator cards providing numerous processing cores have proven to be a powerful tool for computationally expensive applications such as ray casting on large data [BHP15; Kno+14]. Consequently, the prevalent usage of multi-/many-core processor architectures and accelerator cards in distributed high performance systems has given rise to diverse hybrid parallelization approaches. Peterka et al. implement hybrid parallel volume visualization of massive data sets on the IBM BlueGene/P architecture using MPI and POSIX-Threads, where up to 90% of the total runtime are dedicated to I/O [Pet+08]. Fogal et al. study direct volume visualization using OpenGL-based slicing on distributed memory multi-GPU clusters in combination with subsequent MPI-based compositing [Fog+10]. Howison et al. compare different common hybrid approaches based on POSIX-Threads, OpenMP and CUDA in combination with MPI for direct volume visualization, In general, hybrid techniques offer improved performance with reduced memory and communication overhead [HBC12].

A crucial bottleneck in the performance of massively parellel sort-last volume rendering algorthms is the final composition of the partial per-process images. A comparison of the common approaches (Direct Send [EP07; Sto+03], Binary Swap [Ma+94], Radix-k [YWM08; Pet+09; Ken+10]) shows that notable runtime benefits can be achieved using hybrid strategies with variable granularity.

Faced by the emergence of increasingly hierarchical and heterogeneous system architectures, the hybrid MPI-threading model prevalent in high performance computing turns out to be more and more suboptimal. The resulting parallelism is fragile due to the lack of a strict separation between computational kernel and parallel execution, in addition to the strong coupling with the underlying architectures.

In scientific high performance computing task-based dynamic runtime environments are considered as a promising alternative model [DG15], whose benefits have already been demonstrated in diverse disciplines. Haidar et al. present the dynamic scheduling of algorithms in linear algebra [Hai+11]. A dynamic runtime environment for grid workflows can be found in [AA07]. Notz et al. demonstrate a graph-based system design with a dynamic runtime environment for multiphysics software based on partial differential equations [NPS12]. The simulation of large biomolecular systems is shown in [Kal+08]. However, the majority of scientific

applications has not yet integrated dynamic runtime environments, or is still in early experimental stages [DG15].

Current programming languages, libraries or runtime environments start to offer task-based programming models. A comparison of numerous independent runtime environments and task-based execution models can be found in [Gil+13]. Popular frameworks for single shared memory multicore systems are the task implementation in the OpenMP standard (starting with version 3.0), the Intel Threading Building Blocks (TBB) library or Intel Cilk Plus. Our work heavily utilizes the HPX (High Performance ParalleX) framework [Kai+14], which implements the ParalleX execution model and provides task-based parallelization across node boundaries. HPX manages an active global address space and focuses on latency hiding by the dynamic scheduling and asynchronous execution of fine-grained tasks with minimal context switching overhead. Other recent frameworks for distributed task-based parallelization include Charm++ [Acu+14] and Legion [Bau+12], which has been used to explore the applicability of asynchronous many-task (AMT) programming models in the context of in-situ data analysis [Péb+16].

5.3 System Design

5.3.1 Task Granularity

In a task-parallel system the achievable degree of parallelization and scalability is crucially characterized by the so-called task granularity, i.e., the size of the individual tasks, balancing the width of the task graph versus individual task overhead.

Our approach aims to provide a flexibly tunable task granularity by subdividing in both image and data space, thus yielding a hybrid scheme between sort-first and sort-last compositing. Volume data is split into regular blocks of equal size, which are distributed across nodes, whereas the image plane is divided into rectangular tiles. The general approach is to render the visible blocks within each tile, compose all images per tile in correct order and eventually align all completed tile images to form the resulting output image.

An example of the hybrid subdivision in both tiles and blocks is illustrated in Figure 5.2. The corresponding communication tree for image compositing is depicted in Figure 5.3, also showing the interleaved scheduling order of the individual rendering and compositing tasks.

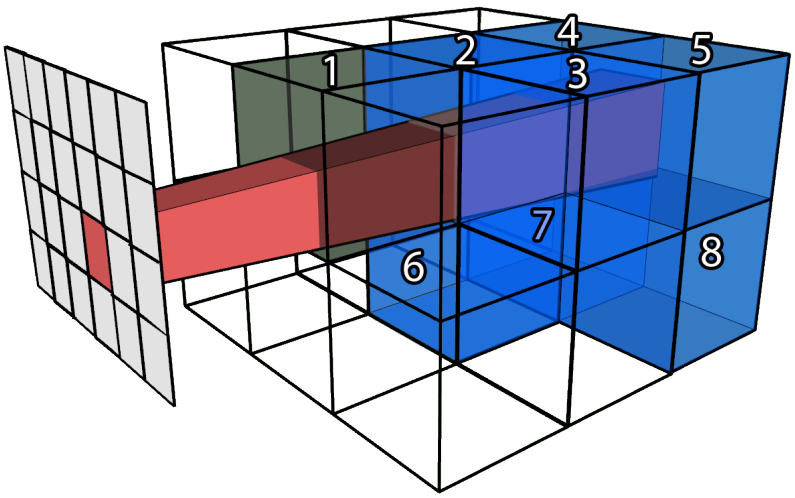


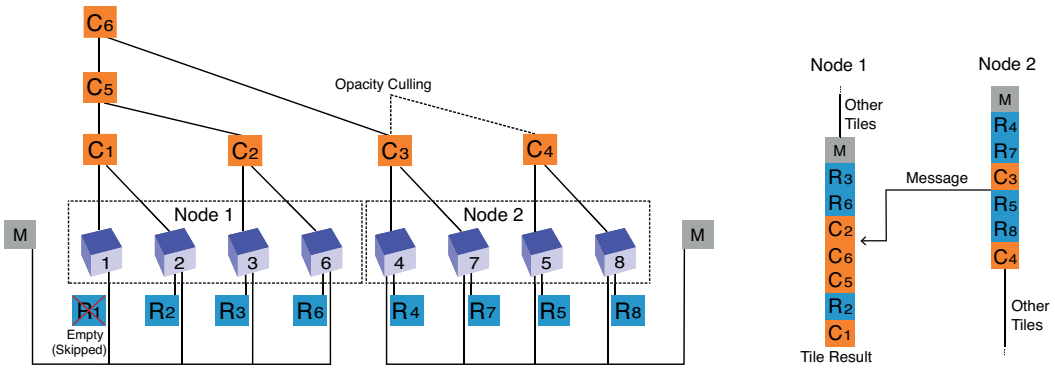
Fig. 5.2: Hybrid parallelization in both image and data space. For each tile, the set of visible blocks is topologically sorted and composited using a distributed binary tree communication scheme. The numbering represents an arbitrary enumeration of the visible blocks within the highlighted tile's frustum. See Figure 5.3 for the corresponding compositing tree. Note that block 1 is discarded due to empty-block skipping.

This scheme allows to balance the number of mutual partners for image compositing by tweaking both block and tile size, while also providing means to incorporate common optimization techniques such as empty-block skipping and early ray termination.

The following Section 5.3.2 describes the interplay and dependencies of the individual rendering and compositing tasks in our novel distributed compositing scheme. The incorporation of additional optimizations is presented in Section 5.3.3.

5.3.2 Distributed Compositing

Initially, each node computes for all tiles the visible blocks within the respective viewing frustums and schedules for each local block a rendering task. As these render tasks begin execution, each node calculates a sequence of compositing steps for each tile. A compositing step consists of an initiating block, whose rendered image will be blended behind the image of the receiving block. The goal is to find a sequence of compositing steps so that every step maintains the correct z-ordering of fragments and ultimately after the last step one block holds the complete image for the respective tile. The node containing that last block will contribute the tile to the final output image.



(a) Communication tree **(b)** Scheduling timeline

Fig. 5.3: Communication pattern and possible task execution order for the eight blocks (numbered cubes) shown in Figure 5.2 after topological sorting (from left to right) distributed across two nodes. In this example, for each block a rendering task (R) is scheduled by the respective node, except for the first block which is empty. Additionally, each node calculates the necessary compositing steps and supplies each local block with the resulting meta information (M) it needs for sending to its receiving block. According to this meta information the ready blocks may send their image data to their receiving blocks, which will initiate the scheduling of composition tasks (C). In this example, the composited image of the partial images from block 4 and 7 is already opaque, allowing early initiation of the next compositing step, thereby skipping the wait for the result of C4. The execution timeline assumes only a single local thread per node for simplicity. Due to the asynchronous execution of tasks the results of C2 and C3 are composited before C1 is performed. Note, that the execution of tasks may be interleaved with tasks from other tiles.

First, all (including remote) blocks inside the viewing frustum of the tile are topologically sorted based on the Manhattan distance to the camera, so that each block in the sorting can never overlap a block prior to it.

A naïve approach would be to iterate over this sorting from back to front and schedule a compositing step for each block as initiator and its successor in the sorting as receiving block, thus yielding a correct, sequential compositing of all block images. This would result in every block being the initiator and receiver of a composition step exactly once, except for the frontmost block who never initiates a compositing and the backmost block who never receives one.

For a scheduled compositing step to start and execute the following dependencies are necessary: First, the rendering task for the initiating block needs to be finished. Second, the compositing step which will be received by the current initiator needs to be completed. An exception to this is the backmost block who can initiate compositing immediately after rendering.

Our communication pattern is more sophisticated than the aforementioned naïve sequential approach. Compositing steps are chosen to form a binary tree over the topological sorting of the involved blocks across node boundaries. Every second block initiates just after rendering is completed for the tile, with its successor in the sorting being the receiver. Every second of these receivers is scheduled to initiate a compositing step with its successive receiver and so on. In the end, every block was initiator exactly once, half of the blocks received once, a fourth received twice and so on. The number of times the frontmost block is receiver of a compositing step is equal to the height of the binary compositing tree formed by the compositing steps.

Note that while the compositing steps are determined and scheduled per level of the tree, there is no barrier in their execution. Each compositing step can be performed once the initiating block has rendered its image and all compositing steps scheduled with him as receiver have been completed. This allows the tree to be worked off in any order and in parallel as long as these dependencies are met.

Furthermore, each node can determine the structure of the compositing tree completely on its own and inform all local blocks appropriately about their roles as initators or receivers, thus completely avoiding costly network communication for coordination.

5.3.3 Optimization

Note that a block for which two compositing steps are scheduled with him being the receiver can merge the two images he receives even before his own rendering task is done, since the compositing steps are associative. However, since the calculation of the compositing steps is performed redundantly on each node and in parallel, one node might finish this calculation for a certain tile long before others. This would require to buffer incoming images for receiving blocks in the nodes lagging behind until they have scheduled the corresponding compositing steps.

To circumvent this buffering, the initiating block of each compositing step not only sends its image for compositing to the receiving block, but also the number of compositing steps the receiver should execute before dealing with the current step. This allows receiving blocks to merge multiple received images before their own image is ready and even before the compositing steps are calculated on its node. Consequently, the successful completion of these local operations is now only necessary to initiate a compositing step, not for being the receiver of one. This

sender-initiated approach allows to perform each compositing operation as early as possible.

Additionally, we have implemented two common acceleration techniques for volume rendering: empty-block skipping and early ray termination. While empty-block skipping can be trivially integrated as a data preprocessing step during block initialization, it should be noted that there is a strong relationship to the optimal block size, as smaller blocks are more likely to be completely empty and can be skipped in compositing.

In constrast to early ray termination in standard shared-memory ray casting, our distributed tile-based compositing scheme performs opacity culling at tile granularity. After each compositing operation, the resulting image is checked against a predefined opacity threshold. Whenever the opacity of all fragments is saturated, all outstanding blend-under compositing operations can be skipped and the tile can be immediately forwarded in the compositing tree. The corresponding superfluous rendering tasks that have not been started yet can be removed from the scheduling system, whereas the resulting images of already executing rendering tasks will simply be ignored.

To assist early opacity culling, we have implemented a custom priority queue task scheduler, where pending block rendering tasks are dynamically kept sorted based on their Manhattan distance to the camera. This ensures full compute occupancy at any time while improving the execution order with respect to opacity culling as more rendering tasks are scheduled.

Figure 5.3 illustrates an exemplary compositing tree pattern and possible task execution timeline on two nodes for the scenario presented in Figure 5.2, featuring both empty-block skipping and opacity culling.

5.4 Implementation

Our novel task-based distributed rendering approach is based on the HPX (High Performance ParalleX) framework [Kai+14], an aspiring task-based runtime environment with means for asynchronous communication across nodes. Each block is represented as an individual component in the active global address space (AGAS), allowing blocks to directly communicate in the compositing pattern. The custom priority queue scheduler is implemented on top of HPX's standard FIFO scheduler by manually keeping track of the number of rendering and compositing tasks being executed by HPX.

OSPRay [Wal+17] is used as rendering backend with its default internal TBB-based parallelization being disabled. A seperate scientific visualization renderer is instantiated for each CPU core, so all scheduled rendering tasks can render concurrently. Each block aggregates a shared structured volume instance and a pre-committed model instance, which is set as active model in the respective executing renderer.

Manual AVX2 intrinsics are used for standard blend-over image compositing, which allows to perform vectorized instructions on 8 consecutive RGBA pixels with 8 bit per channel. The same degree of vectorization was not achievable by relying on compiler-generated auto-vectorized code.

5.5 Results

To investigate the characteristics and potential of our novel task-based rendering system we have conducted comprehensive benchmarks with respect to optimal task granularity, task scheduling and scaling.

The *jet* data set (see Figure 5.1), which results from a direct numerical simulation of a jet of high-velocity fluid entering a medium at rest, was used with a standard fire and ice transfer function, thus yielding empty, transparent and fully opaque image areas. As reference configuration the data set was resampled to 2048^3 voxels (32 GB) on a single node. Timings were measured by rendering a full rotation around the data set at 3840x2160 resolution and computing the mean. The camera distance is adjusted to view the complete volume.

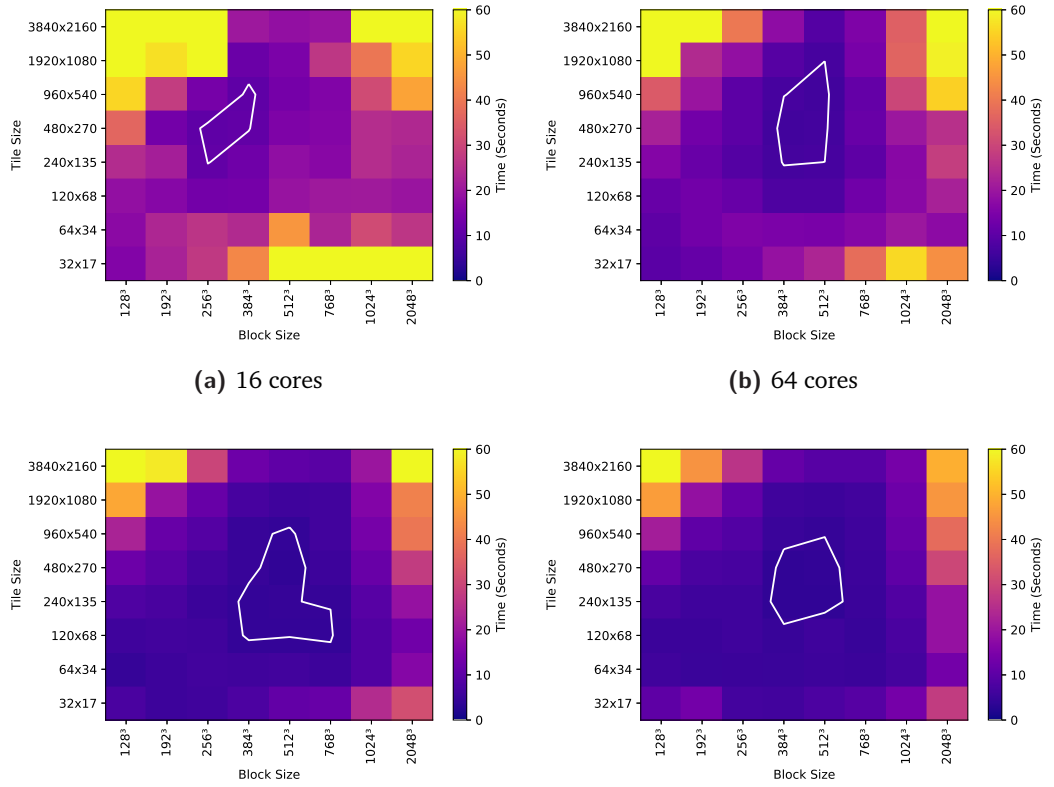
We have identified four important base scenarios to focus on: *in-situ* vs. *offline* and *weak* vs. *strong* scaling up to 512 cores. The in-situ scenario assumes block data is already in memory (e.g. after a preceding simulation run), whereas offline rendering requires additional on-demand I/O to load blocks into memory. For weak scaling, the total data size is upscaled proportionally to the number of cores, e.g. 6502^3 for 512 cores (approx. 275 gigavoxels). Strong scaling keeps the data size constant while increasing the number of cores.

All benchmarks were performed on the *Elwetritsch* cluster providing two Intel E5-2637v3 CPUs (16 cores) per node, 64GB of main memory and InfiniBand QDR interconnect.

5.5.1 Task Granularity

The granularity of the individual tasks in a task-based system crucially defines and limits the degree of possible parallelization and scalability. In our approach, we have two parameters to control task granularity: block size and tile size.

Figure 5.4 illustrates the mean rendering times across the full spectrum of block and tile sizes for 16, 64, 256 and 512 cores in the weak in-situ scenario, thus focusing on pure rendering performance without I/O. The sweet spots are clearly indicated in the middle ranges of both block and tile size parameters, with severe performance penalties in the extreme corner cases. The optimal configuration shifts slightly towards smaller tile sizes and larger block sizes as the number of cores increases.



