A Novel Approach to Interactive, Distributed Visualization and Simulation on Hybrid Cluster Systems

Dissertation

von

Stefan Lietsch

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, Juli 2008

Erstellt am:
Paderborn Center for Parallel Computing - PC^2
Fürstenallee 11
33102 Paderborn

Datum der mündlichen Prüfung:
13. Oktober 2008

Gutachter:
Prof. Dr. Odej Kao, Technische Universität Berlin
Prof. Dr. Marco Platzner, Universität Paderborn

Für meinen Vater und meine Familie,
auf dass diese Arbeit neue Kraft spendet.

Für Anna,
die immer für mich da ist.

Acknowledgment

I would like to thank:

- my advisor Prof. Odej Kao for supporting and inspiring me for many years. I have benefited a lot from his experience and always found a sympathetic ear for my problems and questions.
- Prof. Marco Platzner for giving me valuable advise and for accepting to serve as a co-examiner and head of the commission for my dissertation without hesitation.
- Dr. Jens Simon for helping me to find my place in the scientific world and for many insightful discussions and technical advises.
- Dr. Jan Berssenbrügge, Dr. Christoph Laroque, Henning Zabel and all other members of the VisSim project group for being very inspiring partners and for helping to develop and publish many ideas of this thesis.
- Prof. Burckhard Monien, Prof. Holger Karl and my colleagues at the PC^2 for providing me with an excellent research environment. A special thank goes to the technical staff for helping me out even on the most short-termed and unorthodox requests.
- Dr. Oliver Marquardt for his invaluable support in technical and programming related questions.
- the many students that helped to realize our ideas.
- my family, Anna and her family and all my friends for their continued support and encouragement.

Abstract

The introduction of hybrid cluster systems, which consist of several heterogeneous groups of homogenous nodes (e.g. computing nodes, visualization nodes, hardware-accelerated nodes) marks a paradigm shift in the usage of cluster systems. For the first time, the powerful hardware is open to applications other than classical high performance computing (HPC) applications, such as number crunching and massively parallel simulations. Various new fields of science and industry could use it as a powerful new tool for simulation and visualization. Interactive simulations and their visualization as well as Virtual Reality applications are areas where HPC on hybrid cluster systems can bring significant benefit in many aspects, as for example realism, multiple comparative simulations, multiuser-support etc.. But, to enable and to later-on ease the utilization of the complex cluster systems for such simulations, tools to orchestrate and access these resources are needed. Existing applications should be quickly able to run on and to benefit from the new hardware with only little effort and changes in source code. Currently there are only few systems that (partially) fulfill these tasks. Most of the existing ones for traditional HPC applications lack important features to provide the desired interactivity and flexibility.

This thesis addresses two main aspects of this problem and introduces concepts for the computational steering (CS) and the remote visualization (RV) of interactive simulations and Virtual Reality (IS/VR) applications on hybrid cluster systems. The main focus is to conserve the interactivity and user-integration of these applications and, at the same time, dramatically extend their features by harvesting the power of the hybrid cluster architecture. Thus, the main contribution of this thesis is the introduction of two new subsystems, one for the steering and orchestration of the application's distributed components (CS) and one for the remote access to the interactive graphical applications running on the cluster (RV). The concept for the CS framework bases on three new models for the steering of IS/VR applications that significantly differ from the model used for traditional CS. Besides the original idea that a running simulation is observed and controlled by a connected visualization, many parts of traditional CS needed to be redesigned and adapted to the needs of IS/VR. Especially interactivity and flexibility play a big role during the conceptual design of the system. The proposed framework and its implementation is tested and evaluated by realizing distributed versions of two existing IS/VR applications and is further showing that these applications greatly gain on flexibility, performance and quality for the users. Thereafter, the introduction of a framework for RV gives the users graphical access to the applications running on the cluster system. Again, the developed system takes the idea of traditional RV (e.g. for remote administration purposes) and transfers it to the domain of IS/VR, where interactivity and high quality are of highest importance. To achieve these goals, the developed system, as the first of its kind, makes use of power-

ful graphics cards for image compression. Until now, these GPUs (Graphics Processing Unit), built into most of the existing hybrid clusters, were almost exclusively used for the rendering of the results of HPC simulations. But, only recently these processors have become available for general purpose computing through the introduction of new and universal APIs. By different benchmarks it is shown that the performance of the remote visualization is thereby improved in many cases. Additionally, the framework is able to use the flexibility that clusters provide, to allow the remote access to multiple, simultaneously running applications on the cluster by multiple users.

All in all, a novel approach to effectively use hybrid clusters for interactive simulations and VR applications is presented, and the findings are manifested by exemplary applications and various benchmarks.

Zusammenfassung

Das Aufkommen hybrider Cluster-Systeme, welche aus heterogenen Gruppen homogener Knoten (z.B. Rechenknoten, Visualisierungsknoten oder hardwarebeschleunigten Knoten) bestehen, markiert eine Trendwende in der generellen Einsetzbarkeit von Cluster Systemen. Zum ersten Mal stehen diese leistungsstarken Ressourcen, neben typischen High Performance Computing (HPC) Anwendungen wie Number Crunching und massiv-parallelen Simulationen, auch neuen Anwendungen zur Verfügung. Unterschiedlichste Anwender aus Forschung und Wirtschaft können diese Systeme nun als mächtiges Werkzeug zur Simulation und Visualisierung ihrer Probleme nutzen. Interaktive Simulationen und Virtual Reality Anwendungen, sind Bereiche, denen High Performance Computing auf hybriden Cluster Systemen große Vorteile und neue Möglichkeiten bringen. Beispiele hierfür sind ein verbesserter Grad an Realismus, die Möglichkeit mehrere Simulationsläufe gleichzeitig durchführen und vergleichen zu können oder dynamische Mehrbenutzer-Szenarien zu realisieren. Allerdings werden, um die Benutzung solcher komplexer Cluster-Systeme für diese Anwendungen zu ermöglichen und zu erleichtern, Werkzeuge benötigt, welche die flexible Kopplung der verteilten Komponenten und den Zugriff auf das Gesamtsystem realisieren. Existierende Anwendungen sollten schnell und einfach auf der neuen Hardware lauffähig sein und möglichst ohne große Veränderungen in Code und Design die Vorteile davon nutzen können. Zurzeit gibt es nur sehr wenige Systeme, welche diese Aufgaben auch nur teilweise erfüllen. Bestehenden Werkzeuge für traditionelles HPC mangelt es an wichtigen Eigenschaften um die benötigte Interaktivität und Flexibilität gewährleisten zu können.

Die vorliegende Arbeit beschäftigt sich deshalb mit den zwei wichtigsten Aspekten dieses Problems und führt Konzepte für das Computational Steering (CS) und der entfernten Visualisierung (RV) von interaktiven Simulationen und Virtual Reality (IS/VR) Anwendungen auf hybriden Cluster Systemen ein. Das Hauptaugenmerk liegt dabei auf der Erhaltung der Interaktivität und Benutzerintegration der Anwendungen, bei gleichzeitiger Erweiterung und Verbesserung deren Fähigkeiten durch die Ausnutzung der Kapazitäten von hybriden Clustern. Daraus ergab sich die Idee zu den wichtigen Beiträgen dieser Arbeit: Die Entwicklung und Umsetzung zweier neuartiger Middlewares für die Steuerung und Instrumentation der verteilten Komponenten (CS) und für den entfernten Zugang (RV) zu den interaktiven, grafischen Anwendungen auf dem hybriden Cluster System. Das Konzept für das CS Framework basiert auf drei neuen Modellen für die Steuerung der IS/VR Anwendungen, welche deutlich vom Modell für das Steuern traditioneller HPC Simulationen abweicht. Außer der ursprünglichen Idee, dass eine laufende Simulation durch eine angeschlossene Visualisierung beobachtet und gesteuert werden kann, müssen viele Teile des traditionellen CS neu entworfen und an die Anforderungen von IS/VR angepasst werden. Speziell

Interaktivität und Flexibilität spielten eine große Rolle bei der Konzeptionierung des Systems. Durch die beispielhafte Realisierung von verteilten Versionen zweier existierender IS/VR Anwendungen wird das Konzept des entwickelten Frameworks und dessen Umsetzung getestet und evaluiert. Es wird außerdem in Szenarien gezeigt, dass die Beispielanwendungen an Performance, Flexibilität und Qualität für die Benutzer hinzugewinnen. Das zweite Framework (RV für IS/VR) gibt dem Benutzer grafischen Zugriff zu den Anwendungen, die auf dem Cluster laufen. Erneut wurden die traditionellen Ansätze der entfernten Visualisierung (z.B. Administration entfernter Server) genutzt und an die Anforderungen der neuen Anwendungen angepasst, bei denen Interaktivität und Qualität der Visualisierung eine große Rolle spielen. Um diesen Anforderungen gerecht zu werden nutzt das entwickelte Framework, als erstes seiner Art, die leistungsstarken Grafikkarten, welche in vielen hybriden Clustern verbaut sind, zur Bildkompression. Bis jetzt wurden diese Hardware-Ressourcen fast ausschließlich zur grafischen Darstellung der Ergebnisse der HPC-Simulation genutzt. Erst seit kurzem können diese leistungsstarken Prozessoren durch die Einführung von universellen APIs auch für generelle Zwecke genutzt werden. Durch verschiedene Messungen wird gezeigt, dass durch die Nutzung der GPUs (Graphics Processing Unit) zur Bildkompression eine Performancesteigerung in vielen Bereichen möglich ist. Dies wiederum dient zur Erhaltung der wichtigen Interaktivität auch über längere Strecken und Netzwerken mit geringen Bandbreiten hinweg. Zusätzlich nutzt das entwickelte Framework auch die Flexibilität, welche hybride Cluster bieten, um beispielsweise den entfernten Zugang zu mehreren gleichzeitig laufenden Simulationen für mehrere Nutzer zu ermöglichen.

Zusammenfassend beschreibt die vorliegende Arbeit einen neuen Ansatz zur praktischen Nutzung hybrider Cluster Systeme für interaktive Simulationen und Virtual Reality Anwendungen und untermauert die Ergebnisse durch Messungen und die Umsetzung beispielhafter Anwendungen.

Contents

List of Figures

1

Introduction

For the last 40 years, Moore's Law [55] has determined the performance of modern computer systems. Since 1965, the prediction that the number of elements which can be crammed on a chip of the same size, and thereby the achievable computational power, doubles every 1-2 years, has come true. The performance gains were almost completely achieved by shrinking the microprocessors and thus being able to increase the internal clock frequency. Hence, it was not difficult for software developers to increase the speed of their software: It simply ran faster on a faster machine. However, over the last years it showed that a natural border concerning the clock frequency of microprocessors has been reached: Clock speeds beyond 4 GHz resulted in unpredictable byte shifts, uncoolable chip surfaces and other major errors. That is why other ways of increasing performance had to be found to keep up with Moore's Law. The most promising solution is to fit multiple homogenous or heterogeneous cores into one processor (so called multi- or many-cores) and connect them through fast memory or busses. Those sophisticated architectures can easily fulfill Moore's Law as long as the amount of included cores can be doubled every 18 months. That seems to be no problem for the next few years. But from now on the software becomes the issue. For decades, sequential programming was the standard for all kinds of software, and only few systems cared about parallelization, because it simply was not needed. Now, since the processor architecture changes, the way one thinks about software has to change, too.

Moore's Law not only holds for single processors, but also for supercomputers and clusters, where parallelism plays an even bigger role. Therefore it is important to come up with middleware and systems that help developers and users to harvest the power of all kind of parallel and massively parallel hardware. This need mainly motivated this thesis to deal with middleware for multilevel-parallel and distributed systems and to present possible applications thereof.

1.1 Problem Description

The problem is not hardware. It's software![1]

This quote clearly emphasizes one of the main problems of today's Computer Science: Powerful hardware is at hand but the software leaps behind. In some areas such as High Performance Computing (HPC), this fact is not tolerable and thus the community started early to develop software that was capable to harvest most of the power the hardware has to offer. Traditional applications of this field are highly parallel simulations of physical or chemical phenomena. Besides optimizing the simulation code itself, these applications benefit from subsystems that help them to efficiently run in parallel on powerful cluster systems. Such subsystems or middleware encapsulate special tasks like communication and data transfer over networks and implement them in an efficient and transparent way. One famous example thereof is the Message Passing Interface (MPI) [50], which handles the passing of messages between the components of a distributed or parallel application. Another example are systems for computational steering which allow for the interaction of a user with running simulations through a visualization on sophisticated, hybrid visualization and simulation clusters. However, nearly all of the existing subsystems are optimized for traditional HPC applications and are only hardly suited for new applications from different backgrounds. Examples for such applications, which could also benefit from the power of modern cluster systems, are interactive simulations and Virtual Reality applications (IS/VR) or distributed interactive applications (DIAs) in general. The main problem is that in contrast to the traditional HPC applications, the needs of DIAs and IS/VRs (e.g. (soft) real-time interactivity, multi-user input, high end displaying techniques) are not supported by common and optimized subsystems. Thus, a parallel and distributed execution requires big effort to handle all communication and orchestration of the components. This problem is the motivation for this thesis. In our opinion it is vital to provide new subsystems for applications different than traditional HPC, to pave the way for the efficient usage of modern cluster's capabilities for a broader audience.

1.2 Areas of Interest

As mentioned before, a new field of applications, besides traditional HPC simulations, turned out to need a massive amount of computational power - highly interactive simulations and Virtual Reality (IS/VR) applications. Those related groups of applications are gaining important roles in both science and engineering and allow the simulation of whole virtual environments instead of single phenomena. VR is used for example to train people in all kinds of driving, operating, flying etc. scenarios and is getting more and more realistic with new technologies emerging and existing technologies

[1] Gregory F. Pfister in his book "In search of clusters" [67]

evolving. But, these technologies often require more computational power than one single computer can provide. Indeed, a simple driving simulator can easily run on a commodity personal computer, but if the scenario gets more complex by adding e.g. a sophisticated traffic simulation or additional users in the same environment, the limits of a single node system are reached very quickly. Thus, the development goes towards distributed systems and especially hybrid cluster architectures. They consist of tightly coupled nodes that are specialized either on visualization, simulation or other tasks. Those architectures allow having several simulations and several visualizations running concurrently. In addition to the coarse parallelism through hybrid clusters, IS/VR applications can also benefit from multicore nodes or nodes with special accelerator hardware. But to be able to handle those complex, distributed systems a universal middleware is needed. This leads to the idea of reusing and extending the techniques of computational steering and adopting them to the special needs of highly interactive simulations and VR.

However, a second problem arises from the development of using high performance, hybrid cluster systems for interactive applications. Not everyone who uses such an application can or wants to afford and operate a large hybrid cluster system. In most cases there is only one central and universal cluster that serves all different kinds of users. This leads to the need of remote operation services. Especially the remote visualization of interactive applications is a very sophisticated area of research, since vast amounts of visual data need to be transferred to a client as fast as possible. There are new developments in graphics hardware that allow to implement methods for compression and transmission of graphical data efficiently and without involving the Central Processing Unit (CPU). These advances can be used to realize a fully remotely operable platform for highly interactive simulations and VR.

1.3 Goals and Contributions

Thus, the main goal of this thesis is to design and prototypically implement software subsystems that enable the efficient and comfortable usage of hybrid cluster systems for interactive simulations and VR. The two main contributions of this thesis are:

1. A concept and prototypical implementation of a framework for computational steering which is well adapted to the needs of IS/VR applications. The concept evolves the idea of traditional computational steering and adds new models and data types to fulfill the requirements of IS/VR applications. The proposed framework, for the first time, allows the efficient coupling of multiple simulations, visualizations and steering components, running on a hybrid cluster and on multicore nodes.

2. A concept and prototypical implementation of a testing and benchmarking framework for compression and grabbing techniques for the remote visualization of IS/VR. A main focus lies on the optimization of the achievable FPS / quality ratio

under the assumption to have fast but limited network bandwidth (10MBit/s). The framework is used to design and implement new parallel compression algorithms that make use of programmable GPUs for their computations. The algorithms will be compared to others in this field and their advantages and disadvantages for certain scenarios will be described.

Figure 1.1 depicts the different areas and topics addressed in this thesis in more detail. The red box emphasizes the main contributions - the two new subsystems that enable IS/VR application to use the power of modern, hybrid cluster systems. The blue boxes represent topics or areas that are improved by the new systems or make use of it. This includes for example the two sample IS/VR applications Virtual Night Drive and d^{3}FACT insight which greatly benefit from the introduction of the CS and RV frameworks, but also general improvements in the field of image compression and communication servers through the contributions of this thesis. The yellow boxes represent the technical and conceptual foundations that, on the one hand, inspired the main ideas of this thesis (CS and RV) and, on the other hand, helped to realize and implement the new findings (e.g. CUDA/GPGPU and the Publish/Subscribe approach). To

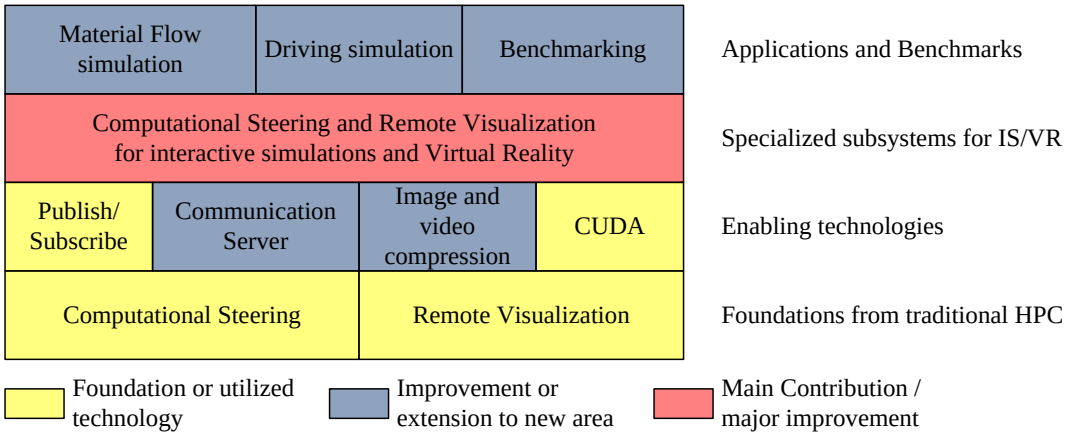

Figure 1.1: The areas covered in and the contributions of this thesis.

evaluate the developed systems and to show the improvements over existing systems, different usage scenarios and benchmarks are described. However, both frameworks have a strong focus on interactive simulations on hybrid cluster systems. There are no intentions to challenge existing systems for long running simulations or homogenous clusters (see chapter 4).