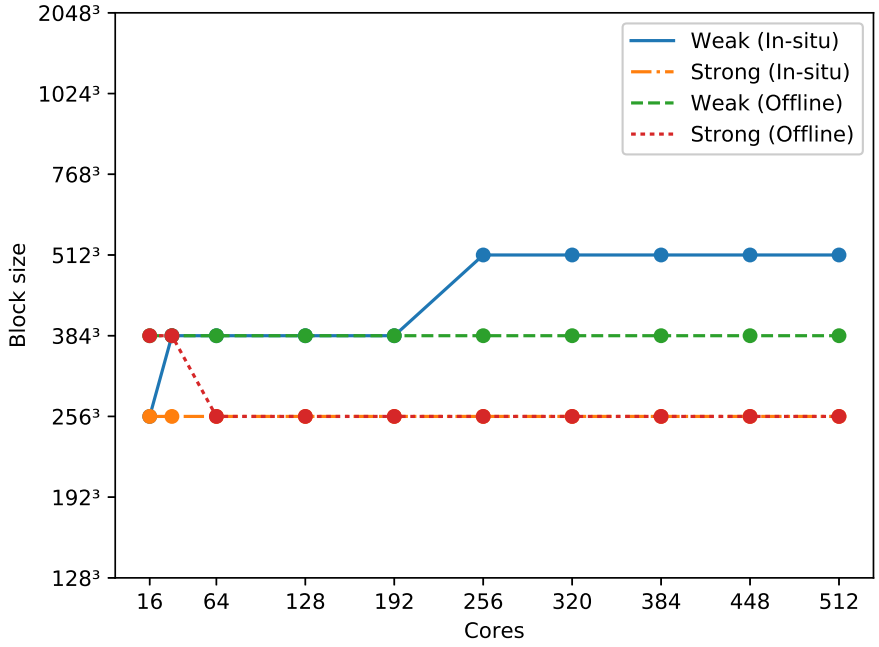
(a) 16 cores

(b) 64 cores

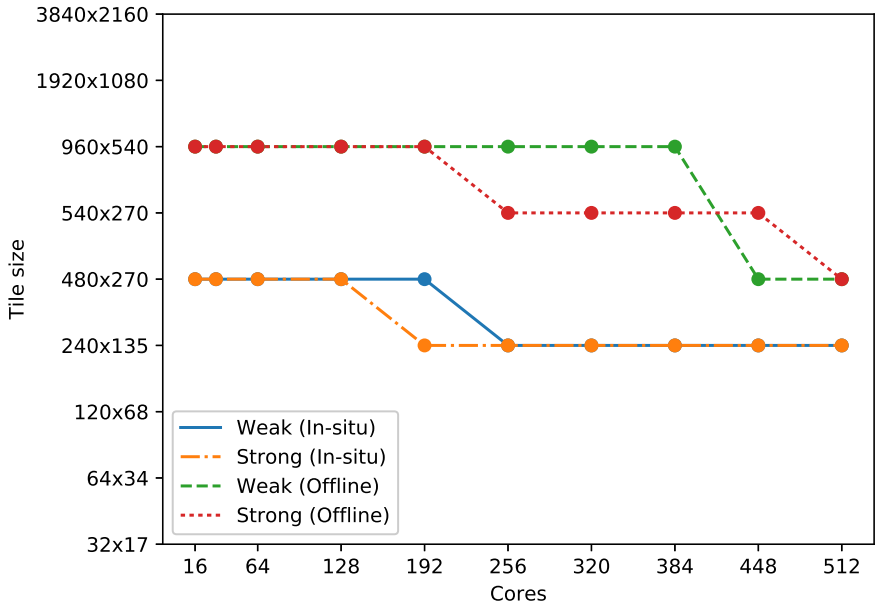
(c) 256 cores

(d) 512 cores

Fig. 5.4: Mean rendering times for different task granularities at 16, 64, 256 and 512 cores (2048^3, 3250^3, 5160^3 and 6501^3 voxels, respectively) in the weak in-situ scenario. The sweet spots are indicated as white contour lines with a threshold of 0.5 seconds around the best rendering time. Optimal performance is achieved in the middle ranges of both block and tile size parameters, with severe performance penalties in the extreme corner cases. The optimal configuration shifts slightly towards smaller tile sizes and larger block sizes as the number of cores increases.



(a) Optimal block size



(b) Optimal tile size

Fig. 5.5: Optimal block and tile sizes for up to 512 cores in the in-situ and offline scenario (both weak and strong scaling). The ideal block count appears to be independent from the number of cores. However, in offline rendering the block size additionally influences I/O performance. In general, the optimal tile size decreases with the number of cores. In the offline scenario larger tiles are beneficial.

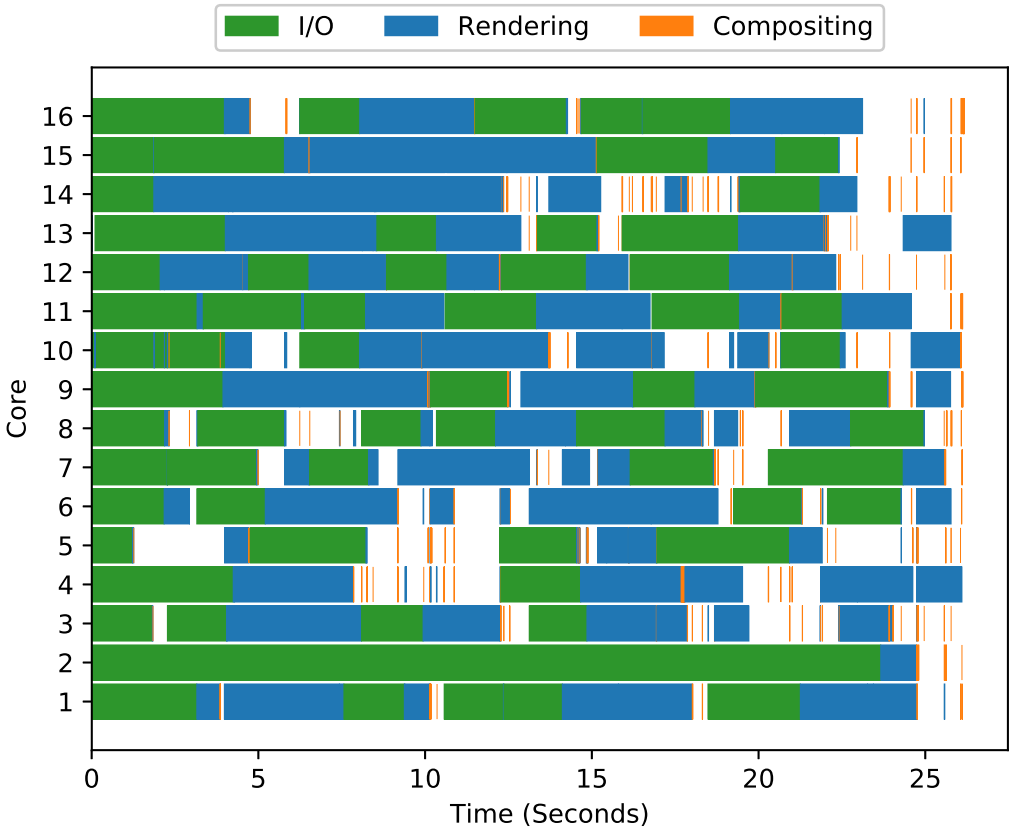
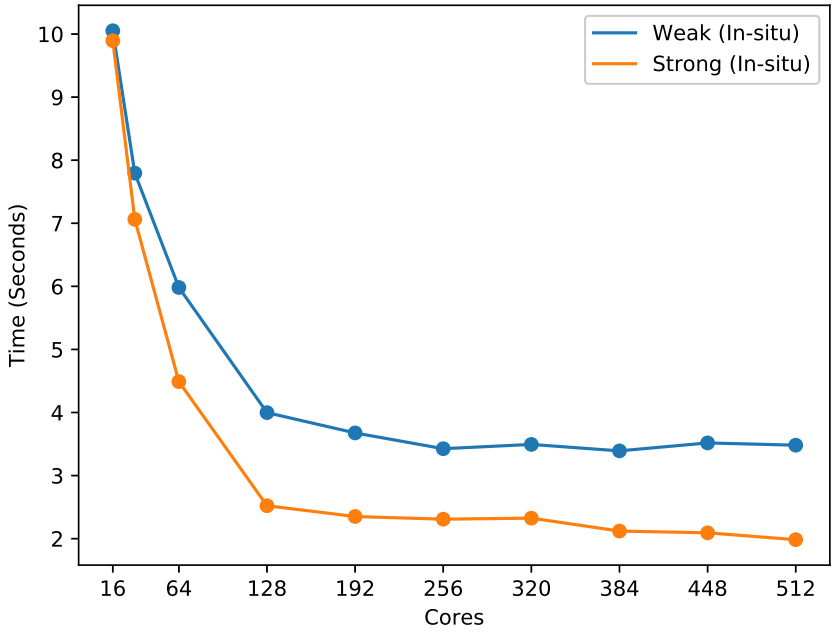
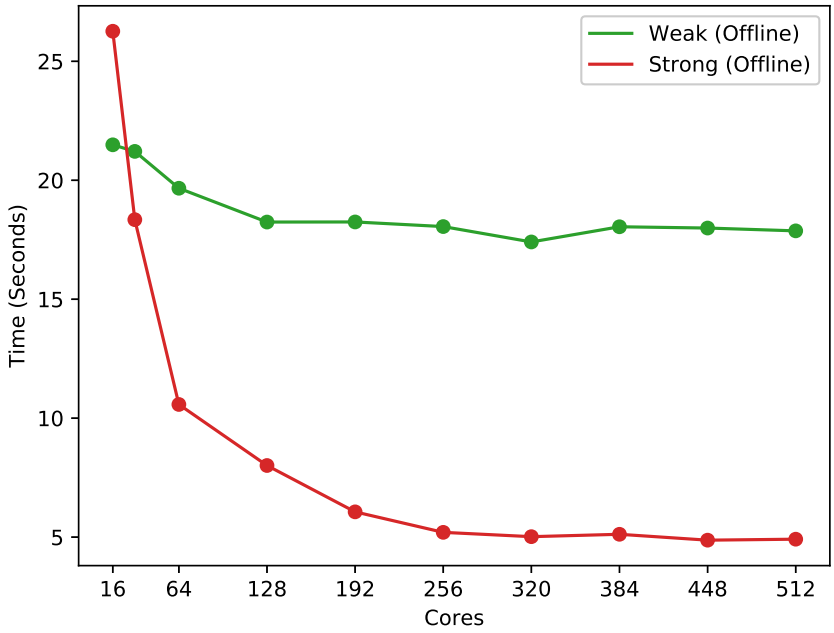


Fig. 5.6: Task scheduling for offline rendering of 2048^3 voxels using 16 cores on a single node at block size 512^3 and tile size 960x540. Block loading, rendering and compositing happen interleaved without barriers, so the resulting image is immediately ready after the final rendering and compositing tasks. Whitespace indicates unmet task dependencies such as outstanding I/O tasks or compositing partners.

The optimal block and tile sizes for all four scenarios across different core numbers is depicted in Figure 5.5. The ideal block count appears to be independent from the number of cores, in both weak and strong scaling scenarios. However, in offline rendering the block size additionally influences I/O performance. In general, the optimal tile size decreases with the number of cores as the increased fine-grained subdivision promotes latency hiding by task overlapping. Interestingly, in the offline scenario larger tiles are beneficial. A possible explanation is that larger tiles improve the early scheduling of on-demand I/O tasks.



(a) In-situ scaling



(b) Offline scaling

Fig. 5.7: Weak and strong scaling up to 512 cores for both in-situ and offline scenarios. The corresponding block and tile sizes are depicted in Figure 5.5. After an initial performance improvement, weak scaling shows for both in-situ and offline cases approximately constant runtime. Strong scaling rendering times in the offline scenario drop rapidly as the I/O overhead is distributed across nodes.

5.5.2 Scheduling

Figure 5.6 depicts the scheduling of the individual I/O, rendering and compositing tasks in the offline scenario onto the 16 available cores of a single node. The diagram is representative for both the single and multi node cases.

Clearly, there are no barriers in our task-parallel approach. Block loading, rendering and compositing happen interleaved, so the resulting image is immediately ready after the final rendering and compositing tasks. Especially the latency of costly I/O is hidden by automatic overlapping with computational tasks. The compositing tasks themselves are rather cheap in comparison to I/O and rendering.

5.5.3 Scaling

We have studied both weak and strong scaling characteristics of our approach in the in-situ and offline scenarios, as depicted in Figure 5.7. Note that in these benchmarks scaling only refers to the data size per node. However, for each benchmark the camera is adjusted such that the complete volume rendering is visible and its image area stays constant, thereby reducing the contribution of each node to the final image at bigger node counts.

After an initial performance improvement, weak scaling shows for both in-situ and offline cases approximately constant runtime, which is near optimal. This initial improvement is explained by the quick reduction in image contribution (i.e. rays) per node. Strong scaling rendering times in the offline scenario drop rapidly as the I/O overhead is distributed across nodes.

In general, strong scaling seems to be relatively limited in the tested scenarios. However, weak scaling suggests that strong scaling would improve at bigger workloads.

5.6 Discussion

We have demonstrated a novel approach to large scale volume rendering based on distributed task-based runtime environments, an emerging trend in modern high performance computing on increasingly heterogeneous architectures. Our technique is based on a hybrid task-definition using parallelization in both image and data space, representing an effective and easy-to-control trade-off between sort-first and sort-last image compositing.

In our distributed asynchronous compositing scheme, each node determines the set of visible blocks for each tile. After an initial topological sorting, a sender-initiated binary tree communication scheme is used to correctly compose all block images within a tile. The compositing scheme incorporates common optimization techniques such as empty-block skipping and opacity culling, which is aided by a custom task priority scheduler based on the Manhattan distance to the camera.

We have conducted comprehensive benchmarks to study the characteristics of possible block and tile configurations in order to achieve optimal task granularity. The employed asynchronous binary tree compositing scheme enables good scalability in combination with inherent dynamic load balancing. The dynamic scheduling of initialization, rendering and compositing tasks on a single node ensures good latency hiding of network communication and I/O access.

Our contribution is intended as a baseline investigation of the applicability of task-based runtime environments to distributed scientific visualization. We anticipate that many enhancements and improvements of our approach are possible:

- Distributed work stealing would be an interesting approach to implement proper load balancing across node boundaries. Since each block is represented by an individual component in HPX's active global address space, the migration of blocks could be performed transparently with little to no modifications to the distributed compositing scheme. Distributed load balancing is especially important in an interactive setup with user-controlled camera navigation.
- In conjunction to distributed work stealing, a more sophisticated scheduling algorithm could also be used to improve task overlap and ensure the available I/O bandwidth is always kept saturated while executing rendering tasks as long as there are still blocks left to be loaded.
- Out of core handling could be used to support larger block counts on individual nodes.
- So far, our approach relies on the regular structure of blocks at various places. Support for unstructured data would involve complex enhancements to the distributed compositing scheme.
- In general, performance benchmarks on larger core counts would be very interesting. The integration of additional accelerator cards such as Intel Xeon Phi or GPGPUs is theoretically easy in the spirit of task-based runtime environments, but in practice still technically challenging.

- Additional benchmarks for comparison against traditional non-task-based approaches are required to further characterize the benefits and drawbacks of the presented approach. We do not expect significiant benefits from applications that already scale well on large machines using traditional data parallel approaches, especially if they are highly tuned and optimized towards a specific system or architecture. However, as elaborated in Section 5.1, we believe the major promising advantages of task-based designs lie in their portability to diverse and heterogeneous architectures, as well as the conceptionally more straightforward formulation of massively parallel programs.
- Besides distributed rendering, other load-sensitive techniques from scientific visualization such as topological methods or integral curve computations would certainly make promising candidates for task-based parallelization.
- Once enough task-based designs of standard visualization algorithms exist, their interplay and dependencies in a (complex) visualization pipeline could be studied.
- The task graph could be used for theoretical models and estimates about runtime, possible parallelization and scalability.

We will investigate these possibilities in future work.

6 Hardware-Accelerated Multi-Tile Streaming

6.1 Motivation

The growing use of distributed computing in computational sciences has put increased pressure on visualization and analysis techniques. A core challenge of HPC visualization is the physical separation of visualization resources and end-users. Furthermore, with increasing dataset sizes, in-situ scenarios, and complex visualization algorithms, transfer to a separate visualization system becomes impractical. Modest demand for interactivity, low screen resolutions and user bases on relatively high-speed connections made frame based compression sufficient to provide a workable remote visualization experience. With novel interactive workflows, commodity high-resolution monitors, complex rendering algorithms, latency sensitive display technologies and globally distributed user bases, new approaches to solve the remoting challenge are required.

The wide availability of GPUs in current and future generation HPC systems allows not only to leverage the GPUs' rendering capabilities, but also their special purpose video en-/decoding hardware: to both weakly scale and render significantly more pixels without requiring large amounts of streaming bandwidth, and to strongly scale the rendering tasks and reduce the overall latency.

Being able to drive high-resolution displays directly from a remote supercomputer opens up novel use cases. In particular, it enables cheaper infrastructure at the client's side, as all the heavy lifting is done on the server side. It also allows the visualization and rendering system to scale with the scale of the simulation, rather than having to scale a separate system for visualization of large-scale data sets. As will be shown in Section 6.2, driving remote tiled displays is nothing new. However, contemporary resolutions of at least 4K or more per display at interactive frame rates far exceed the capabilities of previous approaches.

Strong scaling the rendering and delivery task enables novel interactive uses of HPC systems. Splitting the rendering load enables expensive rendering solutions

at interactive frame rates. This can improve perception when visualizing a high-resolution simulation's results. In addition, the renewed interest in virtual reality with head mounted displays begs the question for streaming directly from the HPC system.

In this comprehensive case study, we demonstrate the impact of video compression and multi-tile streaming for low-latency distributed remote rendering at interactive frame rates. Using comprehensive benchmarks we demonstrate the practicability of this approach and the impact on possible workflows and visualization scenarios. With these investigations, we address several basic questions:

- Is it feasible to stream content directly from a supercomputer to to remote large-scale tiled displays at sufficiently high frame rates (e.g. interactively)? Which latency and bandwidth requirements does this entail, and how do they correlate to image contents?
- How does hardware-accelerated progressive video compression compare to conventional CPU-based compressors applied to individual frames?
- What frame rates can be delivered to a remote end-user by strong scaling the rendering and delivery task, i.e., using video hardware for direct-send sort-first compositing?
- Could this even be used for latency-sensitive environments such as VR?

It is not our intention to validate remote visualization as a general approach, but rather to highlight possible process improvements and streamline the end-user experience through the use of hardware-accelerated video compression.

The outline of the chapter is as follows. In Section 6.2 we provide some background on related activities in this field. Sections 6.3 and 6.4 show the general setup of our multi-tile streaming approach using hardware-accelerated video compression. Section 6.5 demonstrates the achievable frame rates that our system provides in various configurations. Possible opportunities for enhancements are discussed in Section 6.6.

6.2 State of the Art

As observed above, remote visualization techniques have by necessity been in practical use since the advent of computational sciences. Fundamentally, this is a consequence of the fact that end-user analysis resources cannot be expected to

scale with data-production resources, severely limiting visualization capability for state-of-the-art problems.

A substantial body of previous work focuses on using dedicated computational resources (such as a visualization server or visualization clusters) where images are generated and transmitted to commodity hardware such as a PC or mobile device. Engel et al. [ESE00] observe the necessity to access remotely available high-performance visualization clusters, and provide a web-based interface to the remote servers. Lamberti and Sanna [LS07] and Noguera and Jiminez [NJ16] examine the challenges and opportunities of using low-powered, mobile viewing devices, which naturally integrate into this setting. Several general purpose frameworks have been described in the literature, e.g. SAGE2 [Mar+14] and Equalizer [EMP09], but standard visualization tools such as e.g. VisIt [Chi+12] and ParaView [AGL05; Her+08] also support corresponding modes of operation. As visualization is typically used in an interactive setting, latency is important. Stegmaier et al. [Ste+03] describe improvements to naive image streaming aimed at improving poor interactive performance due to high latency. Most notably, they find image compression to be beneficial in reducing latency. Focusing especially on compression of visualization images, specific solutions can be developed for particular algorithms. For example, Cui et al. addressed latency by transmitting annotated depth images from which different viewpoints can be reconstructed without data retransmission [CMP14]. Similarly, Lalgudi et al. [Lal+09] exploit view coherency to derive effective compression for volume rendering. Similar techniques have also been used in other applications, e.g. remote gaming [FE10]. While both these works exemplify non-standard compression schemes for specific visualization techniques, it is difficult to extend them to a general setting.

The present research is inspired by the work of Jiang et al. [Jia+16], who describe a lightweight general purpose image compression library that utilizes the video compression hardware on NVIDIA GPUs [NVI]. They use temporal and spatial image coherence during compression to obtain 25x improved compression ratios at reduced latency, and demonstrated the benefits of their approach through integration with ParaView for single-tile streaming. The compression scheme is based on the H.264 standard [Wie+03], and is thus in principle a lossy approach. However, this is acceptable in practice as evidenced by the nowadays ubiquitous use of such codecs in the entertainment industry. While to the best of our knowledge the specialized video hardware available in modern GPUs is still mostly unused in high performance visualization, an interesting alternative application was recently presented by Leaf et al. [LMM17], who have demonstrated in-situ compression of floating-point volume data using hardware encoders.

In this work, we take the hardware-accelerated video streaming approach to the extreme by applying it to diverse large-scale multi-tile scientific visualization scenarios. Although demonstrated on vendor hardware, the technique is in principle vendor-independent and could be extended to utilize corresponding hardware in Intel [Int] and AMD GPUs [AMD], which provide similar codec capabilities. Furthermore, given modern hardware support, it appears feasible to adopt the HEVC standard [Sul+12] that improves image quality while retaining strong compression. In this work, we consider both H.264 and HEVC codecs, but focus on the still more ubiquitous H.264 variant, given its wide availability of hardware implementations.

A necessity of using distributed resources to compute visualization images in a remote scenario is compositing: to send a single image to the end-user, partial results computed on different nodes must first be composited onto a single node. This implies that the image data incurs latency twice – first when sent to the compositing node, and a second time when sent to the client. Since compositing speed and latency can dramatically limit end-to-end performance of remote visualization systems, a significant body of prior work has therefore investigated how to in particular conduct the compositing phase with minimum latency. Corresponding strategies broadly fall into several classes. In direct send compositing [EP07; Sto+03], render nodes directly send pixel data to the compositing node. The binary swap [Ma+94] and radix-k [YWM08; Pet+09; Ken+10] strategies improve on this by intelligently spreading intermediate compositing operations across many nodes and exchanging pixel data using optimized communication schemes. The performance and scalability of corresponding implementations are demonstrated e.g. in the IceT framework [Mor+11].

In constrast, in this work, we demonstrate that it is feasible and advantageous to perform direct-send compositing on the client and display tiles at very high frame rates. This enables a system design in which render nodes directly send to the end-user client. We set out to demonstrate that with vastly improved (de-)compression, even a large number of render nodes do not overwhelm the recipient of their images.

6.3 Multi-Tile Streaming

Hardware-accelerated multi-tile streaming is a promising approach to make the distributed rendering capabilities of the GPUs within a remote HPC system directly

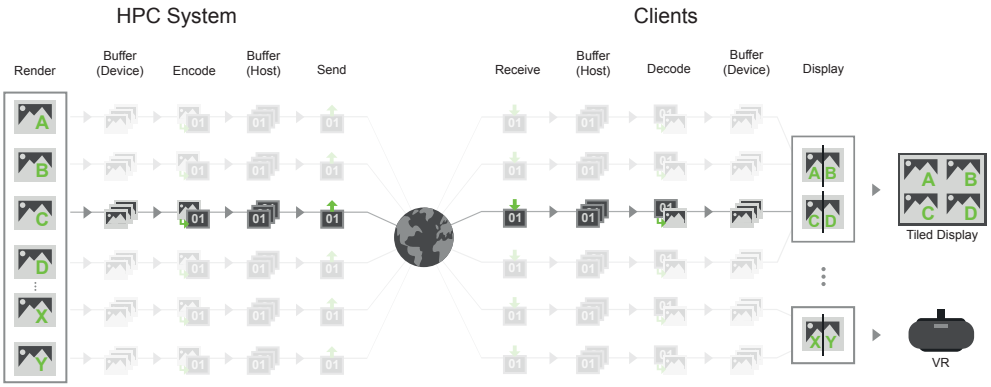


Fig. 6.1: Conceptual overview of the multi-tile streaming approach using asynchronous pipelines. Several tiles are streamed directly from the compute nodes' GPUs in the HPC system (left) to muliple client GPUs for display (right). Each pipeline consists of individual threads for hardware-accelerated en-/decoding and communication. The threads within a pipeline interact through buffers that are strategically placed on device or host memory, thereby minimizing costly PCIe bus transfers. Frames rendered by the source GPU are immediately encoded and sent to the destination GPU for direct decoding to the display engine.

accessible to visualization systems, such as tiled displays, CAVEs [Feb+13], workstations, virtual reality (VR) headsets, thin clients or mobile devices.

A multitude of contemporary GPUs is equipped with dedicated hardware units that enable low latencies and interactive frame rates for high-resolution remote streaming. For instance, NVIDIA GPUs contain one or more hardware-based decoders and encoders (separate from the CUDA cores) that provide hardware-accelerated video decoding and encoding for several popular codecs [NVI]. With decoding/encoding offloaded, the rendering engine and the CPU are free for other operations, such as visualization, computation, and data management. In contrast to the stand-alone compression of individual frames based on compression algorithms such as LZ4, Squirt (run-length encoding) or Zlib that have been used in previous remote visualization applications [AGL05], progressive video-based encoders such as H.264 or HEVC provide better compression ratios by exploiting not only the spatial coherence within but also the temporal coherence between frames in scientific visualization applications. Using the hardware-accelerated codec units of NVIDIA GPUs, we have implemented a prototypical client/server library providing concurrent streaming pipelines between source and target applications.

Conceptually, a pipeline connects two GPUs and streams (parts of) a source image from the rendering framebuffer to a remote destination for display. Figure 6.1 illustrates multiple concurrent pipelines connecting rendering nodes from a remote HPC cluster and local visualization systems driving tiled displays or virtual real-

ity devices. A streaming pipeline essentially consists of three stages for encoding, transmission and decoding. Technically, each pipeline is implemented as a series of parallel tasks connected through thread-safe queues. This ensures the high degree of asynchronicity within each pipeline that is crucial to achieve high throughput at low latencies. All involved hardware units, i.e., rendering/display (GPU), encoding/decoding (GPU) and network transmission (CPU), work concurrently. Depending on the stage, buffers are strategically placed on either device or host memory to minimize host bus transfers. Notably, host memory transfers can be completely avoided by using DMA transfers (e.g. GPUDirect RDMA on recent NVIDIA GPUs). This technique requires certain hardware configurations and is out of the scope of this work.

The following section describes all stages in the life of a streamed frame (or tile) through the pipeline from source to destination. After rendering the main render thread accesses the framebuffer and pushes the requested tile into the device-memory encode buffer. The encode thread pulls from its input buffer and forwards the raw image data into the hardware encoder. The resulting compressed bitstream is enqueued into a host-memory send queue. Compressed data in the send buffer is continuously transmitted onto the network by a separate thread. On the client side, a pipeline consists of a receive thread and a decode thread. As the receive thread receives compressed frames from its network socket, it pushes them into the host-memory decode queue. The subsequent decode thread pulls compressed data from its input queue and performs hardware-accelerated decoding. The resulting raw tiles are placed into a device-memory display queue. The main display thread of the client application pulls from the display queue to assemble the complete output image. A single server or client process can contain one or more concurrent pipelines. For instance, a server application could split the framebuffer into two tiles and use two concurrent pipelines to better utilize the available hardware encoding units. A similar approach can be employed at client-side, either to decode and display full frames to distinct monitors, or composite partial tiles of a single display.

For synchronization purposes the frame number of each tile is passed through the pipeline, in addition to latency statistics. While synchronization between server processes is optional depending on the context (e.g., typically already provided by the underlying simulation in case of in-situ scenarios), synchronization at client-side is obligatory to ensure consistent display of all contributing tiles. In our benchmarks we perform synchronization across processes at both server-side (before frame grab) and client-side (before display). Note that by synchronization across processes we only refer to processes contributing to a single output device, e.g., all pipelines to a tiled display wall in case of weak scaling or all pipelines to a single display in case

of strong scaling. There is no need for precise per-frame synchronization between seperate isolated output devices.

6.4 Implementation

We have implemented the multi-tile streaming approach based on concurrent asynchronous pipelines as a server-client library that can be combined with arbitrary rendering applications. Both server and client internally use MPI for synchronization across processors and support using multiple GPUs for encoding and decoding, respectively. Newly created pipeline instances are assigned to the node's available GPUs in a round robin way.

The streaming server creates a standard TCP socket and listens for incoming connections. The use of the TCP protocol instead of UDP relieves from additional frame/packet loss handling, while still enabling high performance as demonstrated in Section 6.5. For each connection, a server pipeline instance is created as described in Section 6.3. The rendering application can retrieve the bounding rectangles of the required tiles from the server instance to optimize the rendering process by restricting rasterization or ray casting to the relevant regions. OpenGL-based renderers require the use of the CUDA/GL interoperability APIs to copy (parts of) the frame buffer to device memory, either via PBO or framebuffer texture access. OptiX-based renderers output directly to CUDA device memory. Pre-rendered frames as used in our synthetic benchmarks are placed in device memory.

Analogously, the streaming client can connect to one or multiple servers simultaneously and request specific (sub) tiles. For each connection, the application creates a separate client pipeline instance. Similar to an OpenGL-based server, this step utilizes CUDA/GL interop to copy each tile into the mapped color texture of a framebuffer object or a target PBO. Using the frame number provided with each tile, initial client-side synchronization at the beginning of the streaming process is ensured by dropping outdated tiles based on the maximum frame number until all clients involved are synchronized. Since frame loss is prevented by TCP, display of subsequent frames is easily synchronized using barriers. In case of multiple distributed client processes, e.g., multiple nodes driving a tiled display wall, an MPI-based all-reduction is used to collectively determine the maximum frame number all clients must synchronize to.

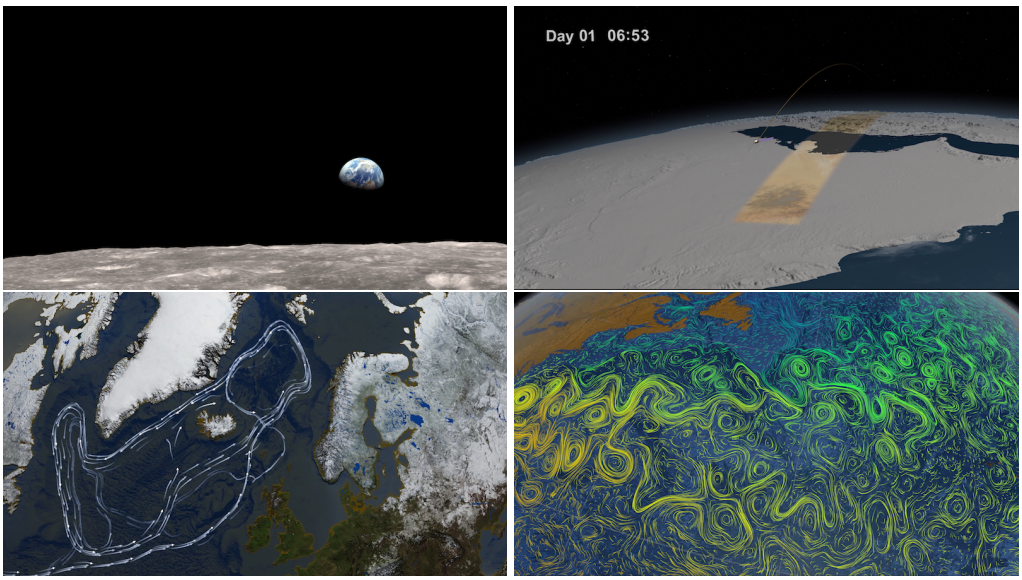


Fig. 6.2: NASA's *Synthesis 4K* video footage used for streaming benchmarks, representing typical content of scientific visualizations at various image complexities: *Space* (low), *Orbit* (medium), *Ice* (high), *Streamlines* (extreme).

6.5 Results

We have conducted comprehensive benchmarks to demonstrate the practicality of the presented multi-tile streaming approach and its impact on possible workflows and visualization scenarios.

As a stand-in for high-resolution scientific visualization renderings, we have selected the *Synthesis 4K* footage from NASA. Figure 6.2 illustrates scenes of varying complexity we have streamed from this source. Pre-rendered frames are extracted from the video, copied into device memory, and pushed into the multi-tile streaming server, as described in Sections 6.3 and 6.4. We have restricted our benchmarks to two commonly used resolutions: 3840x2160 and 2160x1200. The former is usually referred to as 4K and the latter is a typical resolution for current-generation VR devices, such as Oculus Rift and HTC Vive. We have used YUV 4:2:0 color format with a color depth of 8 bits per channel. The required CUDA-based conversion kernels between RGB and YUV have been profiled to be of negligible impact.

All benchmarks have been conducted on the Piz Daint supercomputer at the Swiss National Supercomputing Centre (CSCS). In our experiments we have used up to 512 GPU nodes for simultaneous multi-tile encoding and decoding. Additionally, at client-side we have benchmarked a multitude of scenarios on three locations for multi-tile decoding:

- Site A (ping 5 ms, 1x NVIDIA Quadro GP100)
- Site B (ping 25 ms, 2x NVIDIA Quadro GP100)
- Site C (ping 200-1000 ms, 4x NVIDIA Tesla P100)

Both the NVIDIA Tesla P100 and the Quadro GP100 are from the Pascal architecture family, featuring two independent hardware units for both encoding and decoding. In all benchmarks the hardware encoders were configured to use the *low latency - high quality* preset for both H.264 and HEVC. The target frame rate was set to 90 Hz, which always has to be considered in conjunction with a specific bitrate.

6.5.1 Codec Performance

The encoding quality of H.264 and HEVC can be configured to aim for a constant target bitrate. Intuitively, bandwidth requirements increase with frame complexity to sufficiently represent important details. We have used the structural similarity index (SSIM) [Wan+04] to measure and compare the encoding quality of H.264 and HEVC. Figure 6.3 shows the SSIM for the four test scenes at different bitrate settings. Higher bitrates improve reconstruction detail, where more complex scenes require higher bitrate settings to look acceptable. In general, HEVC outperforms H.264 by providing higher quality at the same bitrate. However, interestingly the

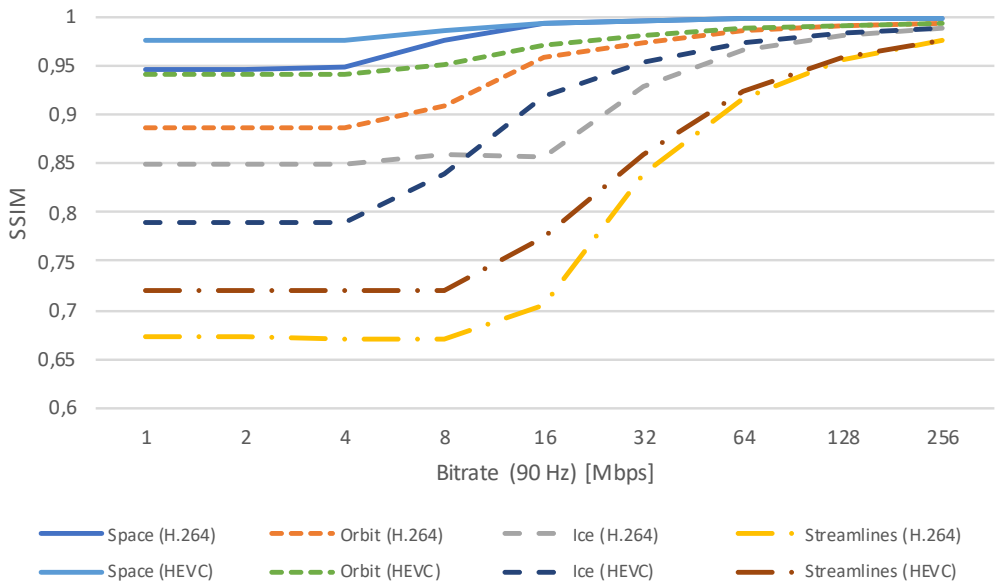


Fig. 6.3: Structural similarity (SSIM) index of the four test scenes at different encoder bitrates for H.264 and HEVC.