All relevant parts of the thesis are published in the proceedings of reputable conferences in the fields of computer graphics, visualization, signal processing and mechanical engineering (see [42], [43], [44], [45], [46] and [47]).

1.4 Thesis Structure

The thesis is structured into seven parts. After this introduction, relevant foundations on hybrid cluster systems and their applications are given. The main intention of this chapter is to clarify the need for specialized middleware to reasonably use the computational power of such hardware architectures. In the second part of chapter 2 a short statement on APIs for hardware accelerators and a deeper insight into the Compute Unified Device Architecture (CUDA) by NVIDIA is given. In chapter 3, a brief overview over existing frameworks in the broader field of Computational Steering and Remote Visualization for HPC is provided and limitations are shown. The subsequent chapter 4 derives the need for two specialized systems for IS/VRs on hybrid clusters from those limitations and briefly introduces their requirements. In chapter 5 a new model for the computational steering of Interactive Simulations and VR is presented and the basic components, functionalities and data types are described. The second part deals with the prototypical implementation of this model, which results in the so called CSIS (Computational Steering of IS/VR) framework. The focus lies on the realization of the framework by combining existing and new software to provide the desired functionality and flexibility. The last two sections of chapter 5 present two practical IS/VR applications that greatly benefit from the introduction of the CSIS framework and thereby help to show its potential. Chapter 6 is meant to introduce the concept of a framework for remote visualization which is also specialized on IS/VR, running on hybrid clusters. It is the first framework of its kind which uses the computational power of GPUs not only for rendering but also for the compression of frames. After briefly introducing the prototypical implementation of the framework, the focus lies on the description and implementation of the utilized compression algorithms, especially those that are GPU-supported. The last section of the chapter presents benchmarking and quality assessment results for the compression and grabbing algorithms, as well as for the overall system performance of the RV framework. Finally, chapter 7 lists the achieved contributions, draws an overall conclusion and gives an outlook on possible extensions and fields of application for the developed concepts and software.

2

Foundations

This chapter establishes the foundations of the thesis. Section 2.1 deals with the development of cluster systems over the last 10 years and the specification of hybrid cluster systems with possible fields of usage and an exemplary setup. In section 2.2 a short overview on Software for Clusters is given. After a general introduction, the focus lies on applications and subsystems that could make use of the special features of hybrid cluster systems. The last section of this chapter introduces a rather new technique that allows the usage of graphics hardware as universal, parallel coprocessors. This will be a new and innovative way to further utilize hybrid cluster systems.

2.1 Clusters Now and Then

To understand the origins and the development of clusters and hybrid clusters, a brief retrospection to the world of HPC 10 years ago, followed by a description of current developments in HPC is given.

2.1.1 The Rise of Clusters

The term cluster is highly unspecific and describes a lot of different systems with a wide variety of hardware, applications and users. However, Pfister gives quite a simple and fitting definition for a cluster in his book "In search of clusters" [67]:

> A cluster is a type of parallel or distributed system that:
>
> - consists of a collection of interconnected, whole computers,
> - is used as a single unified computing resource.

Additionally he lists the classic reasons for deploying clusters in the first place: Performance, availability, price/performance ratio, incremental growth and scaling. The

problem of these reasons, according to Pfister, is that they are quite generic and actually fit on most parallel systems. So he started asking the question why clusters are a good solution for many problems at the time he wrote the book[1]. The answers back then were the following:

1. For the first time very high performance microprocessors are available and are very cheap; that means they offer great price/performance ratio.

2. Also for the first time really high-speed communication techniques are available and getting affordable. Especially optical links are very promising.

3. Since the first days of parallelism, some useful tools for distributed computing have been developed and established to do the basic administration and handling of clusters effectively. That is the starting point of writing sophisticated software for clusters.

4. High availability becomes increasingly important since more and more users are dependent on the inexpensive computing services.

But he also clearly describes the problems that were still to solve to pave the way for an unlimited usage of cluster systems. His two main issues are:

1. The lack of "single system image" software. This means that all available systems for the administration and management of open-system-based clusters are only loosely coupled toolkits that still have a lot of compatibility and usage issues. There is now out-of-the-box easy-to-use for everyone software.

2. The limited exploitation of the possibilities a cluster has to offer. Only a few subsystems of current operating systems exploit the abilities like scaling performance and high availability. This on the one hand comes from the problems of writing parallel software on the low level and on the other hand brings up new problems for the design of sophisticated parallel programs.

By looking at the pros and the cons of clusters at that time, Pfister clearly pointed out:

> **The problem is not hardware. It's software**

Thus, what was primarily needed at that time was software that could make use of the performance and the redundancy of the clusters. During the last ten years a lot of such software was developed (see [6]). On the one hand the basis was created through parallel programming frameworks such as MPI or PVM and specialized cluster operating systems, mostly derived from standard OSes such as Linux (e. g. openMosix [4]) or Windows (e. g. Windows HPC Server 2008 [53]). On the other hand many software companies, especially in the simulation area, ported their basic software stacks to cluster architectures. That ranges from simply starting multiple independent runs on many

[1] The last edition appeared in 1998.

nodes (*capacity computing*) up to massively parallel systems that heavily rely on internode communications (*capability computing*). The availability of commercially developed and maintained software enabled a lot of big companies to use cluster hardware to compute many aspects of their everyday work life. This again reaches from dynamics simulation for product design and testing over CFD simulation for optimization aerodynamics up to geological or financial simulations for various purposes.

2.1.2 Hybrid Cluster Systems

Since Pfister wrote his book in 1998 a lot has changed, especially in the hardware area. On the one hand cluster systems have nearly completely superseded any other parallel computing architecture and are widely used for number crunching and high availability applications. On the other hand the commodity hardware that clusters are built off, has evolved a lot. Not only the CPUs and network interfaces have gotten faster by orders of magnitude, but also other components, which didn't play a big role at Pfister's time, can now be used to enhance the functionality and performance of modern cluster systems. One example of such a component is the Graphics Processing Unit (GPU). Modern GPUs are highly parallel and programmable multiprocessors with a strong focus on the processing of graphical data. But in some cases they also deliver great performance for non-graphic computations as for example shown in the GPGPU [24] forum. The integration of inexpensive GPUs in (some of) the cluster nodes can now serve two purposes:

1. To visualize the data that has been computed by the cluster system and thereby to help users to better understand and explore the results.

2. To support the CPUs of the cluster nodes with the computation of appropriate tasks.

The GPUs are not the only extension of commodity computers one can think of. Another interesting field for example is reconfigurable hardware (e. g. FPGAs) which can be added to cluster nodes as flexible and affordable co-processors for special tasks. That leads to the idea of creating hybrid cluster systems that contain nodes with special abilities (e. g. visualizing data) or specifications (e. g. FPGA accelerated). Hybrid cluster systems are therefore a subset of cluster systems and the extended definition would be:

> A hybrid cluster is a type of parallel or distributed system:
>
> - that consists of a collection of interconnected, whole computers
> - with some of the nodes performing special tasks on special hardware
> - which is used as a single unified computing resource

Hybrid cluster systems have many fields of applications and have a lot of advantages but also disadvantages over homogeneous cluster systems. For example a system with

dedicated visualization nodes[2] allows to do simulation and data analysis in one step and on one cluster. Even direct interaction between simulation and visualization is possible (as described in section 2.2.1). On the downside, the specialized nodes often perform different than homogenous compute nodes and thereby could complicate task scheduling and load balancing. But for many applications it makes sense to deploy special tasks on special machines to either increase throughput or allow data analysis on the fly.

So again, the main difficulty is that adequate software is needed, to efficiently make use of the computational power the hardware provides. This is where this thesis will have its focus. We analyze classes of software that can make use of hybrid cluster systems and address two main issues that will allow the efficient usage of hybrid cluster systems - **computational steering and remote visualization**.

2.1.3 The Arminius Hybrid Cluster at PC^2

This section briefly describes one example of a modern hybrid cluster system, which also was utilized for some of the tests and benchmarks described in this thesis. The Arminius Cluster was installed at the Paderborn Center for Parallel Computing PC^2 in 2005. It consists of 200 nodes for computation and 8 special nodes for visualization. Additionally all of the computing nodes contain basic GPUs and some of them are equipped with FPGAS for reconfigurable computing tasks. The following gives a quick overview over the hardware specifications:

- System Configuration:
  - 400 processors 64-bit INTEL Xeon
  - 16 processors AMD Opteron
  - 2.6 TFLOPS peak performance
  - 900 GByte main memory
- Compute Node Configuration (196 nodes):
  - Dual INTEL Xeon 3.2 GHZ EM64T
  - 4 GByte main memory
  - 80 GByte local disk
  - NVIDIA Quadro NVS 280 PCI-e GPU
  - InfiniBand Host Channel Adapter (HCA) PCI-e
- Visualization Node Configuration (8 nodes):
  - Dual AMD Opteron 2.2 GHz AMD64
  - 8 GByte main memory

[2] E.g. a computer that has a powerful graphics card and is connected to a displaying device.

  - NVIDIA Quadro FX 4500 PCI-e GPU
  - 2 nodes equipped with CUDA-enabled NVIDIA GeForce 9800 GX2 PCI-e GPUs
  - InfiniBand HCA PCI-e
- FPGA Node Configuration (4 nodes):
  - same as Compute Node Configuration
  - AlphaData ADM-XP with Xilinx Virtex-II Pro FPGA
- Infiniband Switch Fabric Configuration:
  - 216 port InfinIO 9200
- Disk Storage Configuration:
  - 5 TByte Fibre Channel RAID
  - 5 TByte parallel file system

The Arminius cluster is used for many different tasks, which can be grouped under the following categories:

- Batch jobs of non-parallel, standard software such as physical simulations or finite element methods for scientific or industrial purposes. Several jobs are started independently on a bunch of nodes and the results are collected and processed after the batch processing finished. Example: Gaussian [23] and Fluent [2]
- Traditional HPC and (massively) parallel applications such as computational fluid dynamics, finite element methods and molecular dynamics simulations that are computed in parallel on the computing nodes. The results are postprocessed manually and can be visualized through the visualization nodes which e.g. drive a stereoscopic power wall. Examples: padfem2 [9] and GROMACS [75]
- FPGA and GPU accelerated, parallel applications such as computer chess or computer go or biometric searches in large databases. Those applications make use both of the parallel computing power and the possibility to accelerate certain algorithms on the special nodes. Examples: Hydra (computer chess) [15] and GOmputer (computer go) [68]
- Distributed, interactive simulations and VR applications such as driving simulator or material flow simulations, which make use of the new subsystems for IS/VRs developed in this thesis. Examples: VND [46] and d^{3}FACT insight [12].

This is just a quick overview over the specifications and the applications running on the Arminius cluster. For more details on the system and its usage see [65].

2.2 Software for Clusters

As outlined before, the software that runs on clusters is often even more important and expensive than the actual hardware. Therefore it is very important that especially for new architectures such as hybrid clusters, software is available that allows users to utilize these complex systems for their needs. There are, according to Pfister, mainly three levels that help to generate the view and use of a cluster as a single unified computing resource. These levels are shown in figure 2.1. The most extendable and interesting level is the application and subsystem level. The other two levels (hardware level and operating system level) are basically derived from the commodity hardware and operating systems of single user computers. Therefore the top level will be explored more deeply. The levels toolkit and file system have been thoroughly studied and developed over the last decade and there are no big differences between those for traditional cluster systems and those for new systems like hybrid clusters. Thus the focus lies on applications and subsystems especially for hybrid cluster systems. To give a starting point, a short wrap-up on simulation in general is given. Afterwards we describe possible applications for hybrid cluster systems with a special focus on interactive visualization and simulation systems. The second part of this section briefly presents existing subsystems for traditional clusters and shows why the usage is limited for hybrid cluster systems.

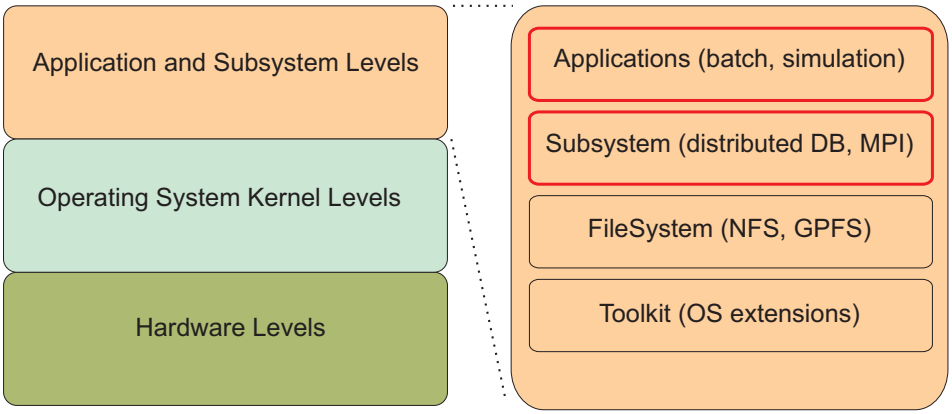

Figure 2.1: Different levels of a cluster architecture. Focus on applications and subsystems.

2.2.1 Application - Simulation

Computer simulations date back to the Manhattan Project during World War II where, for the first time, computers were used to simulate real world effects, in this case the process of nuclear detonation. At that time the computational power hardly exceeded the power of a current calculator. But the results helped the scientists to understand

processes that could not or only merely be investigated through experiments. Since then computers have evolved steadily and helped to realize simulations of more and more complex phenomena. With current technology it is possible to either focus on very specific problems and come really close to reality (e.g. short sequences of complex liquid or air flow simulations) or on approximating large scenes or even worlds to study and train people in virtual surroundings (e.g. a driving or flight simulator). The first kind of simulations is widely used in science and industry and has many applications. A few examples will be given in the following section. The second field is far less developed and currently undergoing a change of paradigms from needing specialized, proprietary hardware to universal (hybrid) cluster systems. Therefore this thesis focuses on the second field and the possibility to transfer well-approved techniques from the traditional field of simulation and visualization to this newer area.

2.2.1.1 Systems, Models and Simulation

To be able to classify both traditional and interactive simulations, the following section gives a short introduction on system, model and simulation theory. One goal of scientists is to gain knowledge through the structured study of facilities or processes in our environment. A closed set of such facilities and processes of interest is called a system. A definition of a system is given for example by [40]: "A system is defined to be a collection of entities that act and interact together toward the accomplishment of some logical end." Systems are classified to be either discrete or continuous. In a discrete system, state variables change instantly and are event triggered (e.g. switch or balance of a bank account). In a continuous system, state variables change continuously over time (e.g. a water level or the position of a car in motion). Systems are rarely purely continuous or discrete. In fact, most systems have both types of state variables. These hybrid systems are classified as continuous or discrete according to their predominate type of state variable change.

There are different approaches to the study of a system. The straightforward method is to study a system directly through experiments. Though this is often possible, some systems are simply not suited for experimental studies. In this case it becomes necessary to conduct studies on a model of the system. Other than that, there are a number of additional reasons to study the model of a system:

1. Cost reduction. For example, in the area of automotive engineering, models are used for conceptual studies and to study the aerodynamics of a proposed design, instead of building expensive prototypes.

2. Risk reduction and prevention of harm. An example is the model of how an infectious disease would spread through a population, without the need for human or animal test subjects.

3. Insight into not yet existing systems. One example would be the planned fusion reactor Iter, which cannot yet be realized but some process can already be

simulated and the planning can be conducted accordingly.

4. Insight into inaccessible systems. This is the case for many studies in the field of astrophysics and molecular biology.

Models of a system are either physical or mathematical. An example of a physical model are clay models used for the study of aerodynamics. Simple mathematical models can be solved analytically, more complex ones can only be simulated on a computer. A simple mathematical model is for example the modeling of an object in motion by applying Newton's second law of motion ($F = ma$). The object's position at any point in time can be easily determined through a simple calculation. Complex mathematical models, such as a model for the description of fluid dynamics, often require iterative computation of approximative solutions. An additional benefit of computer simulation over analytical model evaluation is that, once a model has been built and verified, it is very easy to examine variations of a system in order to find optimizations or verify hypotheses of the model's properties. A common definition of what a simulation is has been given in a VDI[3] directive:

> Simulation is the recreation of a dynamic process in a model, in order to gain valid insight on reality.[4]

Within this definition, a dynamic process is defined as a system's change of state, caused by some action on entities in the system. Though the simulation of a systems is beneficial in many cases, simulations also have some disadvantages. It is identification of a suitable type of simulation model, as well as its creation and verification which can be complicated and expensive. Especially the verification of a model is an important step in a simulation study, which, if done incorrectly, can lead to wrong decisions due to erroneous simulation results.

The last element needed for an exact description of a system is its state. According to [40], the state of a system "is defined to be that collection of variables necessary to describe a system at a particular time."

This raises the question of what exactly should be the initial state of a system's model. Prior to the start of a simulation, its model must be initialized with a feasible system state. Depending on the type of simulation in question, it is possible to either initialize state variables to predefined values or generate randomized initial state values. The latter is a common technique whenever stochastic simulation models are used. Besides the stochastic models, several other kinds of simulation models exist, which are discussed in the following section.

Depending on the objectives of a study, modeling the same system can result in various different models with substantial differences in complexity. A good example is the model for a computer simulation of ship traffic on a canal. In order to study the impact of heavily increased traffic on the length of a watergate queue, it would be

[3] Verein Deutscher Ingenieure (Association of German Engineers).
[4] Translation from German.

sufficient to include entities for every ship, the path of the canal and the watergate. If, however, the objective of the simulation is to examine water erosion at the canal borders caused by ship traffic, the detail of the model would have to be several orders of magnitude higher. For a water erosion simulation the water would need to be modeled with methods of computational fluid dynamics. Moreover, the complexity of a complete canal model would require high performance parallel computers. Besides their complexity, simulation models can be classified along three axes: stiffness, state transition and randomness. The different characteristics of these factors are discussed below.

Static vs. Dynamic Simulation Models:
The stiffness of a model determines whether or not it can change over time. Models which do not change over time are said to be static. Static simulation models could also be called timeless models, as they represent a model at a particular time or a model in which time is insignificant. The simulation of light in a room through methods of ray tracing is an example for a static model. Dynamic models, on the other hand, have properties which are a function of time. Such models evolve during the course of the simulation (e.g. a traffic simulation model).

Continuous vs. Discrete Simulation Models:
State transitions in a model can be discrete or continuous. The definition of continuous and discrete simulation models is defined analogously to the way continuous and discrete systems were defined above.

Deterministic vs. Stochastic Simulation Models:
Deterministic simulation models are models in which the output of a simulation depends on its input only. Numerical simulations, as used for the computation of fluid dynamics, are an example of deterministic simulations. On the other hand, stochastic simulation models use probabilistic components to simulate the distribution of events (e.g. arrival times) or create an initial state in a simulation model. Due to their nature, the output of stochastic simulation models is itself random and multiple simulation runs can only give an estimate of the model's characteristics.

Most traditional HPC applications like CFD and FEM often base on dynamic, continuous and deterministic models. However, there are other simulations with other models such as the material flow simulation which uses a dynamic, discrete and stochastic model. Subsystems that aim to support various simulations must also support multiple ways of data transfer and synchronization.

2.2.1.2 Distributed Simulations on Clusters

A distributed simulation is any kind of simulation executed on a distributed computer system. There are many reasons for a distributed execution of simulations. Firstly, the distribution of computation intensive high performance simulations can reduce the

total execution time of the simulation. Depending on its type, the particular speed-up of an individual distributed simulation varies. Secondly, simulations can be distributed geographically in order to create virtual worlds for multiple participants.

The challenge in the distribution of simulations especially on heterogenous (hybrid) clusters is that in many cases a virtual global time must be maintained. Several solutions for this problem have been devised. An intuitive solution for distributed time is to employ a server to manage a central physical time base for simulations. The downside of this approach is that one node of the simulation is exposed to more network traffic and processor load than others. Moreover, due to network latency time always contains a minimal error. In some cases, however, it is not necessary to implement a physical time; it might be necessary to be able to define an order on events in the distributed simulation. Such a time scale based only on the causal order of events is called logical time.

2.2.2 Application - Interactive Simulations and VR

As mentioned in the beginning of this chapter the simulation of virtual surroundings for training or evaluation purposes plays a minor role in today's scientific world. However, this area, which can be best described as highly interactive simulations or Virtual Reality (IS/VR), is gaining more and more importance in the industry. One example is car manufacturing. Through the extensive use of Computer Aided Design (CAD) nearly all parts of modern cars exist as virtual 3D models. Therefore it is easy to take these parts or whole cars, simplify them a bit and place them in real world like surroundings to perform several test scenarios even before the first prototype has to be build. This saves a lot of money for the companies and has therefore arisen a lot of interest in the past few years. Additionally those interactive simulations, in many cases, show their real potential when they are distributed over several nodes. Only through multi-user scenarios or the execution of several simulation runs in parallel and the visualization on advanced displaying devices, these systems provide their users with the additional information they need.

2.2.2.1 Definitions

There is no commonly agreed definition for IS/VR, because the applications are so various and differ in many aspects. Nevertheless, Jonathan Kaye and David Castillo give a quite good description on the term in their newsletter fittingly named Interactive Simulation Volume 1 [33].

> While David and I certainly didn't want to simply add a new buzzword, we felt that we needed an umbrella term to represent a foundation of core concepts and skills applicable to the evolving role of simulation to various disciplines, from training and assessment/certification to rapid prototyping, predictive modeling, and marketing. The "simulation" part was easy,

> because we certainly wanted to capture the idea of modeling reality, or a plausible reality, in some way. As you will likely hear by talking with us for more than a few minutes, however, "simulation" alone is never a solution; it must always be seen and planned for in the context of careful consideration for the types of interaction necessitated by the project goals. Because we felt it was important to emphasize the study of interaction in the development of useful simulations, we therefore arrived at the term "Interactive Simulation."

The term interactive simulation also appeared years before in the IEEE standard 1278.1-1995 "IEEE Standard for Distributed Interactive Simulation - Application Protocols" [18] where it was used to define a standard for military real-time battlefield simulations and virtual training. They give another, more specific definition which is strongly focused on their military scenario:

> Distributed Interactive Simulation (DIS): A time and space coherent synthetic representation of world environments designed for linking the interactive, free-play activities of people in operational exercises. The synthetic environment is created through real-time exchange of data units between distributed, computationally autonomous simulation applications in the form of simulations, simulators, and instrumented equipment interconnected through standard computer communicative services. The computational simulation entities may be present in one location or may be distributed geographically.

Both definitions show that interactive simulations differ in many points from the classical and traditional simulations mentioned above. They can only achieve their interactivity if they run in (weak) real-time and the main goal is to provide a good approximation of the reality but still in (weak) real time. The second definition also includes the possibility to distribute the simulations for different purposes, which directly leads to the idea of using clusters for their computation.

Virtual reality is another form of simulation and its definitions are also very vague. Roy S. Kalawsky wraps this fact up in his book *The Science of Virtual Reality* [32]:

> ... There are probably as many definitions of the term virtual reality as there are people in this field! ... While it is not important what we call 'the subject', it is important that we understand the limitations of the technique we are dealing with. Personally, I prefer the term 'Virtual Environments' but I recognize that 'Virtual Reality' is a term that is here to stay because that is the description used by international press coverage. In a virtual environment, the human is immersed in a computer simulation, that imparts visual , auditory and force sensations. The computer simulation can present conventional real-world environments without modification or entirely new environments where different (or no) physical law exists. The human operator is allowed to interact with components of the virtual environment

> through his/her responses being sensed appropriately and coupled into the virtual environment simulation. ...

The focus when speaking of virtual reality lies more on the audio-visual and haptical representations of an underlying simulation whereas the focus for interactive simulations lies on the reality-like simulation of an environment. On second thought the two terms have a lot in common. They both let users be part of a virtual reality and interact with its entities. Additionally both terms are only vaguely defined and therefore leave room for all kind of specializations. The next section discusses why a combination of them is very well suited for the execution on hybrid cluster systems, and what is still needed for a practical and common utilization.

2.2.2.2 Interactive Simulations and VR on Hybrid Clusters

In contrast to traditional simulation areas like CFD, where the main advantages of deploying clusters is a better (timely) performance or high availability scenarios (e.g. clustered web servers) where unlimited reachability is the main goal, interactive simulations mostly do not profit from these advantages. Although in some cases a performance increase can be achieved by distributing for example a real time dynamics simulation over several nodes, the problem remains that the parallel overhead eliminates the performance gains for most (weak) real time applications. But besides that, hybrid cluster systems allow the nearly unlimited extension of simulation setups. One demonstrative example would be a driving simulator in an urban scenario. One single computer can handle the graphics output, the dynamics simulation of the steered car and the simulation of a limited amount of computer controlled cars, let's say 10. With 10 of those nodes interconnected one could generate a scenario including 10 human users and 100 computer controlled cars interacting in one common virtual world. This is only a quite simple example and in practice there will be more sophisticated setups that make use of the different abilities of the groups of nodes in a hybrid cluster (e. g. special nodes for the visualization on a tiled display wall, hardware in the loop components etc.), but it shows the main advantages hybrid clusters can provide for interactive simulations and VR - **flexibility and extendability**.

However, the task of orchestrating this much more complex system remains the critical point as already seen for the traditional cluster systems. Good subsystems are needed that control the data exchange, give users the ability to dynamically setup desired scenarios and allow them to access the resources they need. There are good approaches originating in traditional clustering, but adaptations need to be made to make them work for the utilization of hybrid clusters for interactive simulations and VR. Section 2.2.3 introduces two important subsystems of traditional cluster computing and explains why they are also needed on hybrid clusters and what needs to be modified.

2.2.2.3 Research on Distributed Visualization and Simulation under the VisSim Project

The problems and limitations described in the previous section lead to the foundation of the joint VisSim project group at the University of Paderborn. The group consists of several researchers from different backgrounds (e.g. mechanical engineers, computer scientists and business data processing specialists) and was funded by the Ministry of Innovation, Science, Research and Technology of the State of North Rhine-Westphalia for three years. One main goal was to allow various interactive applications from different backgrounds to make use of the hybrid Cluster system Arminius installed at the PC^2. This thesis partially includes generalizations and extensions of the findings of the VisSim project.

2.2.3 Subsystems for Traditional and Hybrid Clusters

There are many subsystems (or middleware in other words) for many different task. All of them share the fact that they are neither a part of the actual application, nor of the operating system and that they provide services to ease the handling of distributed or parallel computers. One of the most prominent middleware is the Message Passing Interface (MPI) [50] which eases the internode communication on any form of distributed architecture. Especially for clusters, which by definition do not have a common shared memory, efficient communication methods are vital. But there are a lot more subsystems which serve very different tasks. Those reach from distributed databases to cluster wide administration tools.

A very interesting class of middleware, especially for hybrid clusters, is called computational steering (CS). In contrast to many other subsystems this class is aware of the different capabilities of the heterogeneous nodes of hybrid clusters. Section 3.1 explains the idea behind computational steering and illustrates why there is a need for further extending this technique to other fields of applications.

Another subsystem that is only needed for graphics or hybrid clusters is the remote visualization (RV). Unlike traditional clusters, which mostly produce textual or binary output once per simulation run, graphical and hybrid clusters often produce a constant stream of images, possibly on various nodes. If those images can not be displayed on a directly connected visualization device, they have to be transferred through the network. This, however, mostly requires some kind of postprocessing, since these data streams are quite big and require fast transmission. This is where remote visualization subsystems come into play. Section 3.2 introduces the most important existing systems and explains why there might be the need and the room for improvements to apply them on IS/VR applications on hybrid clusters.

2.3 Hardware APIs and NVIDIA CUDA

Besides subsystems for a unified usage of the whole systems, also tools to harvest the computational power of the single nodes are needed, in order to efficiently make use of hybrid cluster systems. Besides optimized compilers and high performance mathematical and statistical libraries, APIs are needed to access special hardware like GPUs or FPGAs. In this thesis one focus lies on the relatively new field of using the graphics hardware for computations other than graphics and visualization. Therefore a powerful general purpose API is needed. NVIDIA was the first graphics cards vendor that introduced such an API for their GPUs in 2006. Before that there were efforts to use Graphics APIs such as OpenGL to map general computing problems to the graphics card. The main problem of that approach was that general data structures and functions needed to be artificially mapped to data structures (e.g. textures) and functions (shader programs) in the graphics world, which led to complex and sometimes inefficient code.

By introducing the Compute Unified Device Architecture (CUDA) [61], NVIDIA offered a possibility to use standard C code, standard data types and many other functionalities of general purpose computing, natively on the fast graphics cards. Other companies (e.g. ATI, Intel) are following this trend with own proprietary implementations. Thus, there exists no universal API for general purpose computing on GPUs at the moment. However, NVIDIAs API is the defacto standard at the moment, since it offers by far the best functionalities, has a broad user and developer base and NVIDIA is actively pushing the utilization and development of the API. For that reason, CUDA was chosen for the GPU-based general purpose computing tasks in this work and is briefly introduced in the following.

Other tools, for example APIs for the integration of FPGAs follow a similar principle: They offer a well known interface to new hardware. However, this thesis focuses on the utilization of GPUs for certain acceleration purposes. Therefore the detailed description of other APIs and libraries is skipped at this point.

2.3.1 CUDA - Architecture

CUDA is a combination of software and hardware architecture (available for NVIDIA G80 GPUs and above) which enables data-parallel general purpose computing on the graphics hardware. The CUDA software stack is depicted in figure 2.2. The graphics hardware is accessed through a special CUDA-enabled driver, which can either be addressed directly by the application or through utilizing the CUDA Runtime API or optimized high-level libraries [5]. The CUDA API is an extension to the C programming language which helps to minimize the learning curve. It also offers read and write access to all areas of graphics RAM which was prohibited by graphics APIs before.

[5] Currently available are libraries for Fast Fourier Transformations (CUFFT) and Basic Linear Algebra Subprograms (CUBLAS).

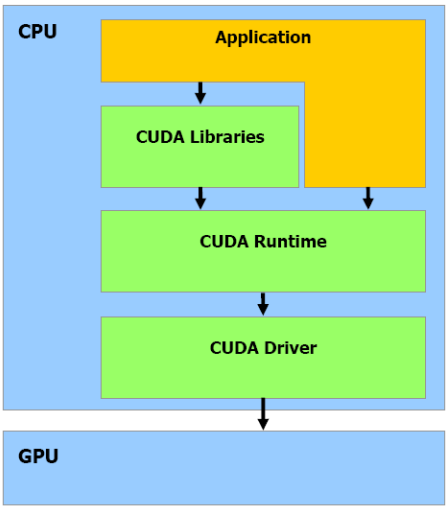

Figure 2.2: The CUDA software stack. Source [61]

2.3.2 CUDA - Programming Model

Through CUDA the GPU can be accessed as a coprocessor which is capable of executing a large amount of parallel threads. Thus, compute intensive parts of an application that base on similar operations on variant data can be offloaded to the GPU very efficiently. This is done by specifying one or more functions that will be executed by every thread of the GPU, compiling the functions to kernels and uploading them to the GPU. Input data can be copied from main/host memory to graphics/device memory and the other way round through optimized API calls.

Figure 2.3 depicts the thread hierarchy provided by CUDA. A kernel is executed by a *grid* of *thread blocks*. A *thread block* consists of a limited amount of threads that can communicate and synchronize very efficiently through fast but limited shared memory. Each thread inside a *thread block* can be addressed by a unique 1, 2 or 3 dimensional ID. This ID can be used to assign tasks or data to each thread. Threads within different blocks cannot communicate through shared memory but only through significantly slower device memory[6]. Through splitting the *grids* into *thread blocks* ideal scalability is achieved. Depending on the devices processing power the blocks can be executed sequentially or in parallel. However, it is vital to choose adequate parameters for the amounts of blocks and threads in a block for one kernel. *Thread blocks* can be addressed by 1 or 2 dimensional IDs and help to globally identify threads by combining the block ID and the thread ID.

[6]On current hardware accessing shared memory costs 4-6 clock cycle, whereas accessing device memory costs 200-300 clock cycles.

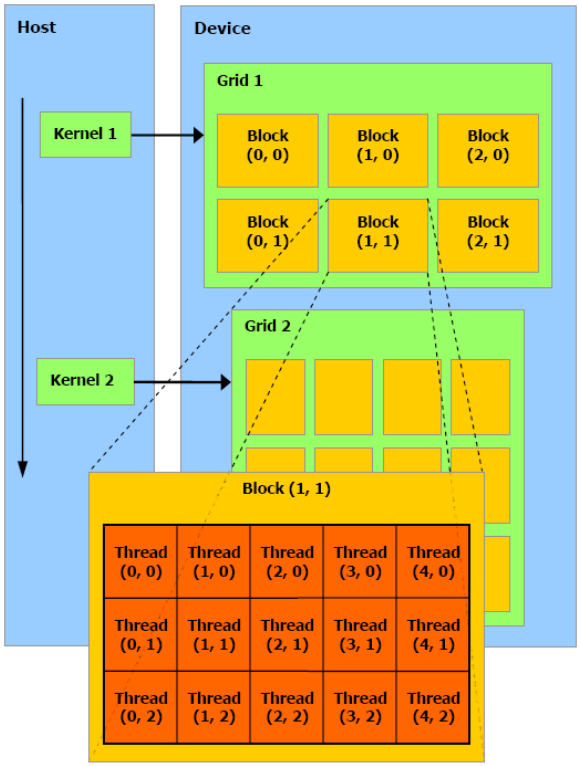

Figure 2.3: The CUDA grid, block and thread . Source [61]

2.3.3 CUDA - C Language Extension and Graphics API Interoperability

CUDAs main goal is to provide an easy and comfortable access of the GPUs computing power for developers of all kinds. Therefore, a minimal set of extensions to the C language is introduced to allow them to offload portions of their applications on the graphics hardware. The API is basically split into three components:

A host component that runs on the host and provides functions to control and access the GPU.

A device component that runs on the device and provides device specific functions.

A common component which provides built-in vector types and a subset of the C standard library that is supported in both host and device code.

The syntax of the API and its functions are thoroughly described in [61]. Another feature of the CUDA architecture is the interoperability with graphic APIs (OpenGL and Direct3D) which allows to use, for example, rendered images as input to CUDA kernels. Since this data already resides on the graphics device it only needs to be copied on the device to be processed by CUDA. This offers great possibilities for e.g. online image compression which is one topic of this thesis (see section 6.2.4).

3

Related Work

This chapter gives an overview over related work in the area of middleware for traditional and hybrid cluster systems. The focus lies on traditional computational steering and new approaches towards CS of applications other than classic simulations. Furthermore, related projects and their focuses in the field of Remote Visualization are presented and grouped into categories to allow a classification of the approach described later in this thesis.