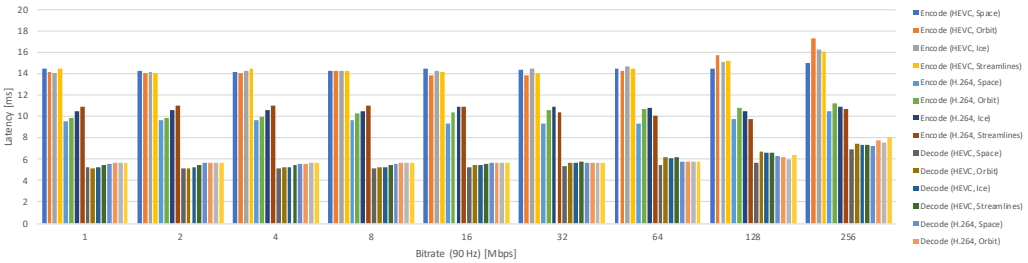


Fig. 6.4: Comparison of en-/decode latencies for the four test scenes at different bitrates for H.264 and HEVC at 4K resolution.

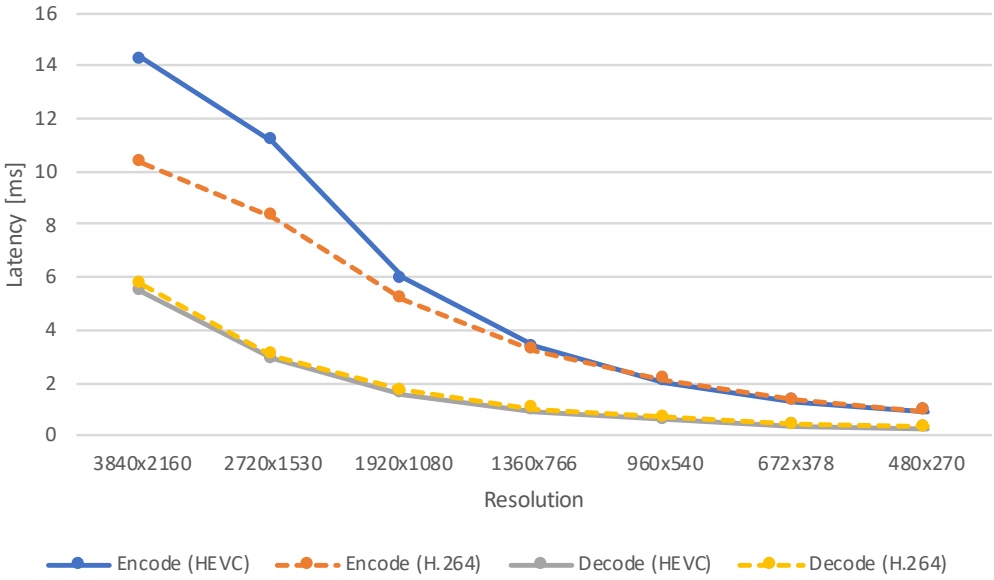


Fig. 6.5: Comparison of en-/decode latencies (averaged across test scenes) at different resolutions for H.264 and HEVC. Default bitrate for 4K at 90 Hz is 32/16 Mbps (H.264/HEVC), downscaled proportionally with pixel count.

high complexity *Ice* scene is slightly better reconstructed using H.264 for lower bitrates. Based on our visual experiments, we have chosen default bitrates of 32 Mbps for H.264 and 16 Mbps for HEVC in all following streaming benchmarks. If not specified otherwise, benchmarks are based on the *Ice* scene as a representative for high complexity.

The impact of image complexity on en-/decoding latencies is close to negligible for both H.264 and HEVC as illustrated in Figure 6.4. Also, the bitrate level only starts to affect latencies slightly at very high settings with increased memory bus pressure. In general, H.264 requires approx. 10 ms for encoding a full 4K frame on our test hardware, whereas HEVC needs 14 ms at the same bitrate. Decoding latency is similar for both codecs at 5-6 ms. When considering smaller tiles, overall latencies

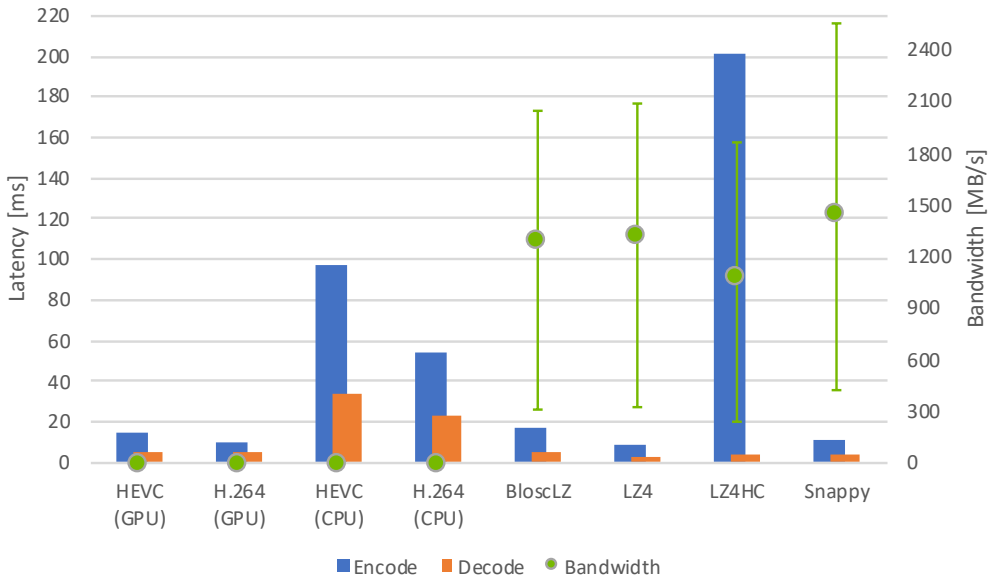


Fig. 6.6: Comparison of average en-/decode latencies and required bandwidths at 90 Hz for 4K using different GPU and CPU codecs. Bandwidth shows min/max ranges across test cases. H.264/HEVC configured to 32/16 Mbps at 90 Hz, others to maximum compression level.

decrease as expected and the gap between encoding latencies narrows, as shown in Figure 6.5.

We set out to demonstrate that hardware-accelerated progressive video encoding is superior to conventional single frame compression approaches for realtime streaming scenarios. Figure 6.6 shows the latency and required bandwidths at 4K resolution and a fixed frame rate of 90 Hz for several popular compression algorithms in comparison to the GPU. Both latency and bandwidth depict averages measured across the four test scenes. While the impact of image complexity is negligible for latency, high complexity content requires increased amounts of bandwidth. This is indicated by the error bars depicting the bandwidth range for the low and extreme complexity scenes. All CPU-based compressor benchmarks have been conducted on an Intel Core i7-6850K CPU at 3.6 GHz with hyper threading. The compressors have been set to use multi-threading and the maximum compression level possible. When considering latencies, GPU-accelerated video codecs are faster than their CPU-variant (using FFmpeg) by an order of magnitude. Interestingly, modern CPU-based compressors such as LZ4 or Snappy are comparable in latency to GPU video encoding, even with the additional PCIe bus transfer of the rendered frame between device and host memory included. The compressors Zlib and Zstd are not shown due to impractically large encoding latencies. In contrast, when considering

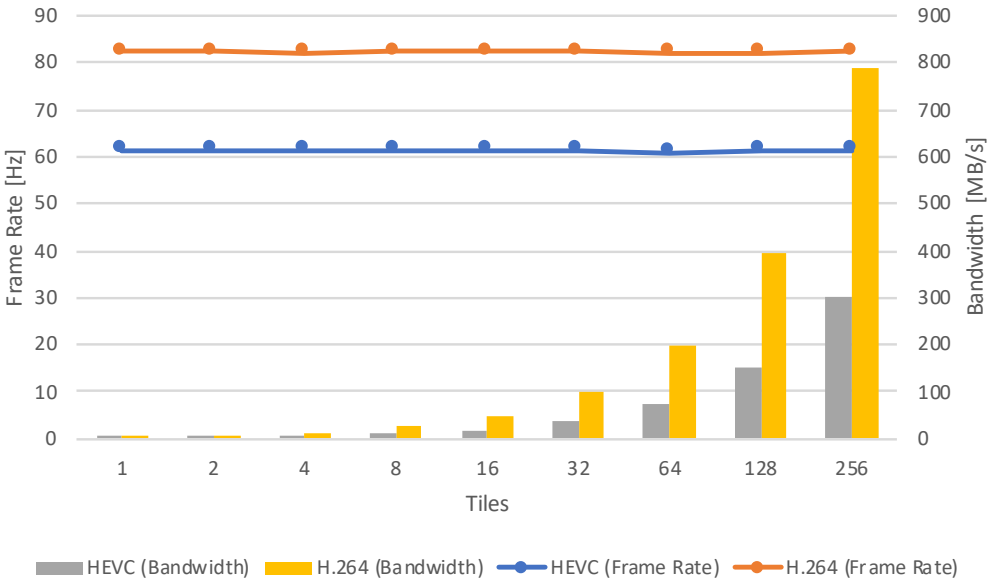


Fig. 6.7: Weak scaling (N:N) within Piz Daint: Client-side frame rates and bandwidths for up to 256 concurrent full 4K streams of the *Ice* video at 32/16 Mbps (H.264/HEVC).

required streaming bandwidth, progressive video encoding starts to shine due to its consideration of temporal coherence between frame. While for instance HEVC at 16 Mbps needs approximately 2 MB/s to stream the high complexity *Ice* scene at good quality, LZ4 requires an excessive bandwidth of more than 2 GB/s.

6.5.2 Full Tiles Streaming

Being able to drive tiled displays directly from a remote supercomputer enables cheaper infrastructure at the client side, as all the heavy lifting is done on the server side, and allows the rendering system to scale with the simulation, rather than having to scale a separate system for visualization.

This begs the question of how many simultaneous full resolution streams can be handled by GPUs and network to fulfill given requirements such as interactivity. For instance, how many GPUs are required to drive a particular tiled display or CAVE configuration at a certain target frame rate? To investigate such questions, we have conducted synthentic benchmarks for streaming within the Piz Daint supercomputer: up to 256 synchronized server nodes stream 4K frames from the complex *Ice* scene to up to 256 synchronized client nodes, which virtually display into an offscreen framebuffer via EGL. Figure 6.7 shows the mean frame rates and accumulated bandwidths for both H.264 and HEVC. Even for 256 concurrent streaming pipelines,

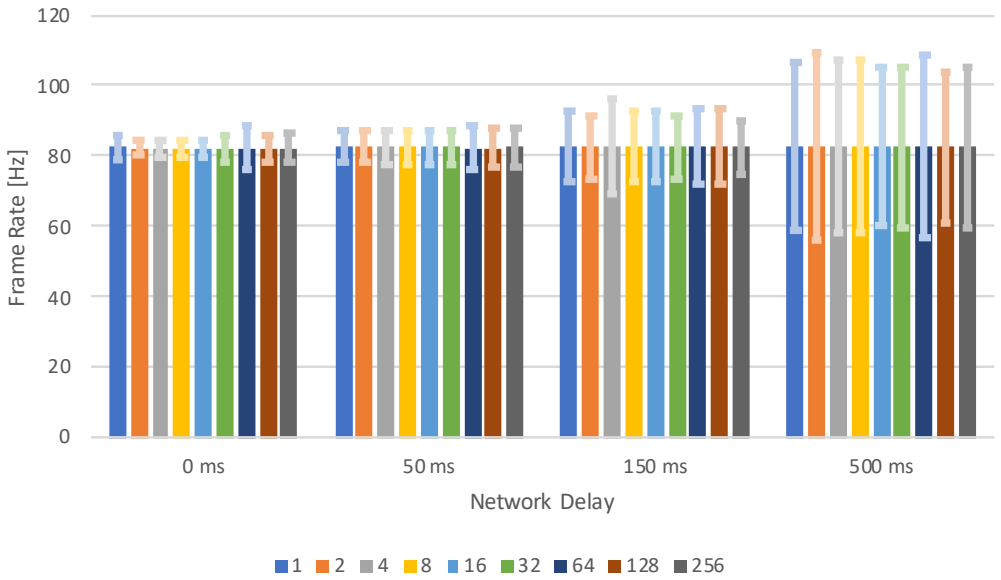


Fig. 6.8: Weak scaling (N:N) within Piz Daint: Mean frame rates and min/max ranges for different simulated network delays (+10% jitter), streaming up to 256 concurrent full 4K tiles of *Ice* sequence using H.264 at 32 Mbps.

the system manages to sustain stable frame rates of 80 Hz and 60 Hz, respectively. The required bandwidths increase linearly with node count. While this setup may appear overly optimistic at first, we expect this to be a plausible environment for possible improvements towards cheaper infrastructures: provided a sufficiently large low-latency link, a large-scale tiled display wall could be directly driven by super computer located either at the same site or possibly even a remote facility.

By delaying the sending of frames for a predefined amount of simulated network latency plus a random jitter within a given range, we have measured the impact of network latency on the multi-tile setup. A sufficiently large buffer was used to correctly simulate in-flight network packets without blocking the encoding pipeline. As each frame is assigned a random amount of delay within the specified window, frame burst patterns start to emerge, thereby affecting frame rate stability. Figure 6.8 shows the mean, minimum and maximum frame rates for different network delays with 10% jitter. While frame rate stability decreases with simulated network latency due to increased amounts of jitter, the overall frame rate remains in interactive magnitudes.

However, not only throughput, i.e., frame rate, but also the complete pipeline latency for a single frame from source to destination has to be considered for interactivity. Figure 6.9 illustrates the mean latencies of each pipeline stage across nodes without simulated network delay. Besides the expected encoding and decoding latencies,

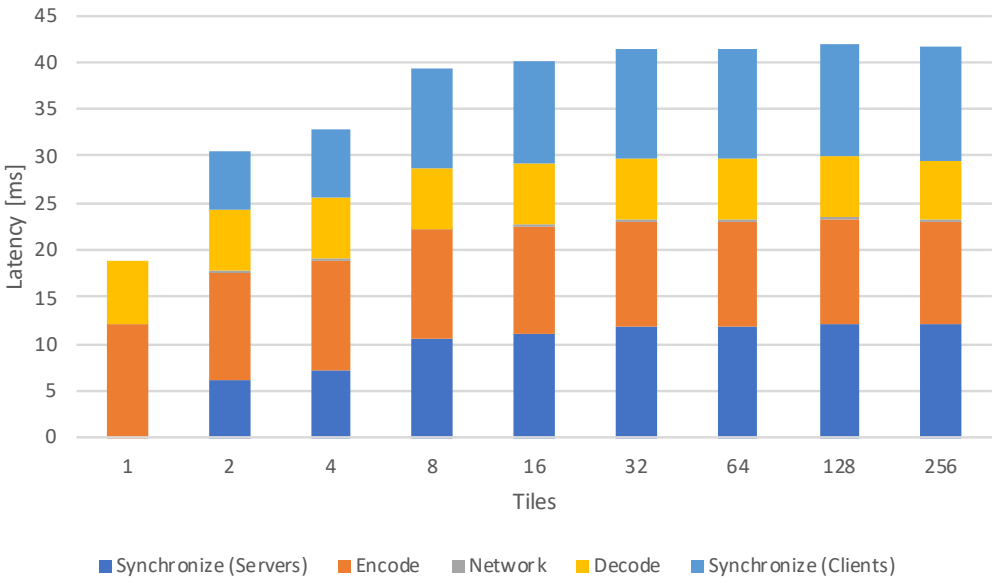


Fig. 6.9: Weak scaling (N:N) within Piz Daint: Pipeline latencies for full 4K streams of *Ice* using H.264 (32 Mbps).

synchronization across nodes at both server and client side contributes a considerable amount of additional latency, which increases with node count. With overall latencies of up to approx. 40 ms plus potential network delay this can still be considered highly interactive. While server-side synchronization can be eliminated depending on the scenario, client-side synchronization is obligatory for multi-node clients, e.g. for coherent display and buffer swap on a tiled display wall.

Since modern GPUs can drive multiple screens at 4K or higher resolutions, an interesting question is how many full streams can be handled by a single device at which frame rates, esp. when aiming for cheaper infrastructure. Figure 6.10 shows the achievable frame rates at client side for 4K resolution when streaming multiple concurrent full size tiles to Site A and Site B. A single NVIDIA Quadro GP100 card can decode up to two full 4K streams at approximately 80 Hz, using its two independent hardware decoding units. Given that this card features four display connectors, it is interesting to see that it can still handle four full 4K streams at 50 Hz. This performance capability enables high-resolution tiled displays with high tile counts driven by only a few GPUs for multi-tile decoding at interactive frame rates. As indicated by the shifted curve when using two GP100s for decoding, using hardware-accelerated multi-tile streaming is a scalable approach with the number of devices and is mainly limited by the available network bandwidth. As expected, the difference of network latency between Site A and Site B seems to be of negligible impact on frame rate.