3.1 Computational Steering

Computer simulations help scientists to study the behavior of complex systems and to speed up design and development in advanced fields of engineering. With simulations running in environments where accessibility is limited to assure best cost performance ratio and a secure environment, controlling and surveillance of such simulations requires more effort and thought than in conventional environments. Therefore a collection of methods and techniques has been developed to accomplish what is known as *computational steering* (CS). Mulder et al. are giving a good introduction to CS and an overview over current frameworks in [76] and [56]. The basic idea is briefly described in the following. Figure 3.1 shows the process of traditional computational steering. The components *User Interface / Visualization*, *Communication and Data Transfer* and *Application / Simulation* form the computational steering system. All components have well-defined interfaces and may be run in a networked, distributed environment. Additionally the steered application might also be distributed (e.g. a parallel CFD simulation) or run on a parallel machine. However, the most important part of the CS system is the component that handles Communication and Data Transfer which is often called communication server. It has to monitor both, the User Interface / Visualization and the Application / Simulation and update information that has changed. This data can be of different nature depending on the model of the simulation (see

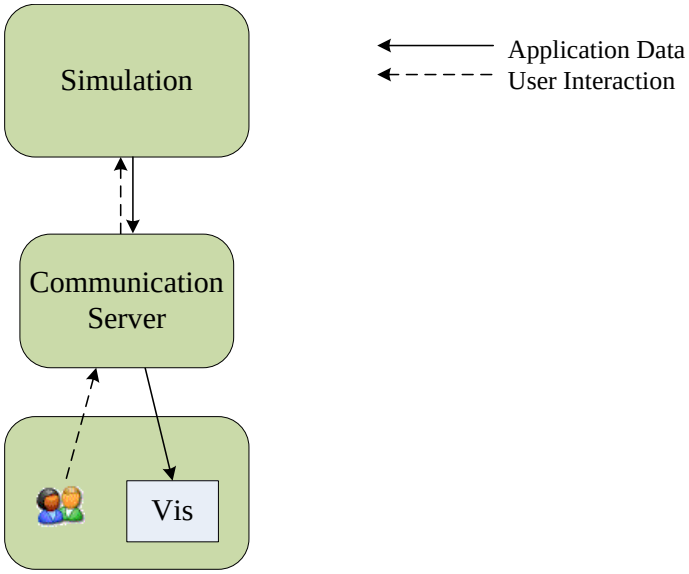

Figure 3.1: Components and data transfer paths of traditional computational steering.

section 2.2.1) and thereby influence the choice of methods for data transfer and synchronization. For model exploration, for example, the output data of the Application / Simulation and the input parameters of the User Interface / Visualization need to be exchanged. In distributed scenarios additional information on the distribution of the processes and the network load are of importance to the user. To be able to provide the data the communication server needs to have access to the Applications / Simulations input and output parameters, its execution code and its configuration. The access to parameters and monitoring information might be synchronous or asynchronous since not every operation on that data is permitted or valid at any time of the simulation process.

The other side of the system, namely the User Interface / Visualization has two tasks. The first is to present or visualize the extracted data from the Application / Simulation to the user in an appropriate way. The second task is to provide the user with the ability to change the steerable items of the Application / Simulation. These changes may vary for different CS scenarios from the setting of parameter values over exchanging source code to reconfiguring an application for optimization. Different User Interfaces / Visualizations may offer different methods for the setting of the items according to their nature: For example, textual input fields for source code or sliders or buttons for parameters.

3.1.1 Existing Systems for Traditional CS

CS systems can differ in many ways depending on the application that needs to be steered. Therefore several different systems were designed and published in the past that all have special characteristics. Three significant systems are:

SCIRun [64] and [64]: SCIRun is an open source system which integrates scientific simulation and visualization and is actively developed by the Scientific Computing and Imaging Institute (SCI Institute) at the University of Utah. Among the computational steering environments discussed, SCIRun is unique with its presentation of a consistent user interface allowing to design, execute, visualize, and steer scientific simulations. The fundamental concept of SCIRun is the visual programming of simulations and visualizations. SCIRun applications are constructed as a network of modules. Each module in a SCIRun network acts as a single-purpose unit whose parameters can be steered by the application's user. The SCIRun framework has many advanced modules for scientific simulation and visualization but is practically restricted to shared memory systems. This is due to its limited support for remote steering on multi-computer systems.

Cumulvs [36]: Other frameworks are specialized on certain parallel computation libraries. These frameworks usually allow steering of simulations which deal with large distributed data sets, but the provided methods are not generally applicable. In particular, CUMULVS, which represents this sort of steering frameworks, is restricted to simulations developed with the Parallel Virtual Machine (PVM) library.

CavernSoft G2 [63]: The most interesting framework, with regard to interactivity, is the CavernSoft virtual reality framework. It stands out from the other frameworks with its two types of update methods (UDP, TCP), which can be chosen as parameters, in dependence of a volatility/reliability trade-off. CavernSoft has a strict distinction between volatile variables which are (mostly) passed as TCP streams and the notification of events or the change of shared parameters which are passed as UDP packets. The whole framework was designed as a modular system that supports multiple protocols (UDP, TCP, HTTP) as well as distributed shared memory systems on the one hand and message passing architectures on the other hand.

The computational steering methods of the discussed frameworks are mostly concerned with the steering of long running simulations/applications and, unfortunately, none of the discussed frameworks satisfies the requirements of a flexible steering framework for highly interactive simulations and Virtual Reality. The reasons are manifold and vary from the lack of support for networked systems and synchronization mechanisms over high latencies to not supporting complex visualization scenarios.

3.1.2 Approaches towards CS for Interactive Simulations

In addition to the established systems for CS there are approaches to port the technique to other fields of applications. The first one was undertaken by Kesavadas et al. in 2000 [34]. They transfer traditional methods of CS to discrete events simulations of manufacturing systems. This allows them to control and steer a running simulation

through a graphical user interface. Changes made in this GUI influence the simulation immediately. To proof the validity of their work the authors simulate a single-server-queuing system for a certain amount of time. They allowed a user to control three significant parameters and showed that, by interactively adapting these parameters to the output of the simulation, the final results of the simulation could greatly be improved over those of uncontrolled execution. This approach is only a first step in the direction of using CS for interactive simulation. The authors did for example not consider the possibilities the approach can bring for distributed computing scenarios or advanced visualization setups.

Another approach to use VR to steer simulations is the CaveStudy project [69]. It uses existing systems for distributed VR (Cavernsoft[63] and Cavelib [49]) to build an environment which is able to steer remote simulations. Their main contribution is that the source code of simulation to be steered has not to be altered. This is achieved by a code generator which uses description files to generate a wrapper for the simulation. This wrapper acts as the server part of the framework. The generator also generates a proxy component which is the counterpart of the server on the VR client. This proxy communicates with the actual VR GUI. However, to guarantee the successful deployment of this approach, the steered simulation needs to fulfill certain requirements. One of them is that the simulation generates output that is written to a file or standard output and that there is only one output source for one simulation. Thus, it is not possible to directly integrate distributed simulations into the framework. Since the framework mainly bases on CavernSoft (see above), it also inherits some of its advantages and disadvantages. One main advantage is the distinction between events and volatile variables which allows to separately handle both types of data and provide best possible transmission conditions for both of them. A disadvantage is that only point-to-point connections are foreseen by the framework and thus in scenarios with a lot of m x n communication many redundant links have to be established. A central communication server might solve this problem. A second issue is a missing global synchronization method. Through their peer-to-peer oriented approach CavesStudy and Cavernsoft allow very flexible setups but do not offer a possibility to globally synchronize all of the connected components. Therefore it takes additional effort if for example several simulations should be compared stepwise. However, this approach comes very close to what will be presented in the following. It has its strengths in dynamic and collaborative VR environments whereas the proposed approach in this work aims for a well-scaling and synchronization-optimized system for hybrid cluster systems with connected VR / visualization devices.

This topic is still at a borderline between several areas (HPC, mechanical engineering, visualization, simulation) and just about to rise, so there are not many more systems to compare to. Currently, there is no system (besides those described above) to our knowledge which aims to allow the generic computational steering of IS/VR on hybrid cluster systems.

3.2 Remote Visualization

Chapter 2 focused on the foundations that are needed to understand the applications that are run on a Hybrid Cluster system in a distributed way. However, one problem arises when moving those simulations and their visualization away from the users' machine to big universal clusters: The access to those machines is physically limited in most cases. Mostly, there is one single point of operation (often a power wall or CAVE device) which is expensive to use and difficult to handle. If a user wants to benefit from the graphics and visualization power of a high-end, distributed system, he can only do this through the main operation site. A solution to this problem is software for remote visualization, which transfers the rendered images from a remote server to a local client that does not need to have significant computational power. There have been different approaches for such software over the past few years with different focuses. Some where designed primarily for remote administration purposes, others for interactive 3D applications (as shown in the following). But all have in common that the sheer size of image data is a limiting factor. The following sections give a quick overview on existing classes of remote visualization and the problems that may occur while dealing with these systems.

3.2.1 Classes of Remote Visualization

There are different classes of remote visualization for different tasks. In the following we will introduce three classes that group the most common existing systems. This should help to understand what the problems of the different classes are, and to which class the approach proposed in this thesis belongs. The table in figure 3.2 gives a rough overview over the three classes and their main differences.

3.2.1.1 Client-Side Rendering

Systems in which the application runs on a server but the graphics data (polygon meshes, textures, volumes) is sent to the client and rendered there belong to this class. The best known system in this class is a remote X-Server [20] which allows to start X-based applications on a remote server as if they were local. The main disadvantage, however, is that the rendering power of the server system is not used at all, while all rendering work is done by the client. This is fine for small desktop applications but insufficient for complex visualization applications that require certain rendering power. Another popular representative of this class is the Chromium [28] framework which is able to transfer the whole OpenGL stream from a server to a client. This is only one feature of Chromium but has the same problem as the remote X-Server, namely it does not use the servers rendering power. Chromium is a very flexible framework and is not only designed to do remote rendering. It also offers a lot of features to support all

	client needs advanced GPU	optimized for slow internet connections	optimized for high FPS	Compression capabilities
Client-side rendering	yes	partially	partially	none
Server-side rendering for 2D / administration	no	yes	no	high
server side-rendering for 3D /interactive applications	no	no	yes	medium

Figure 3.2: Classification of Remote Visualization Systems

kind of distributed and parallel rendering tasks. But for this work we only consider the remote rendering capabilities and classify them as Client-Side Rendering.

3.2.1.2 Server-Side Rendering for 2D and Administration

This class contains all systems that are mainly used for server administration and remote control. They render the images on the server side, compress them and send them to the client where they can be viewed with simple viewers. They also work with low bandwidth connections and mostly show the whole desktop of the remote server. They perform well with simple 2D and little interactive applications (e.g. administration and settings dialogs) but they are insufficient for highly interactive 3D applications such as a driving simulator or an interactive 3D viewer. The most popular representatives of this class are the Virtual Network Computing Framework (VNC) [70] and Microsoft's Terminal Service architecture [51]. Both offer possibilities to adapt the remote visualization to the available bandwidth and client device. But they are not yet optimized to achieve high frame rates and low latency for interaction.

3.2.1.3 Server-Side Rendering for 3D Applications (with and without Transparent Integration)

The third class of systems is specialized on displaying interactive 3D applications on remote clients. The server (or servers) with a lot of graphics processing power renders a 3D application (e.g. OpenGL-based). Every frame is read back, compressed (lossy or lossless) and send to a client. In most cases, only the window of the 3D application is processed, to save bandwidth and processing power. Popular systems of this class

are the SGI Viz-Server [72] and VirtualGL [78]. All of these systems are optimized to provide high frame rates with the available bandwidth. This is supported by the possibility to choose different resolutions and compression methods as well as certain level of detail mechanisms. Most systems allow a transparent usage of existing applications (mostly OpenGL-based). This guarantees a comfortable and universal utilization. Unlike that there are systems that are tightly coupled to a certain application to further increase performance by adjusting the rendering process dynamically to fit the needs of remote visualization. The subsystem presented in this thesis belongs to this class and will support both transparent and non-transparent integration. Furthermore it utilizes advanced hard- and software mechanisms to achieve high performance remote visualization.

3.2.2 Limitations and Problems

All of the systems mentioned above have to deal with a vast amount of data that needs to be transferred. This is because visual data is mostly pixel or voxel based. Therefore even single images which are displayed for only a fraction of a second can consist of one or more Megabytes of data. To provide a smooth animation, 30 or more frames are needed every second. All have to get from the GPU of the server, over the network, to the GPU and finally to the display of the user. There are already many approaches to minimize the amount of data that needs to be transferred for example through prefetching certain data to the client or by introducing level of detail mechanisms and compression of all kind. But there is still no solution on how to enable users to interact with a remote system as if it was local.

4

Two Essential Subsystems for IS/VR on Hybrid Cluster Systems

This chapter is meant to clarify the goal of this thesis and states what its impact is and also what it is not. The reasons why this thesis deals with software (middleware) for hybrid cluster systems were described in chapter 1, 2 and 3. In short the main points are:

- There are powerful hybrid clusters and various new applications that could make use of them, BUT there is no practical and universal way to get those applications to run on the cluster
- Clusters are still quite expensive and therefore have to be shared between different facilities. Hybrid Clusters add a new dimension of accessibility issues since they can, in most cases, generate graphical output which needs to get to its users. There currently exists no good solution to fulfill this task.

These are, in our opinion, the two main issues, why hybrid clusters are still only rarely used in the field of interactive simulation and VR, despite the fact that applications of this field could make good use of the power of clusters. Other than these problems there is still a lot of general work to be done until clusters (no matter if traditional or hybrid ones) can be used by the everyday user without limitations, but that is certainly not the focus of this thesis. This thesis focuses on designing, implementing and testing two systems to improve each of the problems described above.

4.1 Traditional Usage of Hybrid Cluster Systems

In order to analyze which software systems are needed for the usage of hybrid clusters for IS/VR, it greatly helps to take a look at existing systems for traditional HPC on

hybrid clusters and, in a second step (see section 4.2), derive the new systems from them. In figure 4.1, the major existing subsystems for traditional HPC are shown. It is observable that all of them address only parts of the cluster system and therefore hardly allow the simultaneous utilization of all resources of the whole system. However, this is only rarely required for traditional HPC applications since most of them simply do not need the advanced features of a hybrid cluster system (e.g. sophisticated, distributed visualization or the option to use a multi-user input). For traditional HPC applications, the focus lies on maximizing the utilization of the computing resources. Thus, most effort is put into the subsystems for parallel simulation (such as MPI and PVM, but also application level parallelism). Another important point for traditional HPC is the visualization of the computed data to help the users to understand and analyze the results of the simulation. In most cases this is done by post-processing the data, save a visual model and display it off-line to the users. Therefore no special subsystems are needed. With the upcoming of hybrid clusters new possibilities arose and subsystems were developed to enable the utilization of these resources. Computational Steering, as described before, allows for the (limited) interactive steering of running HPC simulations through a visualization. Additionally, distributed visualization or remote visualization systems allow for the usage of the powerful hardware for graphics processing for elaborate visualization tasks. However, all the subsystems depicted in figure 4.1 are independent and hardly any of them is ready for interactive usage, which is one of the main requirements of IS/VR applications. Therefore this thesis introduces two new subsystems that allow interactive simulations and VR applications to efficiently run on hybrid cluster hardware.

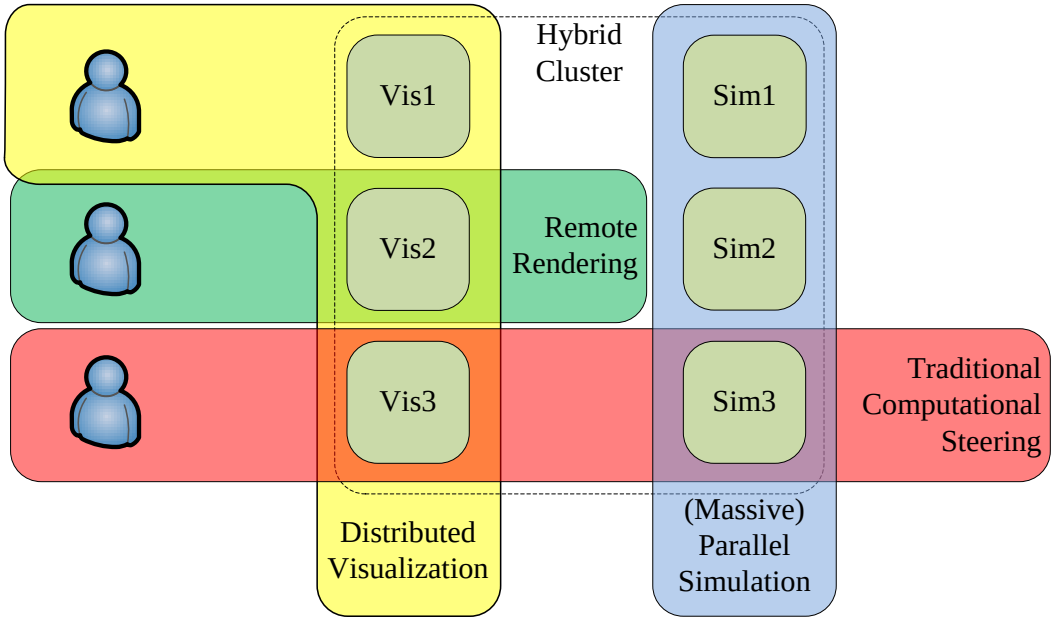

Figure 4.1: Different subsystems in existing hybrid cluster environments.

4.2 Hybrid Cluster Systems for IS/VR

Figure 4.2 depicts the vision of how hybrid clusters can be efficiently used for interactive simulations and VR. This is what will be presented and outlined in the rest of the thesis. The first subsystem, Computational Steering for IS/VR, for the first time offers the possibility to compose setups with arbitrary numbers of simulation, visualization and input components, and therefore allows to create flexible and scalable systems for the execution of the IS/VR. This helps for example to realize multi-user scenarios, comparative, parallel simulation runs or the integration of multiple autonomous simulations into one system. The second subsystem, a framework for remote rendering, addresses the problem of accessability of the cluster's visualization resources. It is meant to be a testing, evaluating and productive framework to develop, optimize, benchmark and use methods for the remote visualization of rendered simulation results. Again, the focus is to optimize the subsystem and its methods for best interactivity. The following two sections quickly introduce the main and new ideas for both subsystems whereas chapter 5 and 6 describe and evaluate them in detail.