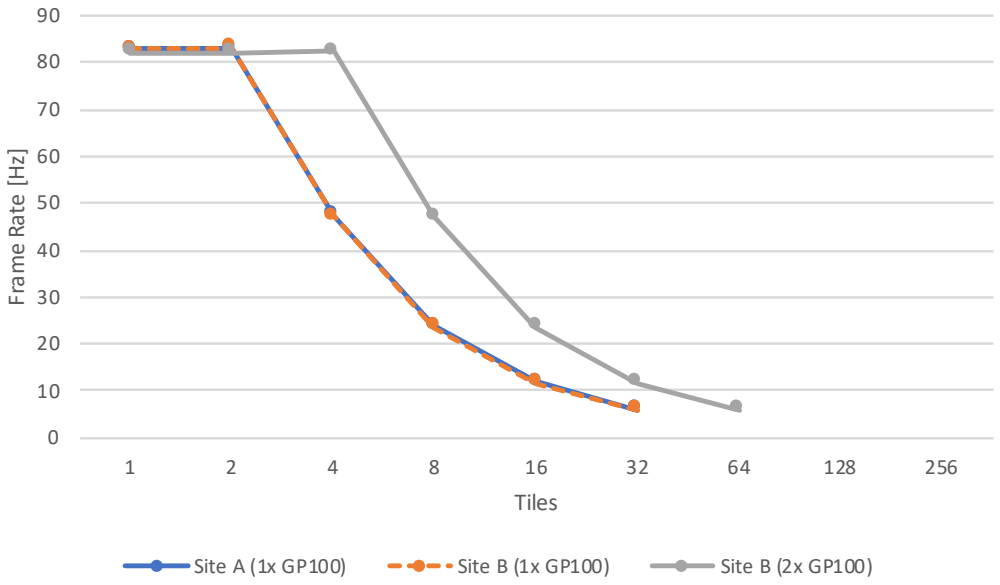


Fig. 6.10: Weak scaling (N:1): Client-side achieved frame rate for streaming full 4K frames of *Ice* sequence to Site A (blue) and Site B (orange, gray) using H.264 at 32 Mbps.

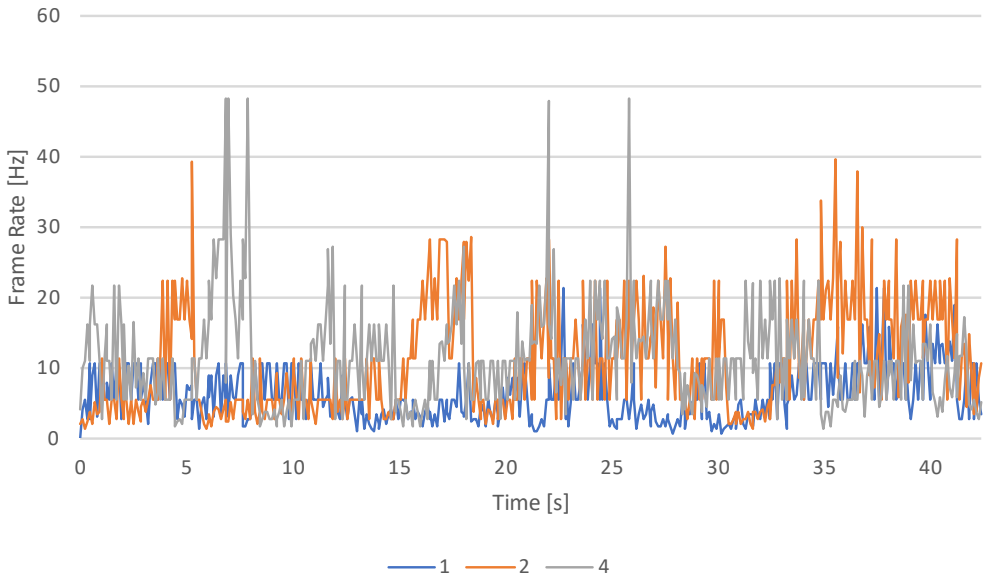


Fig. 6.11: Weak scaling (N:1): Streaming up to 4 full 4K frames of *Ice* sequence across Atlantic Ocean to Site C using HEVC at 16 Mbps. Frame rate sampled at 10 Hz.

To demonstrate the severe effect of a highly erratic link on frame rate, we have benchmarked multiple 4K streams from Piz Daint to Site C, California, USA. Figure 6.11 shows the effective client-side frame rate sampled at 10 Hz over 45 seconds.

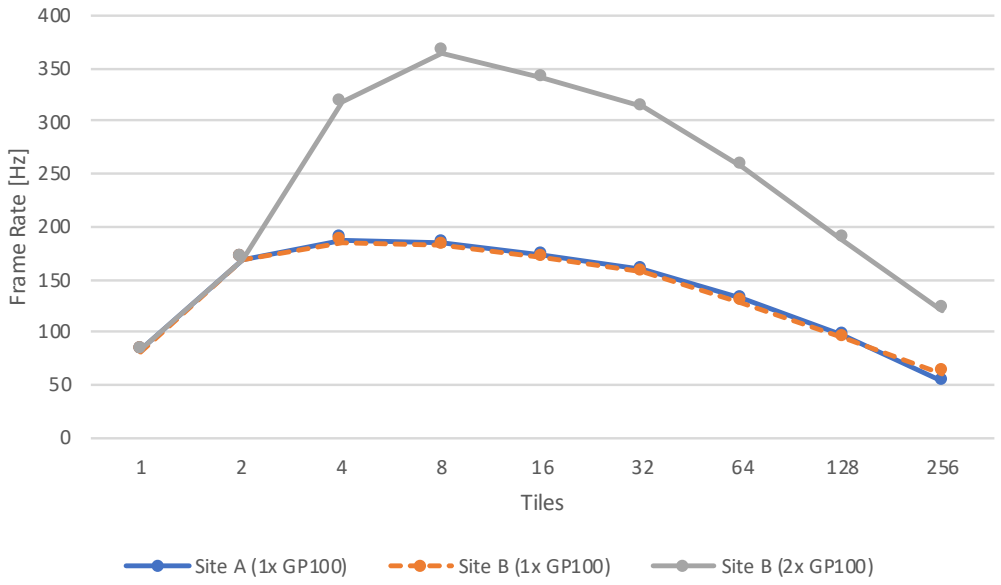
The presented graphs exemplify the unpredictability in such a setup, where due to frequent extreme latency spikes the overall frame rate drops significantly.

6.5.3 Strong Scaling / Sort-First Compositing

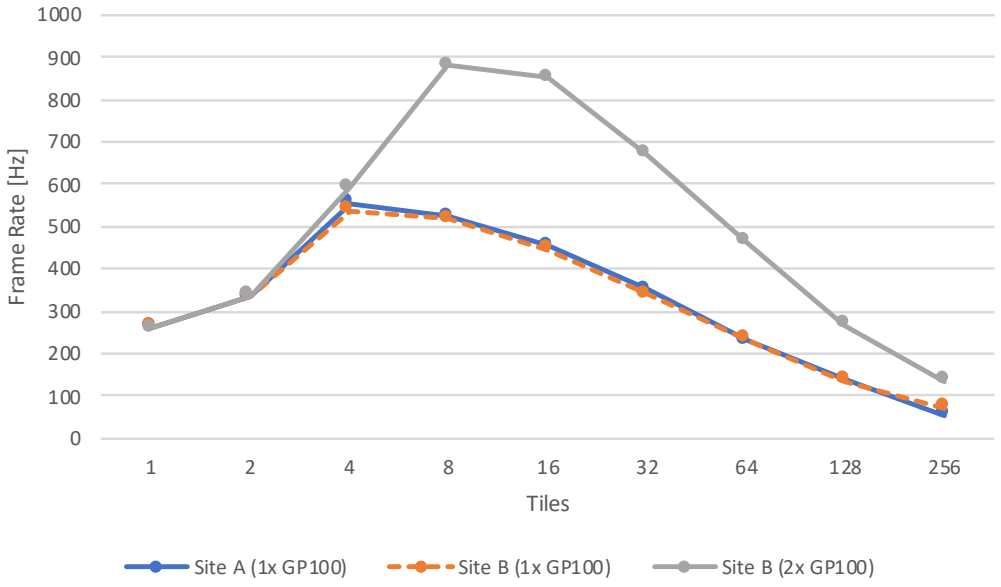
Strong scaling the remote rendering and delivery task opens up novel interactive uses of HPC systems. A plethora of server nodes enables expensive rendering solutions at interactive frame rates and low latency. This can be in support of improved perception in challenging visualization tasks, for instance by computing sophisticated global illumination models for complex geometries. In addition, the renewed interest in virtual reality (VR) gives rise to the question if streaming directly from the HPC system is a viable approach for time-critical VR scenarios. We have conducted several experiments to quantify the strong-scaled multi-tile streaming performance using up to 256 Piz Daint nodes streaming to a single client which performs sort-first compositing.

Figure 6.12 shows the achievable frame rates using strong scaling for 4K and VR resolution. Considering a single Quadro GP100 for decoding, frame rates up to 190 Hz are possible for 4K when utilizing 4 servers for encoding (Figure 6.12a). Full 4K encoding on a single node is limited by the encoder at approx. 80 Hz. With increased tile counts the maximum achievable frame rate slowly decreases due to increasing overhead from decode session multiplexing. The curves for Site A and Site B are almost identical for the 1x GP100 case, confirming that 20 ms of network latency has little effect on achievable throughput. Adding a second GP100 for decoding strongly amplifies the decoding capabilities at client side, enabling frame rates up to 365 Hz for 4K when utilizing 8 servers for encoding. Evidently, also the second card is subject to decode session overhead at high tile counts. Yet, strong scaled multi-tile 4K streaming from 256 servers is still possible at 120 Hz using two GP100s for decoding.

The graphs look similar for the reduced VR resolution (Figure 6.12b), generally showing much higher frame rates, which is expected due to reduced en-/decoding latency. A full 2160x1200 tile can be streamed from a single server node at 260 Hz, whereas streaming multiple tiles peaks at 520 Hz in the 1x GP100 case. Strongly scaling to 8 server nodes can achieve frame rates up to 900 Hz in the 2x GP100 case. Despite the decoding bottleneck at high tile counts, utilizing two GPUs for decoding of 256 tiles we can still achieve 135 Hz, comfortably above the suggested 90 Hz threshold for common VR applications. Again, the curves for Site A and Site B are almost identical.



(a) 4K (3840x2160)



(b) VR (2160x1200)

Fig. 6.12: Strong scaling (N:1): Client-side frame rate for streaming *Ice* sequence at 4K (left panel) and VR resolution (right panel) from up to 256 servers to Site A (blue), Site B (orange) and Site B with two GPUs per tile (gray) using H.264 (32 Mbps).

Figure 6.13 illustrates the strong scaled en-/decoding latencies for 4K frames in the 1x GP100 case and helps to further understand curve progression in Figure 6.12. Clearly, full 4K frame encoding (approx. 11 ms) is more than twice as costly as decoding (approx. 5 ms). In this context, note that the NVIDIA Tesla P100 and Quadro GP100 cards are very similar with respect to their en-/decoding hardware

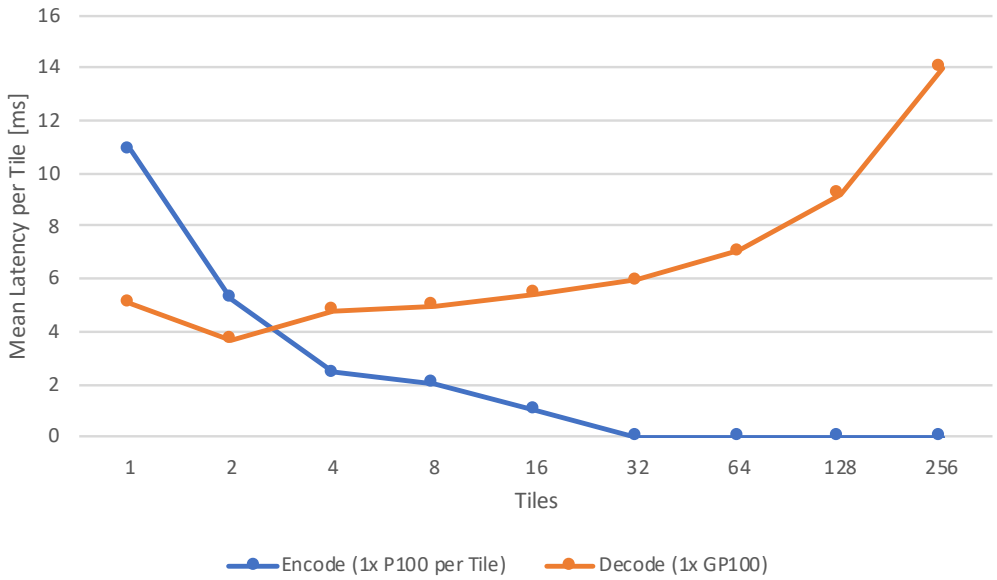


Fig. 6.13: Mean latency for encoding a 4K frame (blue) on the server side (strong-scaled, N:1) and decoding all tiles on the client side (Site B, orange) using H.264 (32 Mbps).

units. In the strong scaling scenario, encoding latencies drop as expected with increased tile counts due to the reduced pixel count per tile. Note that each tile is encoded by a different GPU in this case. At the same time, the mean decode latency per tile increases due to the high number of simultaneous decode sessions on the single decoding GPU. Based on Figure 6.13, the frame rate peak at 4 tiles in Figure 6.12 can thus be interpreted as the setup with minimal decode overhead where the frame rate is not limited by the encoder anymore.

The influence of encoding bitrate on frame rate is depicted in Figure 6.14. The results demonstrate the negative effect of high bitrate settings on the maximum achievable throughput, which drops from 185 Hz at low quality (4 Mbps @ 90 Hz) to 176 Hz at high quality (72 Mbps @ 90 Hz), and to 148 Hz at extreme quality (256 Mbps @ 90 Hz) for H.264. However, note that the extreme quality setting maximizes bandwidth utilization at approx. 46 MB/s, which coincides with the physical bandwidth limit measured in a separate network speed benchmark between Site B and Piz Daint. This finding is backed by the clamped nature of the high bitrate graph in comparison to the lower bitrate graphs, suggesting a theoretically greater achievable frame rate on high bandwidth connections. In practice, the particular bandwidth envelope at hand must be considered for stream design to prevent frame queueing effects from reducing interactivity.

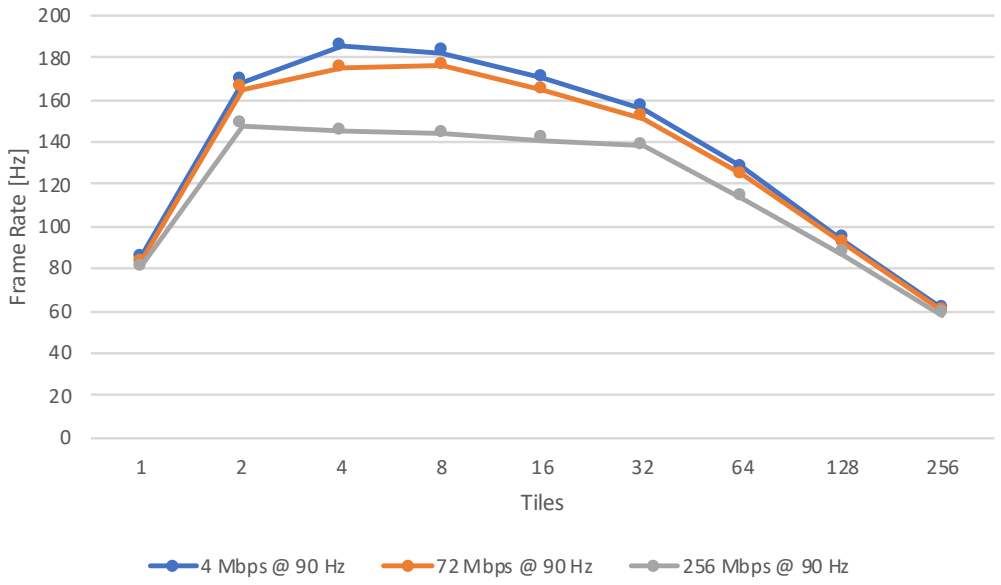


Fig. 6.14: Client-side frame rate for strong-scaled streaming of *Ice* 4K frames from up to 256 servers at different H.264 bitrates. Highest setting is limited by link bandwidth (gray).

In addition to the synthetic pre-rendered benchmarks, we have implemented a basic distributed path tracer, which builds on the path tracer sample from the OptiX SDK and uses screen space tiling in combination with replicated geometry. As a test scene we have used the *Groundwater* data set, which consists of approximately 8 million triangles. Previous benchmarks have used a regular tiling, but this tiling exhibits a strong load imbalance for the path tracing situation. We thus use an irregular tiling, an example of which is shown in Figure 6.15a, for benchmarks with the path tracer.

We have implemented simple two-dimensional recursive auto-tuning for this irregular tiling. The tiling is iteratively modified in a guided way until all tiles are subject to approx. the same rendering times. Starting with the regular tiling for a given number of tiles, e.g. 8x4 for 32 tiles, the algorithm works in two phases. First, only the vertical tiling is optimized by shifting the row heights until the differences between the rendering times of all rows are minimal. Once the optimal vertical tiling has been determined, the same procedure is applied to the horizontal tiling in each row recursively. In both dimensions, each iteration the shifting process determines the tile with maximum rendering time, decreases its size and increases the size of the fastest tile appropriately. After each iteration timings are re-evaluated and the tiling is shifted until convergence. The tiling of each iteration is checked against a recorded history for cycle detection. Convergence is then approximated

by selecting the tiling configuration from the different tilings in the detected cycle with the minimum total rendering latency. An example auto-tuned tiling for 32 tiles is depicted in Figure 6.15a, demonstrating smaller tiles due to increased computational requirements for path tracing in the vicinity of the emissive streamlines going through the stone geometry.

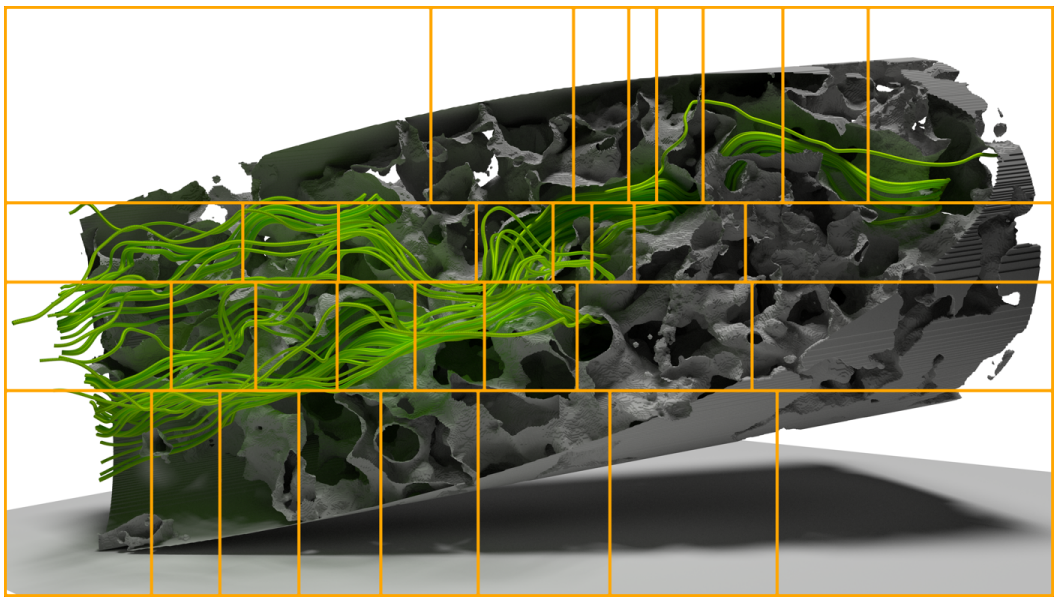
Figure 6.15b shows the achievable client-side frame rates when strong scaling the path tracer at 4K resolution up to 256 nodes streaming to Site A. Clearly, the auto-tuned tiling provides highly improved load balancing and thus reduced overall latencies in comparison to regular tiling. Starting with 0.5 Hz when streaming from a single node, the auto-tuned tiling strong-scales well: up to 14 Hz on 32 nodes, a scaling efficiency of almost 90%. Adding additional servers further increases the overall achievable frame rate up to 28 Hz on 256 nodes, yet with worse scaling efficiency. This can be explained by the high sensitivity of the approach towards load imbalances at high counts of tiny tiles, for which the path tracer in this scenario is a good example.

While effective load-balancing of distributed path tracers is out of the scope of this work, the presented path tracer benchmark shows how an otherwise slow-performing rendering application can easily be strong-scaled to interactive frame rates by using a remote HPC system to benefit from hardware-accelerated multi-tile streaming at low latencies.

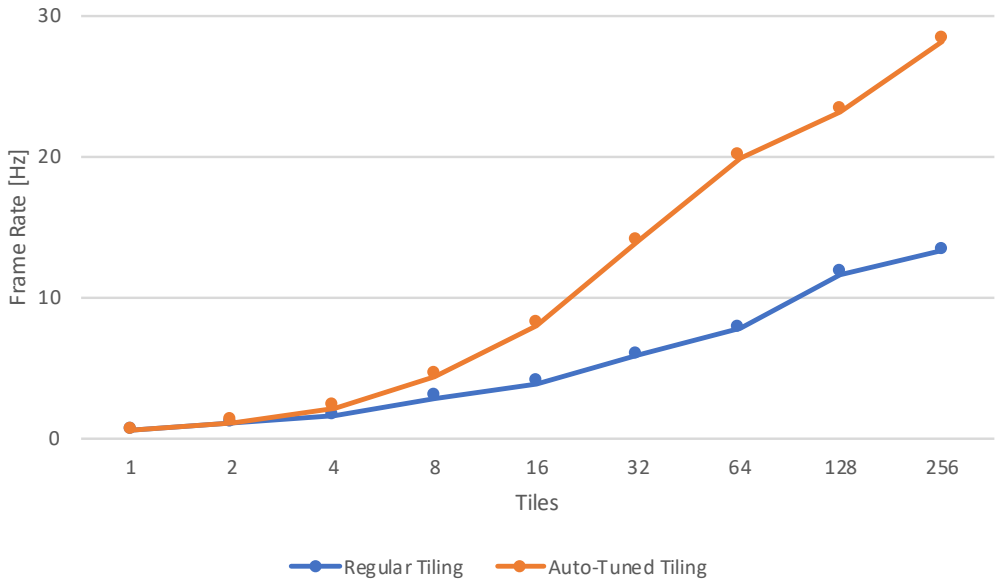
6.5.4 Interoperability

Although demonstrated on vendor hardware, the presented approach is compatible with any hardware or software following the H.264/HEVC specification, thus inherently providing interoperability. For instance, for verification we have successfully mixed GPU and FFmpeg-based compressors, or have directly written a sequence of raw H.264 to a file for playback with VLC media player.

Motivated by direct (potentially hardware-accelerated) support for video decoding in modern web browsers, we have implemented a prototypical streaming implementation based on WebSockets and JavaScript through Media Source Extensions. Figure 6.16 shows the *simpleGL* CUDA example running on Piz Daint and streamed into the browser. Aiming for minimal changes at application-side, we have created an EGL-based shim library as dynamic replacement for GLUT, which is normally used by that demo for display and interaction. Our shim library internally creates an EGL offscreen buffer for rendering, starts a WebSocket server for bidirectional



(a) Groundwater (Auto-Tuned Tiling)



(b) Strong Scaling

Fig. 6.15: Left: *Groundwater* scene used in the OptiX-based pathtracer for the 4K strong scaling rendering experiments with auto-tuned distribution of tile sizes. Data courtesy of TACC/FIU. Right: Client-side frame rate achieved for strong scaling of path traced image at 4K resolution on up to 256 nodes using fixed tile distribution (blue) and auto-tuned tile sizes (orange).

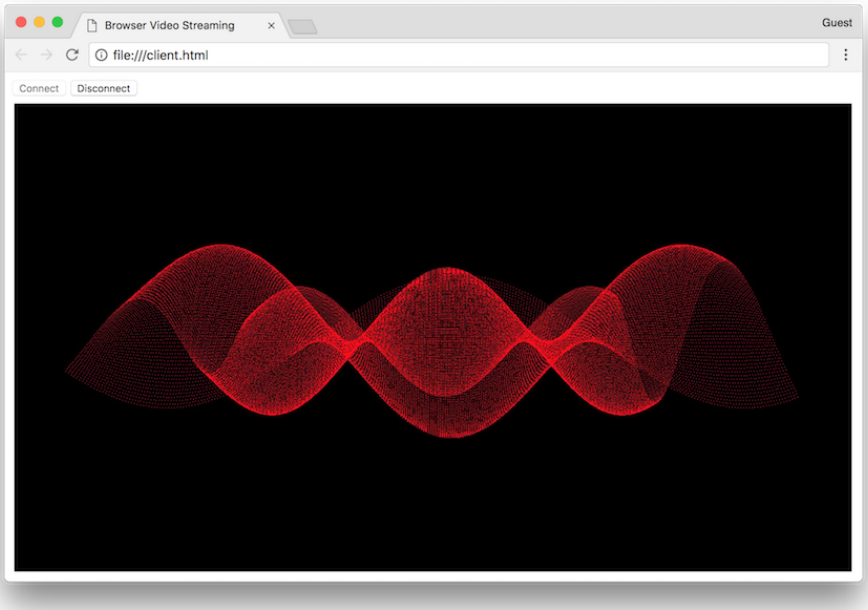


Fig. 6.16: Streaming unmodified CUDA *simpleGL* example from headless node to web browser using EGL-based shim library as GLUT replacement, WebSocket-based bidirectional communication (including interaction), on-the-fly MP4 wrapping of raw H.264 stream, and a JavaScript client.

communication with the browser client (including mouse interaction events), and replaces the buffer swap by encoding and streaming. The latter involves an additional on-the-fly wrapping of the raw H.264 stream into the MP4 container format, as required by the browser. Since the container specification only allows constant frame rates (as common for standard video content), the streaming application must take care not to let the browser buffer frames, thereby delaying interaction events and reducing responsiveness. In our experiments we have specified the container frame rate to 30 Hz and adjusted the encoder to drop frames as nececessary to maintain a constant browser-side buffer underrun.

6.6 Discussion

We have demonstrated how the specialized video encoding/decoding hardware on current and future generation GPUs can be harnessed for high performance multi-tile streaming at low latencies for scientific visualization. It is a promising approach to address modern challenges in remote HPC visualization, such as interactive workflows with complex rendering algorithms, supporting globally distributed

user bases and driving latency-sensitive display technologies at high resolutions. Using hardware-accelerated multi-tile streaming, traditional dedicated visualization clusters or workstations can be reduced to mere thin clients that leave the heavy lifting to the remote supercomputer.

Based on these encouraging results, we anticipate that many enhancements are possible and look forward to seeing multi-tile streaming as a technical foundation for future HPC visualization workflows. Similar to classic direct send, our approach will be limited in scalability at high tile counts. A hybrid tree-based/direct send compositing approach such as radix-k could alleviate this bottleneck. Hardware-accelerated sort-last compositing could be investigated based on depth buffer compression using video codecs. With high frame rates and low latencies indicating suitability for time-critical VR scenarios, the combined strong-scaled hardware power could be used for predictive rendering and streaming to remedy network latency.

7 Conclusion

In this thesis, several approaches based on ray casting have been developed and investigated with respect to a more efficient usage of heterogeneous high performance computers for visualization workloads, addressing a variety of challenges in modern HPC visualization, as discussed in Section 1.1.3.

Particle-based simulation models have assumed a significant role in the numerical computation of complex and highly dynamic transient flow and continuum mechanical problems. In this thesis, a novel direct raytracing scheme for on-the-fly free surface reconstruction has been presented, building upon the rich anisotropic kernel approach, which has been adapted and tuned to the surface definition of FPM-based fluid simulations. The improved anisotropic kernel-based surface definition incorporates automatic kernel scaling for variable smoothing lengths, provides intuitive visuals for isolated particles, and is easily parallelized. For this surface definition, a novel direct ray tracing scheme has been described. This on-demand two-pass iterative sampling algorithm intelligently reduces intersection candidates for both opaque and transparent surface rendering, provides optimization opportunities for secondary rays, and allows the dynamic mapping of particle attribute values on to the surface using arbitrary transfer functions. By reducing the number of candidate kernels evaluated to converge to the surface threshold, the approach runs in image space rather than object space complexity. Based on comprehensive benchmarks on different state-of-the-art hardware setups, including workstation, standard cluster and Xeon Phi accelerator systems, the versatile system has been shown to be suitable for both high quality and interactive desktop rendering, to scale reasonably well even with trivial parallelization and to render up to 170 million particles on 32 distributed compute nodes at close to interactive frame rates at 4K resolution with ambient occlusion. Many improvements to the presented approach are possible. Since FPM relies on local moving least squares interpolation, it by necessity incorporates nearest neighbor search structures with managed ghost particle information in distributed cluster mode, which could be reused in the preprocessing state of the pipeline. Furthermore, it seems natural to investigate the suitability of the system for production visualization, i.e., full path tracing, which is an important use case, e.g., in the automobile industry. Finally, comparing performance of the presented

implementation against a GPU implementation would be interesting to shed light on the relative strength of the differing architectures for the given use cases.

While high-fidelity simulation models on large-scale parallel computer systems can produce data at high computational throughput, modern architectural trade-offs make full persistent storage to the slow I/O subsystem prohibitively costly with respect to time. In this thesis, the feasibility and potential of combining in situ topological contour tree analysis and compact image-based data representation has been demonstrated to address this problem. Based on in situ contour tree analysis and simplification, a segmented representation of the scalar fields contained in the simulation data at every time step is obtained. A rendering of this segmentation is then generated describing all components visible in every pixel, and stored together with the simplified contour tree. These ingredients can then be used in post-analysis to flexibly select specific subsets of the segmentation, after further simplification if required. Several experiments have been conducted to quantify the I/O savings prossible from such an approach, showing significant reductions in storage requirements using topology-guided layered depth imaging, while preserving flexibility for explorative visualization and analysis. The presented technique highlights the feasibility and potential of the combination of topological analysis and image-based representation in large-scale in situ scenarios, and represents an effective and easy-to-control trade-off between storage overhead and visualization fidelity for large data visualization. While the results already show substantial reductions in output file size, especially for larger data sets, the contribution is intended as a baseline demonstration investigating possible advantages of such an approach for the visualization of large-scale data. Many enhancements and improvements of the approach are possible. Similar to Ahrens et al. [Ahr+14], the technique could be easily extended to generate a complete in situ image data base from multiple perspectives, which can be combined in the viewer application to enable a flexible 3D data exploration, or even be used for reconstruction purposes. Multivariate topological methods such as *Joint Contour Nets* [CD14] might be investigated to obtain improved segmentations. A shortcoming of the current implementation is frame-to-frame temporal consistency. Since contour trees are computed and decomposed independently at each time step, the resulting contours can vary noticeably between time steps depending on the chosen automatic simplification criteria, potentially undermining analysis due to the lack of frame-to-frame coherence. This issue could be addressed by incorporating feature tracking techniques into the branch selection and simplification process. The simplified contour tree stored with the compressed image files could be further annotated to compactly contain relevant properties of the original input data set, thereby improving the power and flexibility of the resulting visualization. However,

not only the contour tree, but also the fragments can be used to compactly store local information of the intersected segment useful for later visualization. Notably, the general idea of the presented concept is not restricted to regular scalar data and is easily applicable to different kinds of potentially more complex data structures, providing means for topological segmentation and intersection. More sophisticated compression schemes might be used to further increase the compactness of the layered depth images generated.

An increasingly heterogeneous system landscape in modern high performance computing requires the efficient and portable adaption of performant algorithms to diverse architectures. However, classic hybrid shared-memory/distributed systems are designed and tuned towards specific platforms, thus impeding development, usage and optimization of these approaches with respect to portability. In this thesis, a flexible parallel framework for distributed direct volume rendering has been demonstrated. Built upon a task-based dynamic runtime environment, it enables adaptable performance-oriented deployment on various platform configurations. The novel task-based definition aims to provide a flexibly tunable task granularity by subdividing in both image and data space, thus yielding a hybrid scheme between sort-first and sort-last compositing. Based on an asynchronous binary tree compositing scheme including optimizations such as empty space skipping and early ray termination, the technique has been shown to enable good scalability in combination with inherent dynamic load balancing. Comprehensive benchmarks with respect to task granularity and scaling have been conducted to verify the characteristics and potential of the novel task-based system design for high performance visualization. This contribution is intended as a baseline investigation of the applicability of task-based runtime environments to distributed scientific visualization. Future research includes distributed work stealing to be considered as an interesting approach to implement proper load balancing across node boundaries. Since each block is represented by an individual component in HPX's active global address space, the migration of blocks could be performed transparently with little to no modifications to the distributed compositing scheme. Distributed load balancing is especially important in an interactive setup with user-controlled camera navigation. In conjunction with distributed work stealing, a more sophisticated scheduling algorithm could also be used to improve task overlap and to ensure that the available I/O bandwidth is always kept saturated while executing rendering tasks as long as there are still blocks left to be loaded. Out of core handling could be used to support larger block counts on individual nodes. So far, the approach relies on the regular structure of blocks in various places. Support for unstructured data would involve complex enhancements to the distributed compositing scheme. In

general, performance benchmarks on larger core counts would be very interesting. The integration of additional accelerator cards such as Intel Xeon Phi or GPGPUs is theoretically easy in the spirit of task-based runtime environments, but in practice still technically challenging. Additional benchmarks for comparison against traditional non-task-based approaches are required to further characterize the benefits and drawbacks of the presented approach. Significant benefits are not expected for applications that already scale well on large machines using traditional data parallel approaches, especially if they are highly tuned and optimized towards a specific system or architecture. However, the major promising advantages of task-based designs lie in their portability to diverse and heterogeneous architectures, as well as the conceptionally more straightforward formulation of massively parallel programs. Besides distributed rendering, other load-sensitive techniques from scientific visualization such as topological methods or integral curve computations would certainly make promising candidates for task-based parallelization. Once enough task-based designs of standard visualization algorithms exist, their interplay and dependencies in a (complex) visualization pipeline could be studied. The task graph could be used for theoretical models and estimates about runtime, possible parallelization and scalability.

The growing use of distributed computing in computational sciences has put increased pressure on visualization and analysis techniques. In this context, a core challenge of HPC visualization is the physical separation of visualization resources and end-users. While GPUs are routinely used in remote rendering on GPU-accelerated heterogeneous supercomputers, a previously unexplored aspect has been these GPUs' special purpose video encoding/decoding hardware that can be used to solve the large-scale remoting challenge. The high performance and substantial bandwidth savings offered by such hardware has been demonstrated to enable a novel approach to the problems inherent in remote rendering, with impact on the workflows and visualization scenarios available. Using more tiles than previously thought reasonable, in this thesis, a distributed, low-latency multi-tile streaming system has been implemented, being able to sustain a stable 80 Hz when streaming up to 256 synchronized 3840x2160 tiles and achieve 365 Hz at 3840x2160 for sort-first compositing over the internet. Within a comprehensive case study, the impact of video compression and multi-tile streaming based on the H.264/HEVC codec family has been investigated in order to address a multitude of novel use cases, such as directly streaming content from a cluster to remote large-scale tiled displays at sufficiently high frame rates, or to strong-scale the rendering and delivery task by using video hardware to accelerate direct-send sort-first compositing. Using hardware-accelerated multi-tile streaming, traditional dedicated visualization clusters or workstations can be reduced to mere

thin clients that leave the heavy lifting to the remote supercomputer. Based on these encouraging results, we anticipate that many enhancements are possible and look forward to seeing multi-tile streaming as a technical foundation for future HPC visualization workflows. Similar to classic direct send, the presented approach will be limited in scalability at high tile counts. A hybrid tree-based/direct send compositing approach such as radix-k could alleviate this bottleneck. Hardware-accelerated sort-last compositing could be investigated based on depth buffer compression using video codecs. With high frame rates and low latencies indicating suitability for time-critical VR scenarios, the combined strong-scaled hardware power could be used for predictive rendering and streaming to remedy network latency.

Bibliography

[Acu+14] B. Acun, A. Gupta, N. Jain, et al. “Parallel Programming with Migratable Objects: Charm++ in Practice”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, Louisana: IEEE Press, 2014, pp. 647–658 (cit. on p. 81).

[Ada+07] B. Adams, M. Pauly, R. Keiser, and L. J. Guibas. “Adaptively Sampled Particle Fluids”. In: *ACM Trans. Graph.* 26.3 (July 2007) (cit. on p. 35).

[AP98] J. Ahrens and J. Painter. “Efficient Sort-Last Rendering Using Compression-Based Image Compositing”. In: *in Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization*. 1998, pp. 145–151 (cit. on p. 23).