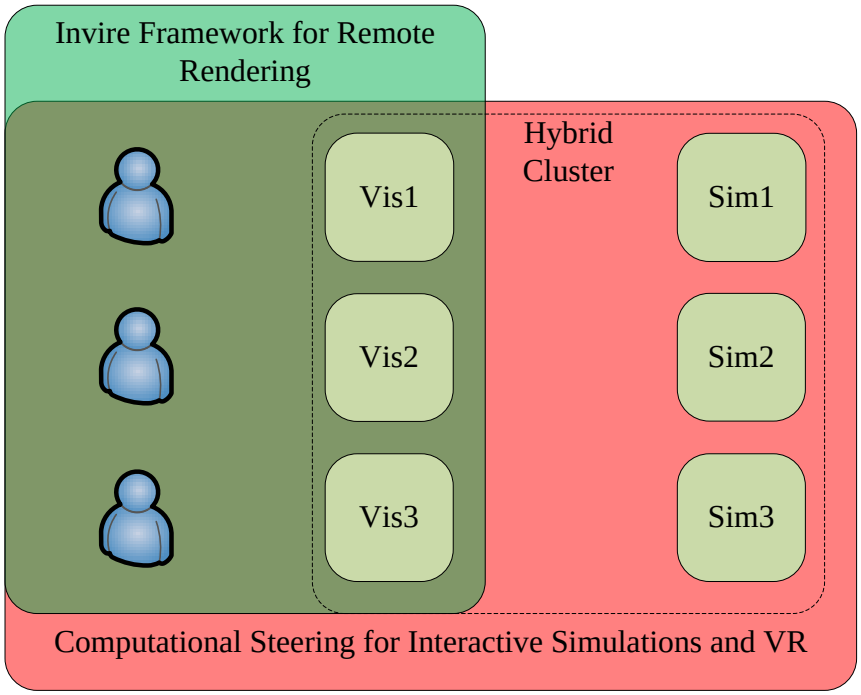

Figure 4.2: Two new systems to allow for the efficient usage of hybrid clusters for interactive simulations and VR.

4.3 CS for IS/VR - Idea and Requirements

The first system bases on the idea of computational steering and extends it for the special purpose of interactive simulations on hybrid clusters. After analyzing existing computational steering frameworks for traditional clusters and number crunching applications, a new approach for computational steering of interactive simulations on hybrid clusters is defined. This includes a theoretical approach supported by several models, as well as a working prototype which is utilized and evaluated by two sample applications.

The systems will cover the following requirements:

- Provide a flexible basis to orchestrate distributed, interactive simulations and VR applications.
- Encapsulate all communication effort and hide it from the user to provide a single-system-image of the cluster to a certain extend.
- Offer interfaces for all relevant components (simulation, visualization, input).
- Be deployable on different platforms and consist only of freely available software.

In chapter 5, section 5.1 introduces a concept for such a system and section 5.2 describes its prototypical implementation. Afterwards, section 5.3 shows the integration of the system into an exemplary driving simulator and section 5.3 does the same for an interactive material flow simulation.

4.4 Advanced RV Techniques for IS/VR - Idea and Requirements

The second system is a platform to evaluate different remote rendering techniques, especially in the field of compressing and grabbing the rendered frames. The main point of this system is to utilize the power of hybrid clusters, namely the High Performance graphics hardware of the visualization nodes to not only render the frames but also to compress them in several ways. Therefore we can achieve better compression rates in shorter times than comparable, existing systems are able to. This again allows users to transparently interact with interactive simulations on hybrid clusters.

The systems will cover the following requirements:

- Provide a framework to test and evaluate different methods for remote visualization.
- Enable the usage of the GPU for compression tasks.

- Offer new compression methods that allow sufficiently high framerates for the remote visualization of interactive simulations.
- Offer an interface for all kinds of graphical applications, possibly also a transparent interface.
- Prepare for the handling of distributed visualizations.
- Be deployable on different platforms and consist only of freely available software.

In chapter 6, section 6.2 introduces a concept for such a system and section 6.4 describes its prototypical implementation. A selection of benchmarks is given in section 6.5.

5

Computational Steering of IS/VR

As discussed before, computational steering (CS) is a valuable mechanism for scientific investigation by which the parameters of a running application / simulation can be altered and the results are visualized immediately (see e.g. [77]). It helps scientists to interact with their simulation and gives them more control than just to react on the results of a long running computation. Several CS frameworks and systems for different tasks and applications in High Performance Computing (HPC) have been developed. The most important architectures are presented and briefly described in section 3.1. They fit for many applications in HPC such as Computational Fluid Dynamics (CFD), Molecular Dynamics (MD) or crash simulation. All of them have in common that they lack the support for real-time interactivity and synchronization. Interaction is basically limited to changing simulation parameters such as for example the position of an object in a flow channel. This is done on a best-effort basis over time in most cases.

The focus of this thesis lies on the research of highly interactive, distributed simulation and distributed Virtual Reality systems (IS/VR). This is quite a new but very promising field in HPC. For a long time those systems were limited in terms of scalability, performance and flexibility. Proprietary systems on specialized hardware were used to simulate fixed use cases, such as driving or flight simulators or material simulation for rapid prototyping. With a flexible framework for IS/VR on a hybrid computing and visualization cluster, a basis for highly flexible and scalable applications could be provided that can be customized, compared and extended by the arrangement of various modules. One of these application is for example the distributed driving simulator Virtual Night Drive (see [7], [22] and [46]). It serves as one demonstrator for the proposed framework and is briefly described in section 5.3.1. However, the computational steering methods of the existing frameworks are mostly concerned with the steering of long running simulations and applications (as shown in figure 5.1), and none of the discussed frameworks satisfies the requirements of a flexible steering framework for highly interactive simulations. First approaches were made into the direction of using CS for other simulations such as discrete event simulations (see [34]). Still the methods

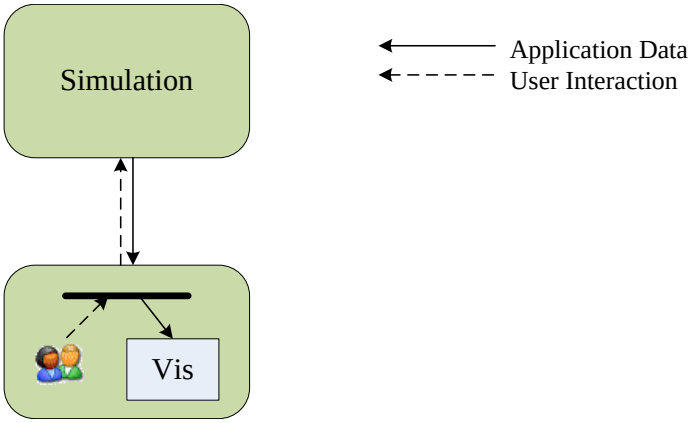

Figure 5.1: Traditional Computational Steering with one user, one visualization and one simulation.

of traditional CS are used, but only in other scenarios. Therefore, the following sections will introduce a concept for a framework especially designed for computational steering of IS/VR. Details are also published in [45], [42] and implementation details in [17].

5.1 A Concept for Extended Computational Steering of IS/VR

In order to have a starting point the next section describes in which aspects the traditional approach of CS needs to be extended to serve the needs of IS/VR. Thereafter, general requirements are specified and a conceptual design is presented.

5.1.1 New CS Models

Traditional computational steering is mostly limited to the steering and visualization of a single simulation (see figure 5.1). This basic definition is very strict and has only limited flexibility in the application of computational steering in areas different from traditional HPC simulations. Usually, there is only one simulation, one visualization and one side of user interaction. By extending each of these domains, three additional models can be identified that are needed to successfully transfer Computational Steering to IS/VR.

5.1.1.1 Collaborative Computational Steering

In order to collaborate on an interactive, distributed simulation, it is necessary to extend the traditional paradigm of computational steering. Any user involved should be able to access and steer the running simulation independently of his location. In this

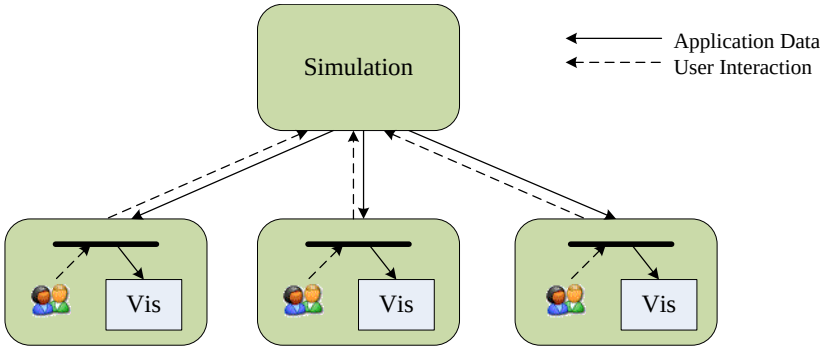

Figure 5.2: Collaborative Computational Steering.

scenario it is possible for one researcher to work on the level of detail of a simulation model and for another to study its particular performance figures. Figure 5.2 shows an example of a collaborative computational steering setup. Multiple sites connect to one high performance simulation, each with its own visualization and methods for user interaction. The simulation offers several parameters to be steered by the user. With multiple users steering the same application, care must be taken to guarantee consistency of steering parameters. In order to avoid conflicts during the access to a parameter, it has to be possible to lock certain parameters through locking mechanisms.

Collaborative Computational Steering especially makes sense in IS/VR since many of the existing applications include multiuser scenarios by nature. In a driving simulator for example several users can control cars in the same virtual world. It is also possible to introduce different roles for steering and visualization instances. One steering/visualization couple could for example act as an administrator for the simulation and the other clients are spectators or have predefined interaction privileges.

5.1.1.2 Synchronized Computational Steering

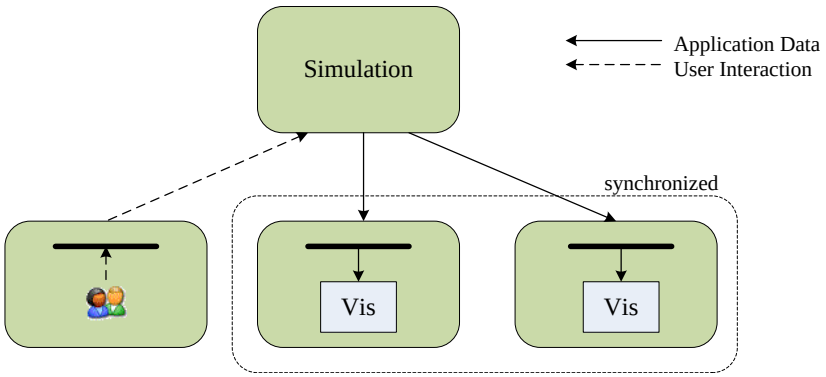

Figure 5.3: Synchronized Computational Steering.

Synchronized Computational Steering opens up new possibilities in the quality of vi-

sualization. This variation of traditional computational steering is a direct requirement of two recent and widespread visualization technologies: Very high-resolution displays and 3D data visualization (see e.g. Figure 5.4). Both cases require decent frame synchronization across multiple visualization nodes. High-resolution visualization is required by users working with data that is too extensive and detailed to fit onto a conventional screen, e.g. chip designers or city planners. Since the data throughput of high-resolution displays or tiled displays often exceeds data rates offered by high end graphic cards, data is fed to a tiled display by multiple graphic nodes, each displaying a magnified partial view of the original scene. The side by side positioning of displays in a tiled wall makes a possible time offset between displays disturbing for the viewer. This especially holds for volatile visualizations with large objects crossing multiple tiles of a display. Thus the exact synchronization of the visualization components is vital to provide a seamless visual impression. The second emerging visualization technology

Figure 5.4: Synchronized CS to render on a Tiled Display.

in which time offsets in visualization are more than a minor annoyance is 3D visualization. For a three-dimensional illusion of a data visualization it is necessary to create a pair of 2D images, of which each image represents a slightly different perspective of the same scene. This slight difference is set up in such a way that the separate presentation of each image to the eyes of the viewer creates an artificial depth perception. Figure 5.3 shows an example of a computational steering environment where a synchronized presentation of two visualizations is required. In this case, two nodes form a synchronization group in order to minimize time offsets. Uneven processor loads and scheduling differences on the visualization nodes are avoided by the introduction of a third node, which handles the user input to the steered application.

In addition to these two cases (high-resolution and stereoscopic displays) even more sophisticated scenarios like CAVE environments or specialized setups for e.g. driving simulators can be realized through Synchronized Computational Steering, by setting up and synchronizing the desired amount of visualization components.

There are already systems (e.g. Chromium [28] or VRJuggler [8]) that are able to distribute graphics output over several computers to drive tiled walls or stereoscopic displays. However, there are two main advantages of using synchronized CS for that task:

- In contrast to systems like Chromium, no graphical data (e.g. OpenGL calls, textures and scene models) has to be sent over the network since all of this data already resides on the client system. Only simulation data, such as positions of moving objects or the current position of a global camera, needs to be transferred to realize a distributed visualization.
- With Synchronized CS it is possible to add and remove visualization instance during the runtime of the simulation, whereas in the established systems all components need to be known a priori.

On the other hand those systems are very good in terms of transparent integration. Chromium for example is able to distribute almost all OpenGL based software and offers various so called SPUs that allow for different post-processing operations directly on the graphics stream. For systems that already provide interfaces for CS, synchronized CS can achieve even better performance and higher flexibility.

5.1.1.3 Concurrent Computational Steering

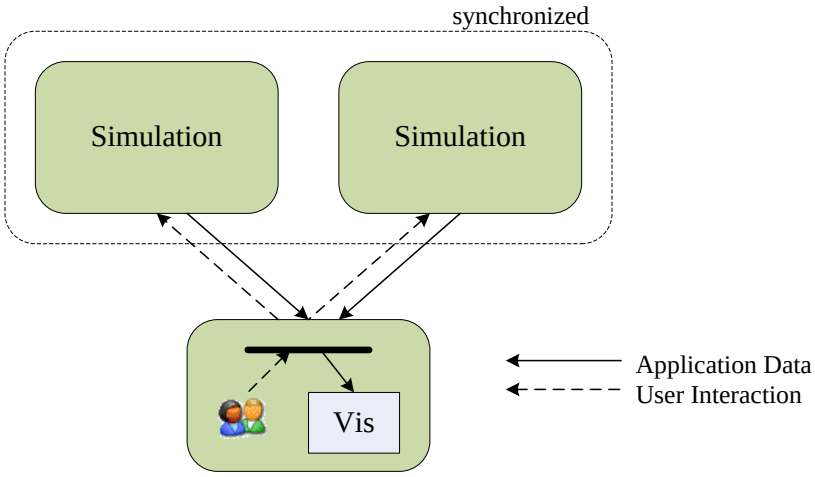

Figure 5.5: Concurrent Computational Steering.

In many scenarios one simulation run is not enough to manifest the simulation results, or one wants to compare the results that are generated by different initial parameters. Therefore several simulation runs are needed. Traditionally, multiple simulations were started sequentially with several parameter variations in order to study their influence on the simulation results. Although, traditional computational steering greatly helped

to reduce the turnaround time of the parameter variation, simulation and visualization loop, the conclusions of a simulation are still drawn based on the comparison of two or more successive simulation runs. Furthermore, it is quite difficult to perceive small effects of a changed parameter in a complex simulation. Researchers then have to rely on a quantitative analysis of the simulation results. In such cases, employing traditional computational steering adds no value to a simulation study at all. In order to make computational steering more useful in comparative simulation scenarios, it is important to allow the concurrent execution of two or more simulations and to allow an aggregated visualization of the simulations' outputs. Figure 5.5 shows an exemplary setup of a comparative simulation scenario and exposes two challenges which arise from concurrent computational steering. On the one hand, the user must be able to selectively change parameter values throughout all concurrent simulations, or it must be defined which user steers which simulation. On the other hand, to improve the usability of concurrent computational steering, certain parameters can be treated as virtual global parameters. Setting the value of such a parameter causes it to be set in all steered applications instantly, without further user interaction. Besides the requirement of methods for user input direction, concurrent computational steering requires synchronization of time among the concurrently running simulations in order to make their results interactively comparable.

5.1.1.4 Combinations of the Models

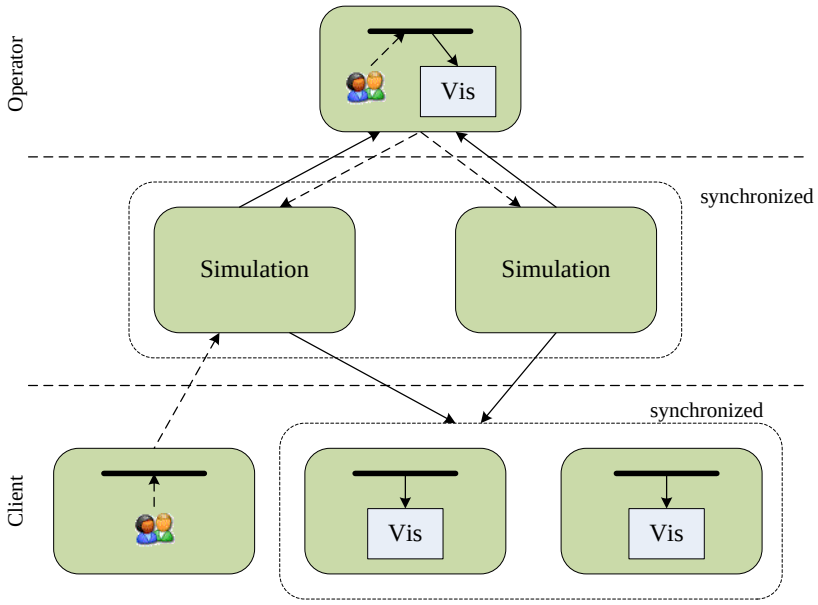

Figure 5.6: Combined, Extended Models of Computational Steering.

In order to achieve maximum flexibility, another goal of computational steering for IS/VR is to allow arbitrary combinations of the models described before. Figure 5.6, for

example, shows a scenario that consists of multiple users (collaborative CS), multiple simulations (Concurrent CS) and a complex visualization device (synchronized CS). Additionally the users are separated into groups with different privileges (operator, test user). A real application of such a scenario is described in the section 5.3.1.

5.1.2 Requirements

In order to realize the models described before and thereby create a framework for computational steering of IS/VR, some special requirements (besides general ones in software engineering, such as maintainability, extensibility, and portability) need to be considered. These are the following:

Dynamic setup: Especially in IS/VR, the configuration of the simulation, input and visualization components is very divers and not necessarily fixed beforehand. Therefore, one important requirement to a specialized framework is the possibility to set up and arrange these components in a flexible and dynamic way. Furthermore, the systems should not be fixed to a certain configuration after the initialization phase but should still be extendable and scalable during its runtime.