[Ahr+14] J. Ahrens, S. Jourdain, P. O’Leary, et al. “An Image-based Approach to Extreme Scale in Situ Visualization and Analysis”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, Louisana: IEEE Press, 2014, pp. 424–434 (cit. on pp. 7, 58, 60, 74, 120).

[AGL05] J. Ahrens, B. Geveci, and C. Law. “ParaView: An End-User Tool for Large Data Visualization”. In: *The Visualization Handbook*. Ed. by C. D. Hansen and C. R. Johnson. 2005 (cit. on pp. 97, 99).

[Aki+12] G. Akinci, M. Ihmsen, N. Akinci, and M. Teschner. “Parallel Surface Reconstruction for Particle-Based Fluids”. In: *Comput. Graph. Forum* 31.6 (Sept. 2012), pp. 1797–1809 (cit. on p. 36).

[Ale+01] M. Alexa, J. Behr, D. Cohen-Or, et al. “Point Set Surfaces”. In: *Proceedings of the Conference on Visualization ’01*. VIS ’01. San Diego, California, 2001 (cit. on p. 35).

[AMD] AMD. *Advanced Media Framework*. `http://gpuopen.com/gaming-product/advanced-media-framework`. Accessed: 2018-01-20 (cit. on p. 98).

[Amd67] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485 (cit. on p. 2).

[Ams+15] J. Amstutz, C. Gribble, J. Günther, and I. Wald. “An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels”. In: *Journal of Computer Graphics Techniques (JCGT)* 4.4 (2015), pp. 72–88 (cit. on pp. 45, 49).

[ATT12] R. Ando, N. Thurey, and R. Tsuruno. “Preserving Fluid Sheets with Adaptively Sampled Anisotropic Particles”. In: *IEEE Trans. Visualization and Computer Graphics* 18.8 (Aug. 2012) (cit. on p. 36).

[AA07] S. Ayyub and D. Abramson. “GridRod: A Dynamic Runtime Scheduler for Grid Workflows”. In: *Proceedings of the 21st Annual International Conference on Supercomputing*. ICS ’07. Seattle, Washington: ACM, 2007, pp. 43–52 (cit. on p. 80).

[BPS97] C. L. Bajaj, V. Pascucci, and D. R. Schikore. “The Contour Spectrum”. In: *Proceedings of the 8th Conference on Visualization ’97*. VIS ’97. Phoenix, Arizona, USA: IEEE Computer Society Press, 1997, 167–ff. (Cit. on p. 59).

[Bar+06] A. W. Bargteil, T. G. Goktekin, J. F. O’Brien, and J. A. Strain. “A semi-Lagrangian Contouring Method for Fluid Simulation”. In: *ACM Trans. Graph.* 25.1 (Jan. 2006), pp. 19–38 (cit. on p. 35).

[Bau+12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. “Legion: Expressing locality and independence with logical regions”. In: *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 2012, pp. 1–11 (cit. on p. 81).

[Bek+12] K. Beketayev, G. H. Weber, D. Morozov, A. Abzhanov, and B. Hamann. “Geometry-preserving Topological Landscapes”. In: *Proceedings of the Workshop at SIGGRAPH Asia*. WASA ’12. Singapore, Singapore: ACM, 2012, pp. 155–160 (cit. on p. 59).

[BCH12] E. W. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. 1st. Chapman & Hall/CRC, 2012 (cit. on pp. 7, 20, 22).

[Bet+03] E. W. Bethel, G. Humphreys, B. Paul, and J. D. Brederson. “Sort-First, Distributed Memory Parallel Visualization and Rendering”. In: *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. PVG ’03. Washington, DC, USA, 2003, pp. 41–50 (cit. on p. 79).

[BHP15] J. Beyer, M. Hadwiger, and H. Pfister. “State-of-the-Art in GPU-Based Large-Scale Volume Visualization”. In: *Computer Graphics Forum* 34.8 (2015), pp. 13–37 (cit. on p. 80).

[BGB11] H. Bhatacharya, Y. Gao, and A. Bargteil. “A Level-set Method for Skinning Animated Particle Data”. In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’11. Vancouver, British Columbia, Canada, 2011, pp. 17–24 (cit. on p. 36).

[Bli82] J. F. Blinn. “A Generalization of Algebraic Surface Drawing”. In: *ACM Trans. Graph.* 1.3 (July 1982), pp. 235–256 (cit. on p. 35).

[BC11] S. Borkar and A. A. Chien. “The Future of Microprocessors”. In: *Commun. ACM* 54.5 (May 2011), pp. 67–77 (cit. on p. 1).

[BR63] R. L. Boyell and H. Ruston. “Hybrid Techniques for Real-time Radar Simulation”. In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. AFIPS ’63 (Fall). Las Vegas, Nevada: ACM, 1963, pp. 445–458 (cit. on p. 59).

[Bre+10] P.-T. Bremer, G. Weber, V. Pascucci, M. Day, and J. Bell. “Analyzing and Tracking Burning Structures in Lean Premixed Hydrogen Flames”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.2 (2010), pp. 1–1 (cit. on p. 60).

[Bre+11] P.-T. Bremer, G. Weber, J. Tierny, et al. “Interactive Exploration and Analysis of Large-Scale Simulations Using Topology-Based Data Segmentation”. In: *Visualization and Computer Graphics, IEEE Transactions on* 17.9 (2011), pp. 1307–1324 (cit. on p. 60).

[Bre+04] P.-T. Bremer, H. Edelsbrunner, B. Hamann, and V. Pascucci. “A topological hierarchy for functions on triangulated surfaces”. In: *IEEE Transactions on Visualization and Computer Graphics* 10 (2004), p. 2004 (cit. on p. 59).

[Car84] L. Carpenter. “The A -buffer, an Antialiased Hidden Surface Method”. In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 103–108 (cit. on pp. 61, 65).

[CD14] H. Carr and D. Duke. “Joint Contour Nets”. In: *Visualization and Computer Graphics, IEEE Transactions on* 20.8 (2014), pp. 1100–1113 (cit. on pp. 74, 120).

[CSP04] H. Carr, J. Snoeyink, and M. van de Panne. “Simplifying flexible isosurfaces using local geometric measures”. In: *Visualization, 2004. IEEE*. 2004, pp. 497–504 (cit. on p. 60).

[CS03] H. Carr and J. Snoeyink. “Path Seeds and Flexible Isosurfaces Using Topology for Exploratory Visualization”. In: *Proceedings of the Symposium on Data Visualisation 2003*. VISSYM ’03. Grenoble, France: Eurographics Association, 2003, pp. 49–58 (cit. on p. 59).

[CSA00] H. Carr, J. Snoeyink, and U. Axen. “Computing Contour Trees in All Dimensions”. In: *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’00. San Francisco, California, USA, 2000, pp. 918–926 (cit. on pp. 28, 59, 62).

[CSP10] H. Carr, J. Snoeyink, and M. van de Panne. “Flexible Isosurfaces: Simplifying and Displaying Scalar Topology Using the Contour Tree”. In: *Comput. Geom. Theory Appl.* 43.1 (Jan. 2010), pp. 42–58 (cit. on p. 60).

[CMF05] X. Cavin, C. Mion, and A. Filbois. “COTS cluster-based sort-last rendering: performance evaluation and pipelined implementation”. In: *VIS 05. IEEE Visualization, 2005.* 2005, pp. 111–118 (cit. on p. 21).

[CGM14] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 2014 (cit. on p. 3).

[Chi14] M. Chiesi. “Heterogeneous Multi-core Architectures for High Performance Computing”. PhD thesis. University of Bologna, 2014 (cit. on p. 1).

[Chi+10] H. Childs, D. Pugmire, S. Ahern, et al. “Extreme Scaling of Production Visualization Software on Diverse Architectures”. In: *Computer Graphics and Applications, IEEE* 30.3 (2010), pp. 22–31 (cit. on pp. 7, 57).

[CDM06] H. Childs, M. Duchaineau, and K.-L. Ma. “A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets”. In: *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV ’06. Braga, Portugal: Eurographics Association, 2006, pp. 153–161 (cit. on p. 80).

[Chi+12] H. Childs, E. Brugger, B. Whitlock, et al. “VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data”. In: *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Ed. by E. W. Bethel, H. Childs, and C. Hansen. Chapman & Hall, CRC Computational Science. Boca Raton, FL, USA: CRC Press/Francis–Taylor Group, Nov. 2012, pp. 357–372 (cit. on p. 97).

[CM93] B. Corrie and P. Mackerras. “Parallel volume rendering and data coherence”. In: *Proceedings of 1993 IEEE Parallel Rendering Symposium*. 1993, pp. 23–26, 106 (cit. on p. 79).

[CMP14] J. Cui, Z. Ma, and V. Popescu. “Animated Depth Images for Interactive Remote Visualization of Time-Varying Data Sets”. In: *IEEE Transactions on Visualization and Computer Graphics* 20.11 (2014), pp. 1474–1489 (cit. on p. 97).

[Dem+12] D. Demir, K. Beketayev, G. H. Weber, et al. “Topology Exploration with Hierarchical Landscapes”. In: *Proceedings of the Workshop at SIGGRAPH Asia*. WASA ’12. Singapore, Singapore: ACM, 2012, pp. 147–154 (cit. on p. 59).

[Din+09] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. “Scalable Work Stealing”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon: ACM, 2009, 53:1–53:11 (cit. on p. 78).

[DTS01] H. Q. Dinh, G. Turk, and G. Slabaugh. “Reconstructing surfaces using anisotropic basis functions”. In: *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*. Vol. 2. 2001, 606–613 vol.2 (cit. on p. 36).

[DLP02] J. J. Dongarra, P. Luszczek, and A. Petitet. *The LINPACK benchmark: Past, present, and future*. 2002 (cit. on p. 5).

[DN09] H. Doraiswamy and V. Natarajan. “Efficient Algorithms for Computing Reeb Graphs”. In: *Comput. Geom. Theory Appl.* 42.6-7 (Aug. 2009), pp. 606–616 (cit. on p. 59).

[DCH88] R. A. Drebin, L. Carpenter, and P. Hanrahan. “Volume Rendering”. In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’88. New York, NY, USA: ACM, 1988, pp. 65–74 (cit. on pp. 17, 79).

[DG15] A. Dubey and D. T. Graves. “A Design Proposal for a Next Generation Scientific Software Framework”. In: *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*. Ed. by S. Hunold, A. Costan, D. Giménez, et al. Cham: Springer International Publishing, 2015, pp. 221–232 (cit. on pp. 77, 80, 81).

[Ede+03] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. “Morse-smale Complexes for Piecewise Linear 3-manifolds”. In: *Proc. of the Nineteenth Annual Symposium on Computational Geometry*. SCG ’03. San Diego, California, USA: ACM, 2003, pp. 361–370 (cit. on p. 59).

[EMP09] S. Eilemann, M. Makhinya, and R. Pajarola. “Equalizer: A Scalable Parallel Rendering Framework”. In: *IEEE Transactions on Visualization and Computer Graphics* 15.3 (2009), pp. 436–452 (cit. on p. 97).

[EP07] S. Eilemann and R. Pajarola. “Direct Send Compositing for Parallel Sort-last Rendering”. In: *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV ’07. Lugano, Switzerland, 2007, pp. 29–36 (cit. on pp. 22, 80, 98).

[ESE00] K. Engel, O. Sommer, and T. Ertl. “A Framework for Interactive Hardware Accelerated Remote 3D-Visualization”. In: *Data Visualization 2000: Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization in Amsterdam, The Netherlands, May 29–30, 2000*. Ed. by W. C. de Leeuw and R. van Liere. Vienna: Springer Vienna, 2000, pp. 167–177 (cit. on p. 97).

[ELF05] D. Enright, F. Losasso, and R. Fedkiw. “A Fast and Accurate semi-Lagrangian Particle Level Set Method”. In: *Comput. Struct.* 83.6-7 (Feb. 2005), pp. 479–490 (cit. on p. 35).

[Fan+14] J. Fang, H. Sips, L. Zhang, et al. “Test-driving Intel Xeon Phi”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. Dublin, Ireland, 2014, pp. 137–148 (cit. on p. 4).

[Feb+13] A. Febretti, A. Nishimoto, T. Thigpen, et al. *CAVE2: a hybrid reality environment for immersive simulation and information analysis*. 2013 (cit. on p. 99).

[FE10] P. Fechteler and P. Eisert. “Accelerated video encoding using render context information”. In: *Proceedings of the International Conference on Image Processing, ICIP 2010, September 26-29, Hong Kong, China*. IEEE, 2010, pp. 2033–2036 (cit. on p. 97).

[Fog+10] T. Fogal, H. Childs, S. Shankar, et al. “Large Data Visualization on Distributed Memory Multi-GPU Clusters”. In: *High Performance Graphics*. Ed. by M. Doggett, S. Laine, and W. Hunt. HPG ’10. Saarbrucken, Germany, 2010, pp. 57–66 (cit. on p. 80).

[FAW10] R. Fraedrich, S. Auer, and R. Westermann. “Efficient High-Quality Volume Rendering of SPH Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (2010) (cit. on p. 35).

[Fre+14] S. Frey, F. Sadlo, K.-L. Ma, and T. Ertl. “Interactive Progressive Visualization with Space-Time Error Control”. In: *Visualization and Computer Graphics, IEEE Transactions on* 20.12 (2014), pp. 2397–2406 (cit. on p. 60).

[Fuj+00] I. Fujishiro, Y. Takeshima, T. Azuma, and S. Takahashi. “Volume data mining using 3D field topology analysis”. In: *Computer Graphics and Applications, IEEE* 20.5 (2000), pp. 46–51 (cit. on p. 59).

[Gil+13] T. Gilmanov, M. Anderson, M. Brodowicz, and T. Sterling. “Application characteristics of many-tasking execution models”. In: *The 19th International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, USA, 2013 (cit. on p. 81).

[Gos+10] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola. “Interactive SPH Simulation and Rendering on the GPU”. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’10. Madrid, Spain, 2010, pp. 55–64 (cit. on p. 35).

[Gyu+05] A. Gyulassy, V. Natarajan, V. Pascucci, P. T. Bremer, and B. Hamann. “Topology-based Simplification for Feature Extraction from 3D Scalar Fields”. In: *Proceedings of IEEE Conference on Visualization*. Minneapolis, MN, 2005 (cit. on p. 59).

[Hai+11] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra. “Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures”. In: *Concurr. Comput. : Pract. Exper.* 24.3 (Mar. 2011), pp. 305–321 (cit. on p. 80).

[Her+08] M Hereld, E Olson, M. E. Papka, and T. D. Uram. “Streaming visualization for collaborative environments”. In: *Journal of Physics: Conference Series* 125.1 (2008) (cit. on p. 97).

[Hie+05] D. Hietel, M. Junk, J. Kuhnert, and S. Tiwari. “Meshless Methods for Conservation Laws”. In: *Analysis and Numerics for Conservation Laws* (2005), pp. 339–362 (cit. on p. 36).

[HN81] C. Hirt and B. Nichols. “Volume of fluid (VOF) method for the dynamics of free boundaries”. In: *Journal of Computational Physics* 39.1 (1981), pp. 201 –225 (cit. on p. 35).

[HOK16] H. Hochstetter, J. Orthmann, and A. Kolb. “Adaptive Sampling for On-the-fly Ray Casting of Particle-based Fluids”. In: *Proceedings of High Performance Graphics*. HPG ’16. Dublin, Ireland, 2016, pp. 129–138 (cit. on p. 35).

[HBC12] M. Howison, E. W. Bethel, and H. Childs. “Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), pp. 17–29 (cit. on pp. 7, 77, 80).

[Hsu93] W. M. Hsu. “Segmented ray casting for data parallel volume rendering”. In: *Proceedings of 1993 IEEE Parallel Rendering Symposium*. 1993, pp. 7–14 (cit. on p. 22).

[Hum+10] M. Hummel, C. Garth, B. Hamann, H. Hagen, and K. I. Joy. “IRIS: Illustrative Rendering for Integral Surfaces”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (Nov. 2010), pp. 1319–1328 (cit. on p. 68).

[Int] Intel. *Intel Media SDK*. `https://software.intel.com/en-us/media-sdk`. Accessed: 2018-01-20 (cit. on p. 98).

[JH95] J. Jeong and F. Hussain. “On the Identification of a Vortex”. In: *Journal of Fluid Mechanics* 285 (1995), pp. 69–94 (cit. on p. 68).

[Jia+16] J. Jiang, T. Fogal, C. Woolley, and P. Messmer. “A Lightweight H.264-based Hardware Accelerated Image Compression Library”. In: *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. 2016, pp. 99–100 (cit. on p. 97).

[Kai+14] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proc. of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS ’14. Eugene, OR, USA: ACM, 2014, 6:1–6:11 (cit. on pp. 78, 81, 85).

[Kaj86] J. T. Kajiya. “The Rendering Equation”. In: *Computer Graphics*. 1986, pp. 143–150 (cit. on p. 15).

[KV03] A. Kalaiah and A. Varshney. “Statistical Point Geometry”. In: *Proc. of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP ’03. Aachen, Germany, 2003, pp. 107–115 (cit. on p. 36).

[Kal+08] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. “Programming Petascale Applications with Charm++ and AMPI”. In: *Petascale Computing: Algorithms and Applications*. Ed. by D. Bader. Chapman & Hall / CRC Press, 2008, pp. 421–441 (cit. on p. 80).

[Ken+10] W. Kendall, T. Peterka, J. Huang, H.-W. Shen, and R. Ross. “Accelerating and Benchmarking Radix-k Image Compositing at Large Scale”. In: *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*. EG PGV’10. Norrköping, Sweden, 2010, pp. 101–110 (cit. on pp. 24, 80, 98).

[Kno+14] A. Knoll, I. Wald, P. Navratil, et al. “RBF Volume Ray Casting on Multicore and Manycore CPUs”. In: *Proceedings of the 16th Eurographics Conference on Visualization*. EuroVis ’14. Swansea, Wales, United Kingdom: Eurographics Association, 2014, pp. 71–80 (cit. on p. 80).