Locking: Dedicated methods to exclusively lock parameters of a component are other main requirements for the realization of concurrent CS and collaborative CS. These models can only work correctly if there is a mechanism that takes care of data consistency and the queuing of simultaneous write requests.

Low latency: Low latency is a direct requirement of IS/VR since interaction is a crucial component of these systems. Lack of immediate interactivity is the main obstacle to user acceptance of a computational steering framework for highly interactive simulations. Therefore techniques need to be considered which ensure the fast and possibly prioritized transfer of latency relevant information.

Synchronization: The requirement for synchronization among simulations or visualizations in the framework is a direct requisite from the synchronized and concurrent computational steering models.

Classes of data: This last requirement is not that obvious but will turn out to be very important. Especially in IS/VR there are different types of data that need to be treated differently by the CS framework. Therefore it is very important to offer special mechanisms to, for example, publish parameter changes once to a special group of receivers or update simulation data in every step of the simulation in each connected visualization.

These requirements form the basis of the concept for computational steering of IS/VR and their realization is described in the following sections.

5.1.3 Conceptual Design and Classifications

In the following sections, the concept for a flexible computational steering framework is explained. In general, it is based on the idea of a centralized communication server where this server and attached simulation, visualization and steering clients should be as dynamically and loosely coupled as possible to grant extensive flexibility. To design the framework, the actors need to be identified and a classification of the data that will be transferred has to be made.

5.1.3.1 Actors in Computational Steering

The idea of the extended models of computational steering is to allow multiple simulations, visualizations and steering sites to interact as one seamless application. Figure 5.7 combines the preceding specific scenarios of computational steering into one general model of a steering framework. It allows any number of the three different classes of actors in computational steering. The common definition of an actor is that of the Unified Modeling Language (UML) standard: "an actor is something or someone who supplies a stimulus to the system"[62]. From the framework's point of view, actors are simulation, visualization and steering actors. The steering component is an active actor since it can "initiate interactions with a system", the visualization components are passive actors since they "act as targets of requests or are activated by the system" and the simulation components are both active and passive. In the following, the functional

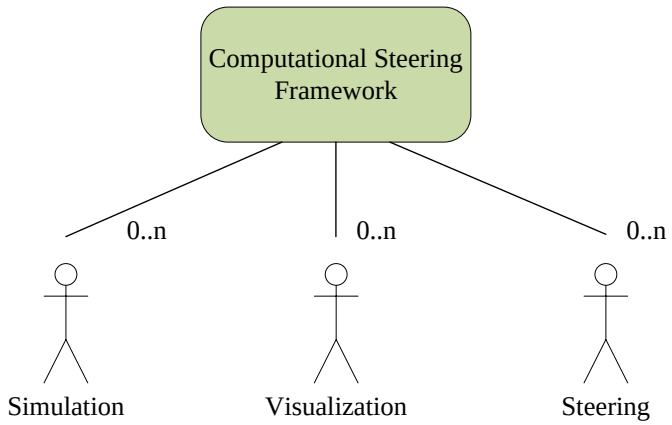

Figure 5.7: General model of the steering framework.

role of each type of actor is described.

Visualization: The purpose of visualization in computational steering is to provide a visual representation of the state of a simulation, where the state itself can be distinguished into simulation parameters and the results of a simulation. A visualization can have several parameters which define the way in which the state of a simulation is presented. The visualization parameters are local to the visualization and have no effect on the computation of the visualized data.

Simulation: The simulation is where the results of the simulated process are computed. A simulation is considered as a single component; a potential distribution of the simulation among multiple processing nodes is neglected from the framework's point of view. Thus, a distributed simulation mostly provides a single point for data access and therefore can be handled as a single component in the most cases. All data provided by a simulation is regarded as its output, which itself is considered volatile, as it is recomputed in every iteration of the simulation. In the special case of IS/VR the duration of one iteration needs to be short enough for the whole application to be interactive – i.e. less than 100 ms (see section 5.1.5). Since the simulation needs to run in (soft) real time all results need to be ready before the end of an iteration. Otherwise a mechanism is needed which aborts computations that take too long. In order not to be faster than wall clock time, the simulation is idle for the remainder of the iteration. Like the visualization, the simulation also has local parameters. As they are used in the computation of a simulation step, altering/steering these parameters directly influences the computation of the output data.

Steering: Although steering, which is the actual user interaction, is often integrated into visualization, it is useful to regard it as an independent component. This is especially the case for scenarios where autonomous steering devices are used to interact with a simulation. The steering component is used to change parameters in visualization and simulation components. Where steering is done by input devices delivering a continuous stream of data (e.g. analog devices), it is either necessary to adequately quantize this stream of data and frequently update a parameter or to deliver a continuous stream of data.

5.1.3.2 Classification of Simulation Data

Until now no distinction has been made between the types of data accessed and exchanged through the computational steering framework. Especially in highly interactive computational steering of IS/VR, it is essential to distinguish between two types of data, namely volatile state data and static parameters.

Volatile state data: changes in every time step of the application. The main characteristics of volatile state data are, firstly, that it has to be transferred in every pass of the simulation process and, secondly, that all clients have to get it simultaneously. Especially for the synchronization-based models it is very important to ensure on-time delivery of the data.

Static parameters: change only through steering actions by a user or another actor (timer or event triggered) of the application. If no action is taken the values of the parameters are constant over the applications iterations and don't need to be transferred. Only if a new actor joins the system all necessary parameters need to be send to him for initialization.

The differentiation into volatile state data and static simulation/visualization parameters allows to apply different methods for their distribution across all actors. The following two sections describe how volatile state data and parameters can be passed in a flexible and dynamic way.

5.1.3.3 Passing of Volatile State Data

There are two requirements for the passing of volatile state data by the computational steering framework. On the one hand, the volatile data needs be updated in every time step. On the other hand, data passing needs to be synchronized. The second requirement is easily fulfilled if the framework passes an item of input data to all its respective subscribers prior to their internal iteration, and fetches their output immediately afterwards. Thus, the main loop for passing volatile data between the actors of a computational steering application is the following (from the actor's point of view):

1. For any variable which has been computed in the previous iteration (output data): Copy its value to all interested actors.

2. Iterate all actors.

3. Collect output data values from actors for use as input in the next iteration.

The association of producers and consumers, which is necessary for this approach, can be stored in a simple array and is essentially a graph data structure (map). This data structure determines the routing of data between actors.

5.1.3.4 Scheduling of Volatile State Data

The scheduling of the data is done in a dependency agnostic way. That means that the realities between consumer and producer are only used to determine the volatile state variables to pass. Data dependencies between the actors are ignored in terms of scheduling. This might lead to delayed data as shown in figure 5.8 a) : The variable x from the input component first reaches the visualization after the second iteration since its computation is depended on the simulation component. In other words, data that reaches the visualization is already outdated due to the scheduling that is used. This fact is inevitable since the dependency exists independent of the utilized scheduling method. Thus, it is important to reduce iteration time in a way that the longest dependency chain can still be computed in an interactive time frame (about 100 ms). If there are multiple dependencies, as for example shown in figure 5.8 b) where the visualization depends on input from the simulation as well as on input directly from the input component, more sophisticated scheduling methods need to be considered. One approach is graph scheduling, where the knowledge of the data dependencies is used to influence scheduling decisions. Thus, for a directed acyclic graph of data dependency it is possible to sequentially order the iteration of actors in such a way

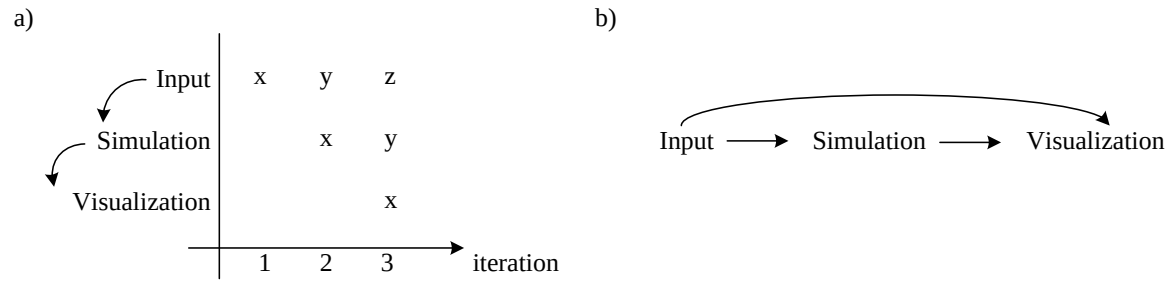

Figure 5.8: a) Data delay through dependency agnostic scheduling b) Graph-based scheduling introduces longer iteration times.

that all data on which an actor depends is computed before that actor's iteration. Sequential iteration is particularly useful in a scenario as depicted in figure 5.8 b). In this example, the actor's visualization depends on data from the actor's input and steering. Through the dependency chain input → simulation a dependency agnostic scheduling algorithm would supply the visualization component with data which has a time offset of 2 times the duration of a single iteration. An algorithm adhering to a dependency graph would sequentially iterate the actor's input, simulation and visualization. Thus, the data input to an actor would always be up-to-date. However, the disadvantage of these algorithms is that one iteration of the scheduling algorithm then requires the time needed to sequentially iterate the longest dependency chain.

5.1.3.5 Dynamic Mapping of Volatile State Data

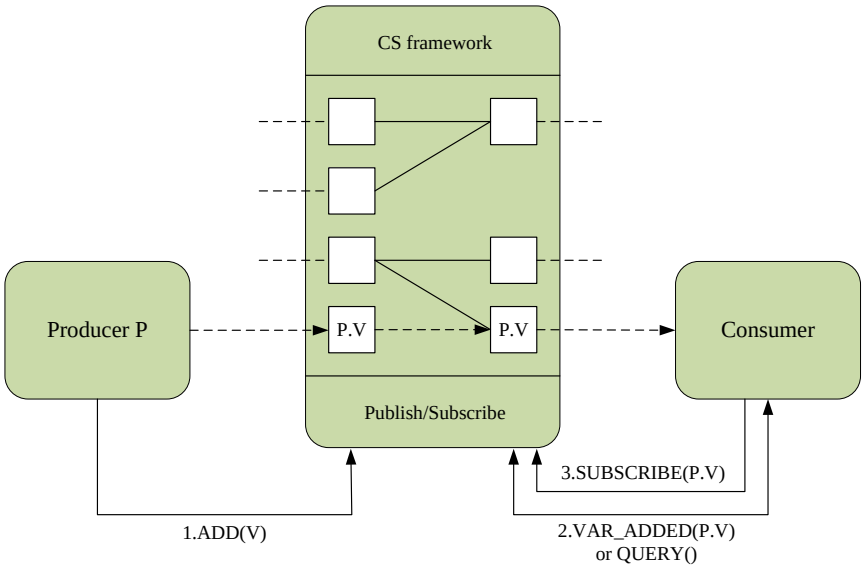

Figure 5.9: Dynamic map for volatile state data.

To ensure the flexibility of the framework, the mapping of volatile data has to be dynamic, which means it must be possible to add consumers and producers of volatile data to a running application. On the one hand, consumers of volatile simulation data must be able to notice when a new producer provides an additional set of volatile variables. This of course implies that producers must be able to add new data sets on joining an application. On the other hand, consumers, which are joining an already running application (e.g. when registering a new visualization), must be able to determine available sets of volatile state variables. Figure 5.9 shows how these requirements are met by using publish/subscribe communication and adding a dedicated remote procedure call (RPC) interface to the communication server. By this interface, the communication server provides access and notification methods for its internal mapping data structure. These methods are

```
ADD / REMOVE (var_name)
SUBSCRIBE / UNSUBSCRIBE (var_name)
QUERY ()
```

The first two methods are used by a producer to add or remove volatile state variables to, respectively from, the mapping. When adding or removing a variable to the mapping, the communication server sends a notification of the form:

```
VAR_ADDED / VAR_REMOVED (var_name)
```

to inform clients which might be interested in subscribing to it. To request a particular parameter, a consumer can use the `SUBSCRIBE` method. If a producer has already registered a variable with this name, the mapping is established immediately. It is not necessary for the consumer to wait for the notification of the addition. If the variable the consumer wishes to subscribe to has not yet been registered, the mapping is delayed until a producer registers the matching variable. The last method of the remote interface, `QUERY`, allows a connecting client to determine what volatile data is already available. More sophisticated querying mechanisms can be implemented but are not relevant for the conceptual design.

5.1.3.6 Decoupled Handling of Shared Parameters

In contrast to volatile state data, parameters change less frequently and only as a consequence of user action. Thus, instead of updating parameters every time before an actor's iteration, a parameter should only be updated when necessary. Since the number of actors in the steering application is not assumed to be fixed during runtime, the decision was to base the concept for steering of shared parameters on the flexible publish/subscribe method. As publish/subscribe can implement a group-based communication, in which nodes are unaware of group structure and size, this approach is expected to lead to the most flexible framework for computational steering. Due to

the use of publish/subscribe, a steering client can be implemented with a minimum of knowledge about the actual steered application and its components. Just knowing the name and type of a steerable parameter is sufficient to compose a message which sets this parameter in one or multiple components.

As for the usage of publish/subscribe for volatile state data, the connected actors exchange publish/subscribe messages for the static parameters. However they do not subscribe for the reception of parameters through a communication server, but directly to the reception of updates of the parameter. That means that whenever a producer changes the value of a parameter, he publishes it and all subscribers receive the updated value directly. All parameters ever published by a producer are listed in a separate parameter map and can be queried by potential consumers, according to the handling of volatile state data.

5.1.4 Architecture, Components and Exemplary Workflow of the Framework

Figure 5.10 shows the basic architecture of the CS framework for IS/VR. Several actors are connected through a steering library that encapsulates all functionalities of the Steering API (adding/removing of variable, subscribing/unsubscribing to parameters/variables, querying etc.). An actor can also be on both sides of the systems if he offers input and output variables. An example for such an actor is a simulation instance which can be steered through a steering instance and displays its output through a visualization instance.

In the following the two main server components of the framework are presented and a sample workflow is described to understand the interaction between the actors and the framework. Implementation details can be found in section 5.2.

5.1.4.1 Communication Server

The communication server is the central part of the CS framework. Its task is to map the volatile state data output of the producers to the input of the consumers. This is done by a dynamic variable map which links one or more variables of each side. It also serves as a synchronization instance as it has an internal clock, which provides a stroke after which the values of all variables are transferred from left to right. This clock limits the time the components have for their iterations and offers several modes. It can be set to strictly adhere the time limit and resend old values if an actor did not compute inputs in time or it can be set to be less strict and allow certain delays, which can cause the whole system to stutter. Thus, the clock settings need to be taken carefully and be adapted to the application's needs and capabilities.

5.1.4.2 Publish/Subscribe Service

The Publish/Subscribe service has two main functionalities. The first is to allow producers to announce their volatile state data output. This information can be queried by the consumers or is pushed to them if they subscribed to receive announcements of that producer. The P/S service is tightly coupled to the communication server and invokes the dynamic map generation if a consumer subscribes or unsubscribes to a producers volatile state data variable. Additionally, the P/S service is responsible for the handling of the static parameters. Those can be announced and subscribed to by all actors of the system. Once an actor subscribed to such a parameter, he receives all further updates until he unsubscribes it or the producer ceases to offer the parameter.

5.1.4.3 Exemplary Workflow

In the following, an exemplary workflow is described for the actor on the left. The depicted actor computes a couple of variables in each of its iterations. Thereby, a set of parameters is incorporated into the computation. A user who, in direct interaction with the actor, changes a parameter, implicitly notifies other users of the particular parameter's change. For that purpose, the steering library employs the P/S service. Additionally, the volatile variables are concurrently transferred to other potential actors through the mapping and synchronization component of the system. Not shown in the diagram is the actual synchronization which is imposed on the actor's iterations. The

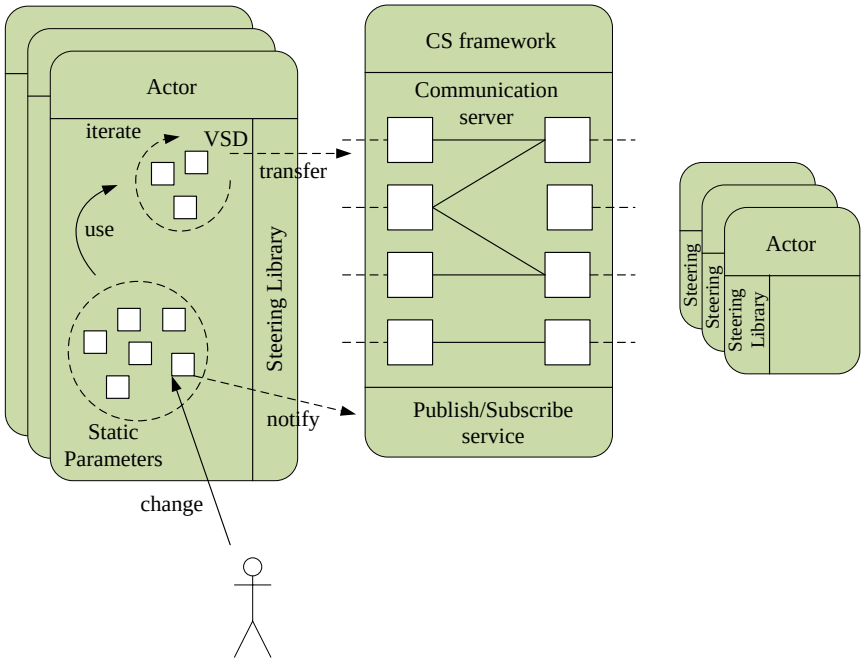

Figure 5.10: Architecture and collaboration of the framework's components.

components of the computational steering system (visualization, simulation, etc.) need

to be implemented against the steering library. Through the framework, an actor can achieve the following objectives: 1. receive or transmit volatile state data. 2. share static parameters among a group of actors. 3. decide on the subscription to newly added volatile variables. 4. initiate internal function calls on a parameter change. In order to develop an actor for computational steering, the general approach is to declare all produced volatile variables, as well as subscriptions to volatile variables and parameters to the framework. Furthermore, the developer is able to associate callback functions with certain parameters.

Section 5.2 describes the implementation of the presented concept for Computational Steering of IS/VR and section 5.3.1 whereas section 5.4 show two scenarios where it is used in practical applications.

5.1.5 Consistency, Performance Estimation and Latency Optimization

One very important aspect of the proposed framework is the achievable performance for the distributed, interactive simulations. In contrast to traditional HPC applications, where in most cases the network throughput is the limiting factor, interactive simulations heavily depend on very low latencies. In [14] Delaney et al. surveyed the effects of consistency and latency on distributed interactive applications (DIAs). They found out that the best state a DIA could achieve is absolute consistency. That means that "at any point in time, all players (users) should ideally see the same information at the same time independent of the network". However, this absolute consistency is impossible to be attained by a DIA since the existence of network latency and the time to pass information between two actors cannot be ignored when compared to the time between events. Consistency mainly refers to three aspects in DIAs: synchronization, causality of ordering and concurrency. All these aspects influence the system's consistency in one way or the other. Linked to those aspects are the issues of responsiveness ("the time taken for the system to register and respond to a user event") and fidelity ("the degree to which the representation within a simulation is similar to a real-world object, feature, or condition in a measurable or perceivable manner"[18]). These three issues, consistency, responsiveness and fidelity, need to be balanced individually for each application to give the users of the system the best possible result. In the following we focus on the problem of latency since it is a cause for all three issues, because it influences the network that actually facilitates the DIA. The authors of [14] claim that there is no commonly agreed definition of latency since it has so many variants. However, the term network latency (from now on latency) narrows down the problem to the following definition:

> Network latency is the time taken from the start of exchange an application protocol data unit (APDU) at the application layer of one participating node to the end of the exchange of the same APDU with the application layer of a second participating node. [13]

If this latency is higher than a certain threshold, real-time interaction is at stake. According to Delaney, an application is still noticed as interactive by general users when it has a latency below 40 to 300 ms, depending on the application. However in practical application 100 ms proved to be a realistic value for the maximal tolerable round trip latency. That is the time for transferring input data to a simulation, process it and transfer it back to the visualization. 100 ms seems to be a lot of time in terms of network latency in HPC clusters, where current Infiniband networks have latencies between 2 and 10 µs (see for example [74]. Those 100 ms are the upper bound for all components including transfer and computation of the simulation. Especially for applications with large numbers of components, the central communication server instance might become a bottleneck, but also components that do not work correctly or simply take too much time to compute their results may slow down the whole system. Thus it is important to think about mechanisms to prevent the whole system from halting and to improve the scalability to ensure the adherence to the latency threshold. The following sections presents two approaches to tackle this problem.

5.1.5.1 Best Effort and Dropping of Latecomers

The simplest approach for eliminating sources of extraordinary latency is the rigorous dropping of latecomers. Since there exists a global synchronization instance with a global clock, it is possible to just send the last available result to the receivers instead of the updated computation which arrives after a certain time limit. However, this might lead to incorrect visualization and erroneous analysis of the simulated data. A more failsafe approach is to handle latency at a best effort basis and ban producers or notice the users if a certain component extends the threshold more often than a specified times. Thus, both methods will not help if the communication server itself is the source of the latency, e.g. because it is not able to handle all the connected components in time.

5.1.5.2 Clustering of Communication Servers

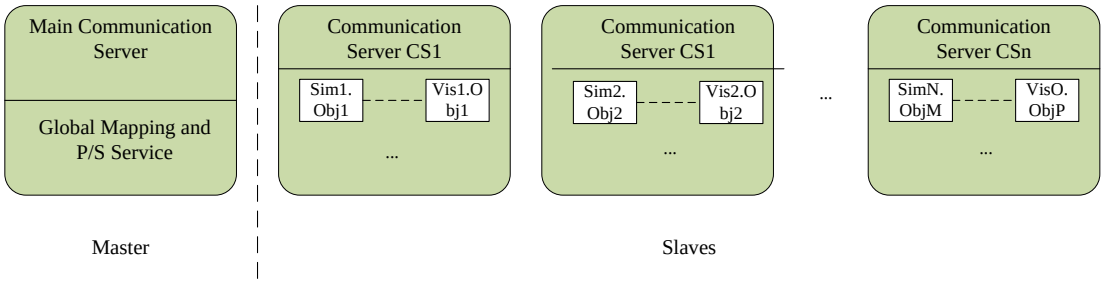

Figure 5.11: Clustering of multiple communication servers according to master slave approach.

To eliminate the bottleneck of a central communication instance a concept for the clustering of communication servers is developed. Figure 5.11 depicts the basic idea behind the clustering. There still exists a master communication server which holds the global map and is interfaced with the P/S service. Additionally there are n slave communication servers to which the master evenly assigns communication links. This method can be compared to the striping of data in RAID systems. The master keeps track of the assignation of communication links and informs the connected components to which and from which of the slaves they send and receive their data. The master tries to assign 1xn links to the same slave communication server to avoid segmentation of those links. However if this is not possible a component has to send outgoing data for the same object to two or more slave communication servers. By clustering the communication servers it is possible to much better scale the system. Even dynamic scaling is an option, where the master launches new servers if it recognizes that a certain load threshold is reached. As the actual communication goes through the slaves and no longer through the master, it can fully concentrate on assigning communication links and synchronizing the slaves. To avoid allocating new nodes for the slave servers it is also possible to start them on simulation or visualization nodes. Especially, if for example one simulation sends a lot of information to one communication server slave it makes sense to put that slave directly on the simulation's machine. The actual computing load of a communication server is insignificant in comparison to the communication load. All this functionality needs to be transparently encapsulated by the steering library.

5.2 Prototype - The CSIS Framework

In the following the implementation of the framework for computational steering of interactive simulations (CSIS) is briefly described on the basis of the three main components:

1. Communication server for the distribution of volatile state data.

2. A publish/subscribe server for the dynamic mapping of the volatile state data and the passing of static parameters.

3. An API which offers the frameworks functionality to applications.

The following sections will introduce the two supporting technologies used for implementation of the CSIS framework: the Commuvit communication server and the D-Bus for publish/subscribe services, and describe the necessary adaptations. Afterwards, the focus lies on the actual steering library and the introduction of the API used to implement actors of a steering application. For further implementation details please refer to [17].

5.2.1 Commuvit

The transfer of volatile state data between actors in the steered application, according to the communication map, is realized with the *Commuvit* communication server (see [5]). It was initially designed for the real-time execution of distributed simulations of mechatronic systems and fulfills the requirements for the transfer of volatile state data: The simultaneous and synchronized transferring of data from sources to drains. After a brief architectural overview, the adaptations that were made for the CSIS framework are described.

5.2.1.1 Architecture

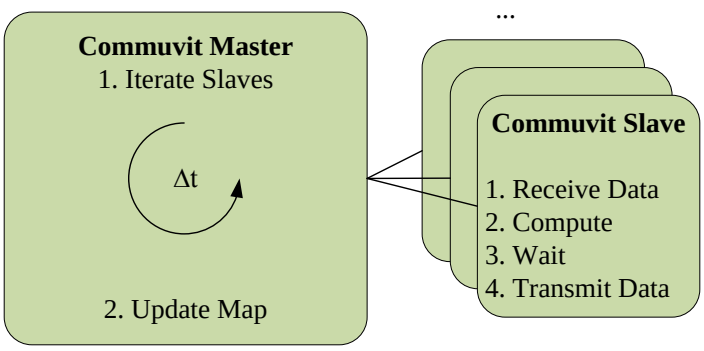

Figure 5.12: Connection handling by the Commuvit server.

The core of the Commuvit communication server is its centrally managed simulation time and a global variable map. The main loop of the Commuvit server steps the time of the simulation with a configurable time Δt in each iteration. The map stores the volatile state variables for each producer and assigns them to the subscribed consumers. As shown in figure 5.12, Commuvit employs a master thread and one slave thread for each connected component; these are employed to synchronize the components of a distributed application. During a single iteration, the master thread instructs its slaves to process the following steps:

1. transmit the component's input data from the map to the component.
2. instruct the component to start its computation.
3. wait for the computation to finish.
4. copy the output data (results of the computation) into the internal map.

The mapping, which specifies how data is transferred from and to the different tools, originally was loaded from a static configuration file at startup. Through the integration of the publish/subscribe service it is now possible to alter this map dynamically.

5.2.1.2 Variable Mapping

The variable mapping is the heart of the Commuvit server. It is stored in a configuration file as shown exemplary in listing 5.13. The server uses this initial map to create all variables and their bindings. The variables have unique names and can be grouped hierarchically. In the example simulation `sim1` offers one car `car1` which has a pos object with 3 volatile state variables `x`, `y` and `z`. Those variables are mapped to two visualization instances `vis1` and `vis2` which also have interfaces for a `car` object with a `pos` object consisting of `x`, `y` and `z` variables. From now on in every global time step, the values in `x`, `y` and `z` are copied synchronously from `sim1` to `vis1` and `vis2`.

```
# vis1
# car1
Map=sim1.car1.pos.x, vis1.car1.pos.x
Map=sim1.car1.pos.y, vis1.car1.pos.y
Map=sim1.car1.pos.z, vis1.car1.pos.z

# vis1
# car1
Map=sim1.car1.pos.x, vis2.car1.pos.x
Map=sim1.car1.pos.y, vis2.car1.pos.y
Map=sim1.car1.pos.z, vis2.car1.pos.z
```

Figure 5.13: Commuvit map for one simulation and two visualization instances.

5.2.2 D-Bus

Where Commuvit is ideal for fast and low-overhead transfer of volatile state data, D-Bus [21] is used to implement group-shared state parameters and the dynamic generation of Commuvit's variable map. D-Bus is used by many Linux applications, such as the Hardware Abstraction Layer (HAL), CUPS, the Gnome Power Manager, or Beagle. Support for the Windows platform is currently under development. It is a very powerful architecture which can be used for several purposes. A good basic description is given in [21]. The D-Bus framework itself actually consists of two major components. On the one hand, at the protocol level, the D-Bus library can be used to implement peer-to-peer remote procedure calls. On the other hand, the framework provides the `dbus-deamon`, a message bus daemon, which acts as a router for messages. Moreover, the message bus allows the monitoring of registered objects and their methods and signals. An object in D-Bus is identified by an object path, e.g. `/de/upb/commuvit/map`. One important property of D-Bus is that it sends messages to objects, not to applications. Thus, applications in D-Bus can register multiple objects. The functionality of

the major part of the D-Bus framework can also be found in many other RPC frameworks. There are, however, two reasons for the application of D-Bus for distributed communication in the steering library. First of all, D-Bus is lightweight and widely available as part of the major Linux distributions. Secondly, the D-Bus framework can be employed as an easy-to-use publish/subscribe system. As any object can subscribe to signals with a lightweight subscription language (string matching), it is possible to implement publish/subscribe communication with D-Bus. The signals in D-Bus make it possible to implement the required group-based communication mechanism for the framework. If, for example, there are several nodes driving a tiled visualization wall, all of these nodes subscribe to an object `/de/upb/steering/tiled-wall` and receive all signals that are addressed to that group. However, the best scalability and performance could likely be achieved by implementing the publish/subscribe module completely from scratch and design it to the special requirements of CS for IS/VR. This, however, could not be done in the limited scope of this prototype and is scheduled for a more advanced implementation of the framework.

5.2.3 The CSIS Server - Integrating Commuvit and D-Bus

The CSIS server in fact is an extended Commuvit server, which incorporates the existing systems with the messaging functionalities of the D-Bus system. All the changes were implemented according to the concepts described in Section 5.1.3.3. In order to facilitate dynamic addition and removal of producers of volatile variables, the variable map has been extended by two remote procedure calls: `add(string var_name)` and `subscribe(string var_name)`. The first method is used by the D-Bus library to add new volatile state variable to Commuvit's internal mapping, whenever a producer announces new volatile variables. The latter method is used to register a consumer's interest in some variable. If this variable is not yet available the request will be queued, otherwise a mapping is created between the producing actor and the consumer.

The second adaption that had to be made was a notification and monitoring interface. Upon a call to the `add_output(string name)` method, the CSIS server has to notify existing actors of the new variable to allow them to subscribe to this variable. This is realized through a notification interface between the communication server and the publish/subscribe service. In the opposite case, i.e. if the consumer of a variable joins after the producer, a method for initially receiving all available volatile state variables has been added. With these implemented changes, the CSIS server can now be started with an initially empty map. Through the connection of producing and consuming actors, started by the user, the map will successively be filled.

The passing of shared parameters is handled by the D-Bus system and a dedicated map is stored in the CSIS server. In contrast to volatile state variables, which are transferred in every step of the global clock between all subscribed actors, shared parameters are only transferred when they are changed. This is a basic feature of publish/subscribe systems, whenever an actor subscribes to a shared parameter it receives every update until it unsubscribes from it. This functionality, as well as the adding and

subscription to volatile state variables, is offered through the CSIS steering library.

5.2.4 The CSIS Steering Library

Instead of going into details of the libraries internal implementation, a more practical description of the framework's API will be given. The actors of a computational steering application (e.g. visualization, simulation) will need to be implemented against the steering API. Through the framework, an actor can achieve the following objectives: 1. receive or transmit volatile state data. 2. share parameters among a group of actors. 3. decide on the subscription to newly added volatile variables. 4. initiate internal function calls on a parameter change. In order to develop an actor for computational steering, the general approach is to declare all produced volatile variables, as well as subscriptions to volatile variables and parameters to the framework. Furthermore, the developer is able to associate callback functions with certain parameters. All in all, the framework's API provides the following basic functions:

`connect(char* host, int c_port, int d_port)`:
Connects to a host with the Commuvit server and the D-Bus service on ports `c_port` and `d_port` (client-side).

`iterate()`:
Must be called in every iteration of the component/actor. Calling this function will send and receive outstanding notifications through the D-Bus and synchronize the component with the COMMUVIT server.

`reg_var(char* name, double* var)`:
Registers a new volatile variable in the COMMUVIT server. In every `iterate()` call to the framework, the value of `var` will be transferred to consumers of this variable via Commuvit.

`subscribe_var(char* name, double* var)`:
Subscribes to the volatile variable "`name`". The location `var` will be updated with the variable's new value in every iteration.

`set_var_filter(void (*filter_func)(char* var_name))`:
Sets a callback function, which will handle notifications of newly available volatile state variables. Thus, a component can decide whether or not to subscribe to such a new variable.

`reg_parameter(char* name, int (*cb_func)(char* var_name, GValue* val))`:
Registers a parameter with the framework. Updating the value of the local copy of this parameter is a task of the specified callback function. Thus, it is possible to implement calls to a component's internal functions on the change of a parameter.

`reg_parameter(char* name, GValue* value)`:
Registers a parameter with the framework. The parameter is automatically updated by the framework.

`update_parameter(char* name, GValue* value)`:
Updates the shared parameter "`name`" with the specified GValue.

`query()`:
Returns a list of all available producers of volatile state data and static parameters which can be used to manually or automatically subscribe to the desired variables.

`disconnect()`:
Disconnects the actor from COMMUVIT and the D-Bus service.

By integrating these functions into an arbitrary IS/VR application, it can now be started and executed in a distributed fashion and makes use of the extended models of CS for IS/VR presented in the sections above. Two examples and the benefits thereof are presented in the following.

5.3 Computational Steering of a Distributed Driving Simulator

Since the goal of CSIS was not primarily to improve the performance of certain applications, but to provide the possibility to run them on hybrid cluster systems, the evaluation of this system focuses on describing successful deployments. The first use case of the CSIS framework and the application computational steering paradigms to a IS/VR application is a distributed driving simulator. Its main focus lies in the realistic simulation on automotive headlights but it also demonstrates the possibilities of a distributed and modular VR application. After a short introduction of the original application, the modularization and integration of the CSIS framework is described. Afterwards examples of new functionalities and new scenarios are presented.

5.3.1 The Virtual Night Drive Simulator

The Virtual Night Drive (VND) simulator was introduced by Berssenbruegge et al. in [7] and in [22] and was developed at the Heinz Nixdorf Institute, Paderborn, in cooperation with the Hella KGaA Hueck & Co. Its main focus is the realistic simulation of headlights on real-world tracks to do physiological tests and to let engineers evaluate prototypes of new headlights. There are several characteristics of headlights that need to be considered when designing a simulator. It is nearly impossible to visualize the complex light distribution with traditional computer graphic's lighting and shading methods like point light sources or Phong shading. Instead, a system was developed that takes advantage of programmable pixel and vertex shaders to project the light of

the headlight onto the scene per pixel. This provides a very realistic impression for test people and engineers and allows the differentiation of several types of headlights. It is also possible to check the compliance of the headlights to strict standards in a very early stage of development (e.g. dimmed headlights may not beam over a certain horizon). Figure 5.14 shows a screen shot of the original VND simulator running on one single computer.

Figure 5.14: Screen shot of the Virtual Nigh Drive simulator.

5.3.2 Modularizing the VND Simulator

The original version of the simulator works quite well as long as only one car with headlights is simulated. For more cars the light simulation requires more computational power than a single CPU/GPU workstation can provide. Additionally, it is not trivial to realize effects like blending with the existing shader approach since no real light sources are utilized. But especially for the psychological testing it is extremely important to have more than one car with realistic headlights to simulate effects like oncoming traffic or colons of vehicles which are quite important in everyday life. This led to the idea of distributing the whole system onto several computers and bundle their computational power to provide an interactive system for complex scenarios.

Additionally, the need for various display and computing scenarios arose, for example to be able to drive a stereoscopic wall or tiled displays. Thus, a modularization and a flexible basis for coupling the components was needed. This basis was the CSIS framework. Before the integration of application and framework is described, a quick introduction of the three VND modules simulation, visualization, audio and input is given.

5.3.2.1 The Input Component

The VND simulator supports different types of input classes and devices. For the steering of a simulated car in the virtual world, there currently exist three different input devices:

- Standard keyboard input, which serves only for testing and supervising purposes since it does not simulate a realistic driving experience.
- A steering wheel for gaming, which is cheap and can be easily connected to standard PCs. It provides a basic driving experience, but lacks the haptical feedback of a real driving wheel.
- Different professional force feedback wheels which are amongst others used to evaluate steer-by-wire approaches. They are connected through a standard CAN-Bus interface (see for example [19]) and can simulate the haptic feedback of a real driving wheel.