[Kre+97] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. “Contour Trees and Small Seed Sets for Isosurface Traversal”. In: *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*. SCG ’97. Nice, France: ACM, 1997, pp. 212–220 (cit. on p. 59).

[LGS09] W. J. van der Laan, S. Green, and M. Sainz. “Screen Space Fluid Rendering with Curvature Flow”. In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D ’09. Boston, Massachusetts, 2009, pp. 91–98 (cit. on pp. 35, 36).

[Lal+09] H. G. Lalgudi, M. W. Marcellin, A. Bilgin, H. Oh, and M. S. Nadar. “View Compensated Compression of Volume Rendered Images for Remote Visualization”. In: *IEEE Transactions on Image Processing* 18.7 (2009), pp. 1501–1511 (cit. on p. 97).

[LS07] F. Lamberti and A. Sanna. “A Streaming-Based Solution for Remote Visualization of 3D Graphics on Mobile Devices”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.2 (2007), pp. 247–260 (cit. on p. 97).

[Lan+14] A. G. Landge, V. Pascucci, A. Gyulassy, et al. “In-situ Feature Extraction of Large Scale Combustion Simulations Using Segmented Merge Trees”. In: *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, Louisana, 2014, pp. 1020–1031 (cit. on p. 60).

[LMM17] N. Leaf, B. Miller, and K. L. Ma. “In situ video encoding of floating-point volume data using special-purpose hardware for a posteriori rendering and analysis”. In: *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. 2017, pp. 64–73 (cit. on p. 97).

[Lev90] M. Levoy. “Efficient Ray Tracing of Volume Data”. In: *ACM Trans. Graph.* 9.3 (July 1990), pp. 245–261 (cit. on pp. 17, 79).

[Li+14] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. “Algorithmic Improvements for Fast Concurrent Cuckoo Hashing”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands, 2014 (cit. on p. 49).

[LL03] G. R. Liu and M. B. Liu. *Smoothed particle hydrodynamics: a meshfree particle method*. 2003 (cit. on p. 36).

[LLL06] M. B. Liu, G. R. Liu, and K. Y. Lam. “Adaptive smoothed particle hydrodynamics for high strain hydrodynamics with material strength”. In: *Shock Waves* 15.1 (2006), pp. 21–29 (cit. on p. 36).

[LC87] W. E. Lorensen and H. E. Cline. “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169 (cit. on pp. 16, 36).

[Ma+94] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. “Parallel volume rendering using binary-swap compositing”. In: *IEEE Computer Graphics and Applications* 14.4 (1994), pp. 59–68 (cit. on pp. 22, 23, 80, 98).

[MM13] M. Macklin and M. Müller. “Position Based Fluids”. In: *ACM Trans. Graph.* 32.4 (July 2013), 104:1–104:12 (cit. on p. 36).

[MMD06] S. Marchesin, C. Mongenet, and J.-M. Dischler. “Dynamic Load Balancing for Parallel Volume Rendering”. In: *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV ’06. Braga, Portugal: Eurographics Association, 2006, pp. 43–50 (cit. on p. 80).

[Mar+14] T. Marrinan, J. Aurisano, A. Nishimoto, et al. “SAGE2: A new approach for data intensive collaboration using Scalable Resolution Shared Displays”. In: *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2014, pp. 177–186 (cit. on p. 97).

[Mau+12] M. Maule, J. Comba, R. Torchelsen, and R. Bastos. “Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer”. In: *Graphics, Patterns and Images (SIBGRAPI), 2012 25th SIBGRAPI Conference on*. 2012, pp. 134–141 (cit. on p. 65).

[Mol+94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. “A Sorting Classification of Parallel Rendering”. In: *IEEE Comput. Graph. Appl.* 14.4 (July 1994), pp. 23–32 (cit. on pp. 20, 21, 79).

[Mol+11] B. Moloney, M. Ament, D. Weiskopf, and T. Moller. “Sort-First Parallel Volume Rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (Aug. 2011), pp. 1164–1177 (cit. on p. 79).

[Mor+11] K. Moreland, W. Kendall, T. Peterka, and J. Huang. “An image compositing solution at scale”. In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2011, pp. 1–10 (cit. on pp. 21, 24, 98).

[MSE06] C. Müller, M. Strengert, and T. Ertl. “Optimized Volume Raycasting for Graphics-hardware-based Cluster Systems”. In: *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV ’06. Braga, Portugal: Eurographics Association, 2006, pp. 59–67 (cit. on p. 79).

[MÖ9] M. Müller. “Fast and Robust Tracking of Fluid Surfaces”. In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’09. New Orleans, Louisiana, 2009 (cit. on p. 35).

[MCG03] M. Müller, D. Charypar, and M. Gross. “Particle-based Fluid Simulation for Interactive Applications”. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’03. San Diego, California, 2003, pp. 154–159 (cit. on p. 35).

[Nav+07] P. A. Navrátil, D. S. Fussell, C. Lin, and W. R. Mark. *Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization*. Tech. rep. 2007 (cit. on p. 80).

[Neu94] U. Neumann. “Communication costs for parallel volume-rendering algorithms”. In: *IEEE Computer Graphics and Applications* 14.4 (1994), pp. 49–58 (cit. on p. 22).

[NJ16] J. M. Noguera and J. R. Jiménez. “Mobile Volume Rendering: Past, Present and Future”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.2 (2016), pp. 1164–1178 (cit. on p. 97).

[NPS12] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland. “Graph-Based Software Design for Managing Complexity and Enabling Concurrency in Multiphysics PDE Software”. In: *ACM Trans. Math. Softw.* 39.1 (Nov. 2012), 1:1–1:21 (cit. on p. 80).

[NVI16] NVIDIA. *NVIDIA Tesla P100 – The Most Advanced Data Center Accelerator Ever Built*. Tech. rep. 2016 (cit. on p. 4).

[NVI] NVIDIA. *NVIDIA Video Codec SDK*. `https://developer.nvidia.com/nvidia-video-codec-sdk`. Accessed: 2018-01-20 (cit. on pp. 97, 99).

[OF03] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. 2003 (cit. on p. 35).

[Owe+98] J. M. Owen, J. V. Villumsen, P. R. Shapiro, and H. Martel. “Adaptive Smoothed Particle Hydrodynamics: Methodology. II.” In: *The Astrophysical Journal Supplement Series* 116.2 (1998), p. 155 (cit. on p. 36).

[PCM04] V. Pascucci and K. Cole-McLaughlin. “Parallel Computation of the Topology of Level Sets”. English. In: *Algorithmica* 38.1 (2004), pp. 249–268 (cit. on p. 59).

[PCMS04] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli. “Multi-resolution computation and presentation of contour trees”. In: *Proc. IASTED Conference on Visualization, Imaging, and Image Processing*. 2004, pp. 452–290 (cit. on pp. 60, 62).

[Péb+16] P. P. Pébay, J. C. Bennett, D. S. Hollman, et al. “Towards Asynchronous Many-Task in Situ Data Analysis Using Legion”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. 2016, pp. 1033–1037 (cit. on p. 81).

[Pet+09] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. “A Configurable Algorithm for Parallel Image-compositing Applications”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon: ACM, 2009, 4:1–4:10 (cit. on pp. 24, 80, 98).

[Pet+08] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. “Parallel Volume Rendering on the IBM Blue Gene/P”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by J. M. Favre and K.-L. Ma. The Eurographics Association, 2008 (cit. on p. 80).

[Pra10] A. Pranckevičius. *Compact Normal Storage for Small G-Buffers*. `http://aras-p.info/texts/CompactNormalStorage.html`. Accessed: 2018-02-22. 2010 (cit. on p. 67).

[Pre+03] S. Premžoe, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker. “Particle-Based Simulation of Fluids”. In: *Computer Graphics Forum* 22.3 (2003), pp. 401–410 (cit. on p. 35).

[Pug+09] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. “Scalable Computation of Streamlines on Very Large Datasets”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. Portland, Oregon: ACM, 2009, 16:1–16:12 (cit. on pp. 7, 77).

[Rei+14] F. Reichl, M. G. Chajdas, J. Schneider, and R. Westermann. “Interactive Rendering of Giga-particle Fluid Simulations”. In: *Proc. High Performance Graphics*. Lyon, France, 2014 (cit. on p. 35).

[Rot82] S. D. Roth. “Ray casting for modeling solids”. In: *Computer Graphics and Image Processing* 18.2 (1982), pp. 109 –144 (cit. on p. 14).

[Sab88] P. Sabella. “A Rendering Algorithm for Visualizing 3D Scalar Fields”. In: *SIGGRAPH Comput. Graph.* 22.4 (June 1988), pp. 51–58 (cit. on p. 17).

[SJ00] G. Schaufler and H. W. Jensen. “Ray Tracing Point Sampled Geometry”. In: *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*. 2000, pp. 319–328 (cit. on p. 35).

[SFS05] C. E. Scheidegger, S. Fleishman, and C. T. Silva. “Triangulating Point Set Surfaces with Bounded Error”. In: *Proceedings of the Third Eurographics Symposium on Geometry Processing*. SGP ’05. 2005 (cit. on p. 35).

[SA13] M. Segal and K. Akeley. *OpenGL 4.3 Core Profile Specification*. `https://www.opengl.org/registry/`. Accessed: 2018-02-22. 2013 (cit. on p. 65).

[Sod+16] A. Sodani, R. Gramunt, J. Corbal, et al. “Knights Landing: Second-Generation Intel Xeon Phi Product”. In: *IEEE Micro* 36.2 (2016), pp. 34–46 (cit. on p. 4).

[SSP07] B. Solenthaler, J. Schläfli, and R. Pajarola. “A Unified Particle Model for Fluid & Solid Interactions: Research Articles”. In: *Comput. Animat. Virtual Worlds* 18.1 (Feb. 2007) (cit. on p. 35).

[Ste07] J. K. Steehler. “Understanding Moore’s Law—Four Decades of Innovation”. In: *Journal of Chemical Education* 84.8 (2007), p. 1278 (cit. on p. 1).

[Ste+03] S. Stegmaier, J. Diepstraten, M. Weiler, and T. Ertl. “Widening the remote visualization bottleneck”. In: *3rd International Symposium on Image and Signal Processing and Analysis, 2003. ISPA 2003. Proceedings of the*. Vol. 1. 2003, 174–179 Vol.1 (cit. on p. 97).

[SAB17] T. Sterling, M. Anderson, and M. Brodowicz. *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann, 2017 (cit. on pp. 2, 5, 6).

[Ste+14] T. Sterling, D. Kogler, M. Anderson, and M. Brodowicz. “SLOWER: A performance model for Exascale computing”. In: *Supercomputing Frontiers and Innovations* 1.2 (2014) (cit. on p. 6).

[Sto+03] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett. “SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering”. In: *Proc. of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. PVG ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 6– (cit. on pp. 21, 23, 80, 98).

[Suf07] K. Suffern. *Ray Tracing from the Ground Up*. Natick, MA, USA: A. K. Peters, Ltd., 2007 (cit. on pp. 13, 15).

[Sul+12] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand. “Overview of the High Efficiency Video Coding (HEVC) Standard”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012), pp. 1649–1668 (cit. on pp. 28, 98).

[TFT05] S. Takahashi, I. Fujishiro, and Y. Takeshima. *Interval volume decomposer: a topological approach to volume traversal*. 2005 (cit. on p. 59).

[TTF04] S. Takahashi, Y. Takeshima, and I. Fujishiro. “Topological Volume Skeletonization and Its Application to Transfer Function Design”. In: *Graph. Models* 66.1 (Jan. 2004), pp. 24–49 (cit. on p. 59).

[Tak+05] Y. Takeshima, S. Takahashi, I. Fujishiro, and G. M. Nielson. “Introducing Topological Attributes for Objective-based Visualization of Simulated Datasets”. In: *Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics*. VG’05. New York, 2005, pp. 137–145 (cit. on p. 59).

[TIH03] A. Takeuchi, F. Ino, and K. Hagihara. “An Improved Binary-swap Compositing for Sort-last Parallel Rendering on Distributed Memory Multiprocessors”. In: *Parallel Comput.* 29.11-12 (Nov. 2003), pp. 1745–1762 (cit. on p. 23).

[Tho+11] D. Thompson, J. Levine, J. Bennett, et al. “Analysis of large-scale scalar data using hixels”. In: *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. 2011, pp. 23–30 (cit. on p. 60).

[TCM10] A. Tikhonova, C. Correa, and K.-L. Ma. “Visualization by Proxy: A Novel Framework for Deferred Interaction with Volume Data”. In: *Visualization and Computer Graphics, IEEE Transactions on* 16.6 (2010), pp. 1551–1559 (cit. on p. 60).

[Tiw+07] S. Tiwari, S. Antonov, D. Hietel, et al. “A Meshfree Method for Simulations of Interactions between Fluids and Flexible Structures”. In: *Meshfree Methods for Partial Differential Equations III* (2007) (cit. on p. 36).

[Top] *Top500 List*. `https://www.top500.org`. Accessed: 2018-03-02 (cit. on pp. 3–5).

[TO02] G. Turk and J. F. O’Brien. “Modelling with Implicit Surfaces That Interpolate”. In: *ACM Trans. Graph.* 21.4 (Oct. 2002) (cit. on p. 35).

[Wal+17] I. Wald, G. Johnson, J. Amstutz, et al. “OSPRay - A CPU Ray Tracing Framework for Scientific Visualization”. In: *IEEE Transactions on Visualization & Computer Graphics* 23.1 (2017), pp. 931–940 (cit. on pp. 49, 86).

[Wan+04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612 (cit. on p. 103).

[WBP07] G. Weber, P.-T. Bremer, and V. Pascucci. “Topological Landscapes: A Terrain Metaphor for Scientific Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Nov. 2007), pp. 1416–1423 (cit. on p. 59).

[WSH03] G. H. Weber, G. Scheuermann, and B. Hamann. “Detecting Critical Regions in Scalar Fields”. In: *Proceedings of the Symposium on Data Visualisation 2003*. VISSYM ’03. Grenoble, France: Eurographics Association, 2003, pp. 85–94 (cit. on p. 59).

[Web+07] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann. “Topology-Controlled Volume Rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.2 (Mar. 2007), pp. 330–341 (cit. on pp. 60, 64).

[Wie+03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. “Overview of the H.264/AVC video coding standard”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003), pp. 560–576 (cit. on pp. 28, 97).

[Yan+10] J. C. Yang, J. Hensley, H. Grün, and N. Thibieroz. “Real-time Concurrent Linked List Construction on the GPU”. In: *Proceedings of the 21st Eurographics Conference on Rendering*. EGSR’10. Saarbrücken, Germany: Eurographics Association, 2010, pp. 1297–1304 (cit. on p. 65).

[YWM08] H. Yu, C. Wang, and K.-L. Ma. “Massively Parallel Volume Rendering Using 2-3 Swap Image Compositing”. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC ’08. Austin, Texas: IEEE Press, 2008, 48:1–48:11 (cit. on pp. 23, 80, 98).

[YT13] J. Yu and G. Turk. “Reconstructing Surfaces of Particle-based Fluids Using Anisotropic Kernels”. In: *ACM Trans. Graph.* 32.1 (Feb. 2013), 5:1–5:12 (cit. on pp. 34, 36, 38, 39, 41, 43).

[Yu+12] J. Yu, C. Wojtan, G. Turk, and C. Yap. “Explicit Mesh Surfaces for Particle Based Fluids”. In: *Comput. Graph. Forum* 31.2pt4 (May 2012), pp. 815–824 (cit. on p. 36).

[ZB05] Y. Zhu and R. Bridson. “Animating Sand As a Fluid”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 965–972 (cit. on p. 35).

List of Figures

List of Tables

List of Listings

Publications

The results presented in this thesis have been published as follows:

T. Biedert[1], C. Garth[1]
Contour Tree Depth Images For Large Data Visualization
In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), Cagliari, Italy, 2015

T. Biedert[1], C. Garth[1]
In Situ Large Data Visualization using Layered Depth Images
In Proceedings of Young Researchers Symposium (YRS), Kaiserslautern, Germany, 2016

T. Biedert[1], K. Werner[1], B. Hentschel[2], C. Garth[1]
A Task-Based Parallel Rendering Component For Large-Scale Visualization Applications
In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), Barcelona, Spain, 2017

T. Biedert[1], J.-T. Sohns[1], S. Schröder[3], J. Amstutz[4], I. Wald[4], C. Garth[1]
Direct Raytracing of Particle-based Fluid Surfaces Using Anisotropic Kernels (Honorable Mention)
In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), Brno, Czech Republic, 2018

T. Biedert[1], P. Messmer[5], T. Fogal[5], C. Garth[1]
Hardware-Accelerated Multi-Tile Streaming for Realtime Remote Visualization (Best Paper)
In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), Brno, Czech Republic, 2018

[1]Technische Universität Kaiserslautern
[2]RWTH Aachen University
[3]Fraunhofer ITWM
[4]Intel Corporation
[5]NVIDIA Corporation

Curriculum Vitae: Tim Biedert

Education

2014	**Master of Science (M.Sc.)** in Computer Science Technische Universität Kaiserslautern, Germany Thesis: *Vortex Visualization of Flapping Wing Insect Flight in Virtual Reality Environments*
2012	**Bachelor of Science (B.Sc.)** in Computer Science Technische Universität Kaiserslautern, Germany Thesis: *FPM Post Processing: Real-Time Ray Tracing of Point Set Surfaces in OptiX*
2009	**Abitur** Elisabeth-Langgässer-Gymnasium Alzey, Germany

Experience

since 2018	**NVIDIA GmbH**, Würselen, Germany Developer Technology Senior Scientific Visualization Developer Technology Engineer
2014 - 2018	**Technische Universität Kaiserslautern**, Germany Scientific Visualization Lab / Computational Topology Group Research assistant / Ph.D. candidate
2017	**NVIDIA Switzerland AG**, Zurich, Switzerland Developer Technology Internship
2010 - 2017	**Fraunhofer Institute for Industrial Mathematics ITWM**, Kaiserslautern, Germany Dept. of Transport Processes Research assistant
2014	**Wright State University**, Dayton, Ohio, USA Advanced Visual Data Analysis Group Visiting researcher