For all of these devices an input component was developed which implements a special interface. This interface consists of three output volatile state variables: `throttle`, `break` and `steering angle`. Whenever an input component connects to the CSIS framework it publishes these three variables and updates them in every global time step.

Additionally, there is a second input component (which can also be integrated in the first component if needed) to launch special events. Examples for such events are the switching between day and night view, the selection of different headlight types or the resetting of the car's position. All events of this fixed set are published to the system as shared parameters and can be subscribed to by interested consumers. The input component reacts on user inputs (e.g. pressing a specific key on the keyboard to trigger a special event) by updating and publishing the appropriate parameter.

5.3.2.2 The Simulation Component

The simulation component of the VND simulator deals with the dynamic simulation of the cars in the virtual world. It uses the volatile state variables `throttle`, `break` and `steering angle` from an arbitrary input component, to calculate the position and direction of one car object in the three-dimensional scene. There can be different forms of simulation, ranging from simple and not very realistic models to fully detailed rigid body dynamics simulations. The only thing they must provide are the following volatile state output variables: `pos` with the subvariables `x`, `y`, and `z` for the three dimensional location of the car and `dir` with the subvariables `x`, `y`, `z` and `phi` for the direction and pitch angle of the car.

In addition to simulation components that accept input from an input component, there also exist input-less simulation components. Those can steer cars autonomically, possibly by following predefined tracks or implementing scripted or simulated behavior. However, they also have to provide the output variables `pos` and `dir` and their subvariables.

A simulation either subscribes or does not subscribe (if it is autonomous) to one input components volatile state variables and publishes the hierarchical state variables `pos` and `dir`. Additionally, it can subscribe to simulation related events such as resetting the position of a car.

5.3.2.3 The Visualization Component

Figure 5.15: Three VND visualization components used to drive a 3-channel projection.

The visualization component is responsible for the graphical output of the simulation data. In the case of the VND simulator it renders the car objects in the virtual world. Furthermore, it is responsible for the headlight simulation of all displayed cars. A visualization can have an arbitrary amount of car objects and for every car object it accepts one dir and one pos volatile state variable. Those variables determine the actual position and direction of the car as they were computed by a simulation.

Additionally, the visualization component can subscribe to several events, e.g. car-dependent ones such as switching headlights or car-independent ones such as switching day and night. Each visualization component also has one fixed field of view and camera angle. This can be used to create sophisticated displaying setups such as tiled display walls or multiple-channel-projections as for example shown in figure 5.15.

A special form of visualization which tries to overcome the limitation of simultaneously simulated head lights is presented in section 5.3.5.

5.3.2.4 The Audio Component

For the acoustic output of the driving simulation there exists an audio component which can be bound to any car in the simulation. The audio component is also designed as an independent application which receives the same information about the car objects as the visualization component. Additionally, it subscribes to the position of a camera in an arbitrary visualization to be able to generate 3D sound for that specific

generation. This allows for the realization of surround sound setups which help the users to further immerse into the virtual reality.

5.3.3 An Exemplary Setup of the Computationally Steered VND Simulator

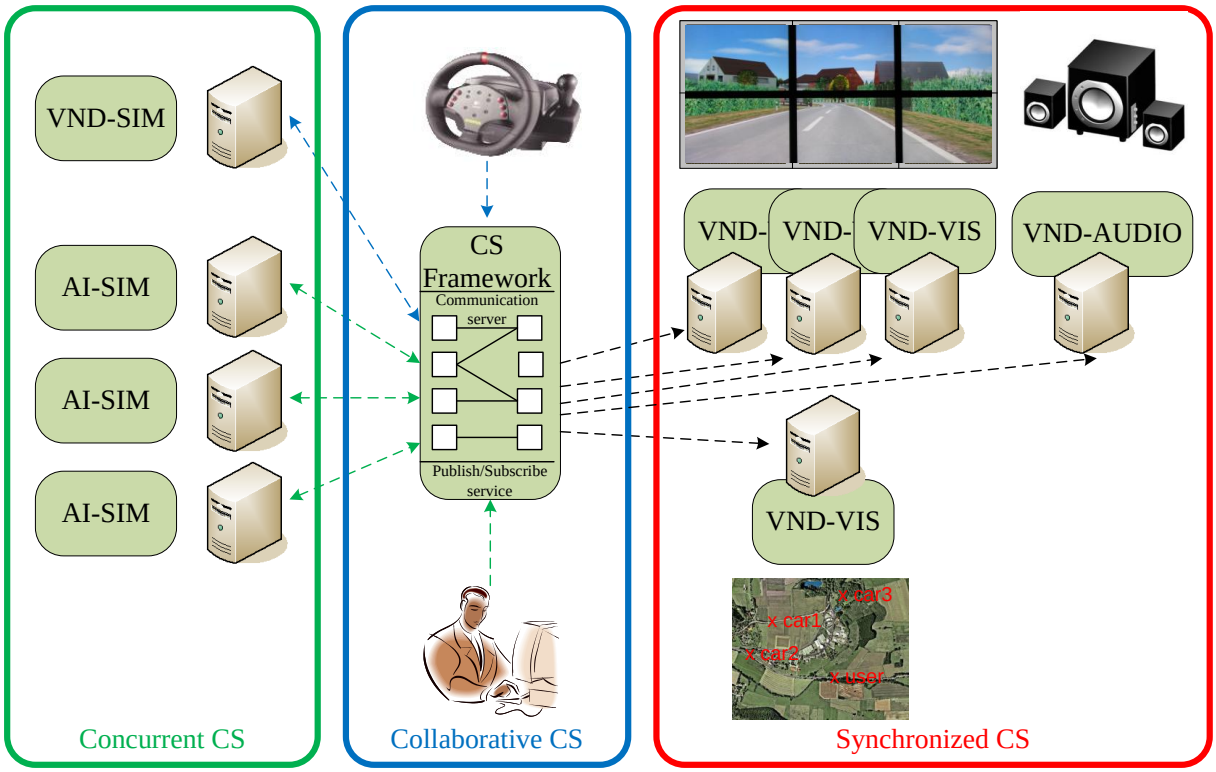

Figure 5.16: Computational Steering of the distributed simulator VND.

Figure 5.16 shows an exemplary scenario of the Virtual Night Drive application where the extended models of computational steering for IS/VR are applied and combined. The scenario that was realized is the following: A user steers one car in the scene which is displayed on a high-resolution tiled wall. To render the images for the 3x2 tiled wall, three different graphics nodes running six instances of the visualization component (VND-VIS) are used. To ensure that every node renders the right view they are synchronized by using *Synchronized CS*. Additionally, an audio component (VND-AUDIO) which receives updates of the camera position of the VND-Vis instances is responsible for the generation of sounds for all cars in the scene. The input signal from the user's steering wheel is processed by an instance of the simulation component (VND-SIM) which computes the position, direction and speed of the steered vehicle. Furthermore, three additional cars are introduced to simulate a populated scenery. They are controlled by simple AI simulations (AI-SIM). These four concurrent simulation threads

(one user, three AI) are coordinated by *Concurrent CS*. Finally, the whole scenario is observed by a supervisor who has a general overview over the scene (also driven by a synchronized VND-Vis component). He can control the environment of the test person and the AI simulated cars. Thus the supervisor and the user interact with the system through *Collaborative CS*.

5.3.3.1 Exemplary Usage of the Framework for the Passing of Volatile State Data

Both the steering data that comes from the input devices and the vehicle-related data (position, direction etc.) of the simulation are obviously volatile. Imagine a new actor (e.g. a spectator) joins the system in the example in Figure 5.16 with an own visualization instance (VND-VIS) that enables him to watch the scenario from a prespecified point. In order to be able to see the vehicles in his instance of visualization he must subscribe to receive the volatile simulation data from all active simulation instances. If this is successful the spectator's VND-VIS receives the volatile data of all cars in the scene (user- and AI-controlled) and is able to display them at the actual position and with the right direction.

5.3.3.2 Exemplary Usage of the Framework for the Handling of Shared Parameters

Analogue to the passing of volatile data, a short example for the handling of shared parameters is given. An exemplary shared parameter is the current type of headlights (regular, Xeon, dimmed etc.). The user or the supervisor can choose it. In order to visualize the right headlights, every visualization instance that displays the corresponding car needs to get informed about changes of that parameter. Hence, a publish/subscribe group is founded and all visualization instances as well as the user's and the supervisor's input instances subscribe to the group. If the supervisor now changes this parameter all subscribers are informed and can update their local parameters and if necessary use them to display the new headlights. If another instance of visualization (e.g. the spectator) joins the systems later, it also subscribes to the group and automatically gets the current parameter status and all future changes.

5.3.4 CS-Enhanced Virtual Night Drive - Conclusion

This sample scenario shows some of the potential of the computational steering framework for a Virtual Reality application. Before the introduction of CS for IS/VR it was hardly possible to realize this scenario without coding the whole application from scratch. Now it is sufficient to integrate the steering-library into the relevant components and declare input and output variables. With that done it is now possible to dynamically arrange various setups and allow for the creation of complex virtual worlds. New modules, for example sophisticated dynamic simulations or advanced steering devices can easily be integrated by defining their interface and dynamically connect them to the system. In addition, various displaying setups can be realized

without making changes to the source code of the visualization. The whole setup can now run on a shared cluster resource such as the Arminius Cluster (see section 2.1.3) and utilize its computational and graphical potential.

5.3.5 Distributed Shader-based Visualization through Computational Steering

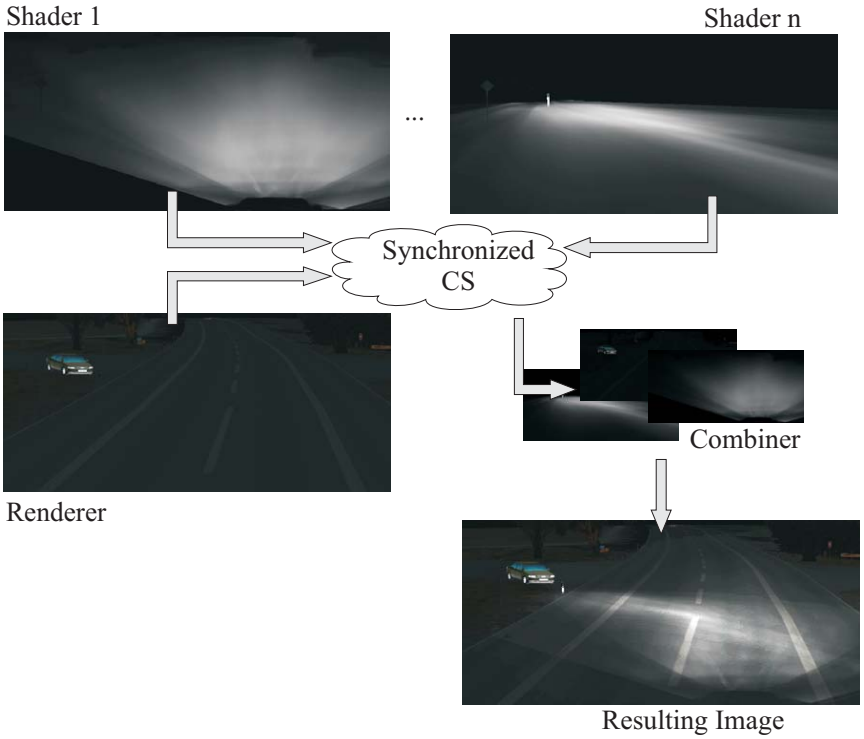

Figure 5.17: Composing synchronously rendered frames.

The framework also enabled new research possibilities in the field of distributed visualization (see [42]). By using multiple nodes for the simulation of headlights and coupling them by synchronized CS it was possible to overcome given limitations of current graphics hardware. In more detail it is now possible to simulate a nearly unlimited amount of cars with headlights only depending on the number of visualization nodes available. All of them render as much lit cars as possible and send these images to one (or more) composer nodes which merges all results to one single view that is shown to the user. Figure 5.17 shows the idea of composing the synchronously rendered frames. Thereby, the Renderer renders the unlit scene and all Shaders do the compute intensive, shader based light simulation. The Renderer and all Shaders have the same view on the scene. Since they are all synchronized through CS, all nodes have the same state of the simulations and thus, a composition of all rendered frames

is possible. This composition is done by the Composer node (cascaded Composer setups are planned to avoid the network-related bottleneck). Benchmarks showed that for 8 nodes speedups between 4-5x could be achieved, when more than 20 headlights are simulated. For more details please refer to [42].

5.4 Computational Steering for an Interactive Material Flow Simulation

A second IS/VR system that was adapted to the CSIS framework is the interactive material flow simulation (MFS) d^{3}FACT insight . It was developed at the Fraunhofer ALB institute in Paderborn and was first presented in [39]. In addition to the traditional tasks of MFS such as planning, safeguarding and improving production processes its main goal is to provide better usability and new methods of exploring the simulated data. The system has a strong focus on the three-dimensional visualization of the simulated material flow in order to give the process and factory designers an insight on how the future systems may look like. After a quick introduction to d^{3}FACT insight this section describes the adaption of the application to the CSIS framework and the resulting benefits. More details can be found in [47].

5.4.1 The d^{3}FACT insight Material Flow Simulation

The d^{3}FACT insight MFS system is split in two parts. The first part offers modeling functionalities and allows the process designer to arrange building blocks (e.g. machines, conveyors etc.) in a two or three dimensional environment. Each of these building blocks has a set of variables which determine their throughput, failure rate, processing time etc.. These can also be set initially in the modeling phase. The model itself is stored in an XML-file and checked against a DTD for completeness and correctness. Thereafter, the model is processed to a Java program by an XSL preprocessor. This Java program serves as input for the actual simulation kernel (the second part) which is also directly connected to the three dimensional visualization. The whole d^{3}FACT insight system is a very sophisticated and complex application and it is out of the focus of this thesis to describe all of its features (including motion planning and various economical models). However, to understand the concept of introducing CS to d^{3}FACT insight and the benefits thereof, the basic components and concepts of this system are briefly introduced in the following.

5.4.1.1 Initialization and Execution of the Simulation Experiments

Before the actual computation of the simulation experiment, an initialization is executed where the simulation is filled with the input data from a simulation database or external sources. In the original version, an experiment manager can manage several

variants of the experiment, so that multiple simulation runs can be computed sequentially or in a distributed, but not interactive mode on different computers. Selected variables and their parameter changes are recorded in each element of the simulation model or are stamped on the tokens, which run through the system. This collected data is saved in the database and analyzed subsequently. Because of the possibility of user interactions during the execution of one simulation model, the parameterizations made by the user are recorded as well. During an experiment, most variables can be viewed and in some cases changed. The analysis of the simulation experiment can be adjusted individually. Some standard analysis and statistics are presented by standard building blocks, available in a modeling library. However, the current approach is limited in flexibility and expendability since for example a comparative visualization of multiple simulation runs is not possible, because the simulation and visualization instances are not connected interactively.

5.4.1.2 Modeling with Buildingblocks and Subblocks

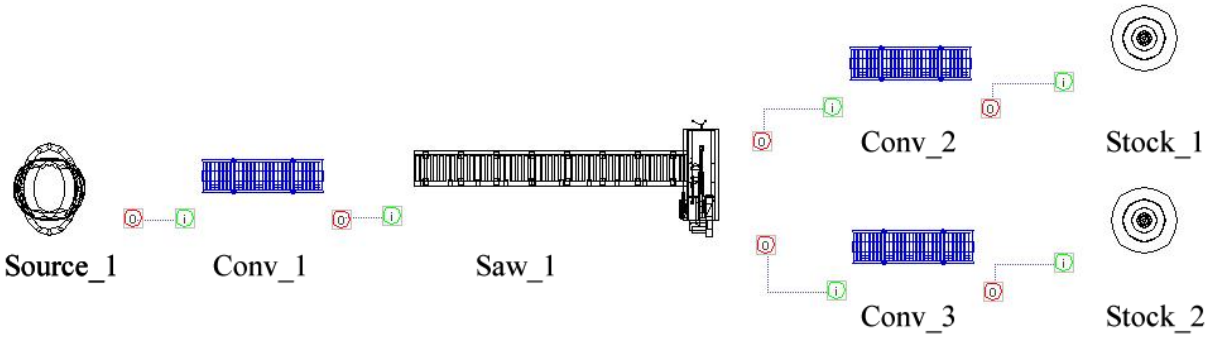

Figure 5.18: Example of an executable simulation model consisting of 7 subblocks.

To better understand how the modeling in d^{3}FACT insight works, figure 5.18 shows an exemplary model in a two dimensional representation. The model consists of four different building blocks: `Source`, `Conv`, `Saw` and `Stock`. Those building blocks can be grouped in libraries (e.g. one library for machines, one for conveyors etc.). Through the definition of variables and event routines, the building block's behavior is described. In order to allow a hierarchical modeling, objects are derived of these building blocks. These objects are called subblocks and inherit all variables, parameters and behavioral descriptions in the event routines from their building block. For each subblock, it is possible to change its actual parameters, but not to change its basic behavior. Subblocks also have several input- and output-channels, so that they can be linked together. In the example in Figure 5.18 the subblocks `source_1`, `conv_1`, `saw_1`, `conv_2`, `conv_3`, `stock_1` and `stock_2` are linked through those channels and thereby define the actual material flow.

5.4.1.3 The Simulation, Tokens and Visualization

The actual simulation follows the principle of a discrete, event based simulation. That means that one or more sources input tokens to the simulation model, which initiate certain events in every subblock they reach. Each of the subblocks also has a processing time, after which a token is passed to an output channel over which it reaches the next subblock and triggers the next event. This process can be simulated in real-time or in accelerated/slowed-down time. The simulation finishes when all tokens reach a sink. The simulation also includes stochastic models to simulate machine failure or the production of junk.

In addition to the simulation-related events, there are also visualization related ones. For example, every time a token enters a new subblock, an animation event is send out to the connected visualization. This event triggers the graphical representation of the token (e.g. a work piece) to move through the current subblock on a specified path. It also holds the computed processing time and other information about the token (e.g. which path inside the subblock it takes). These animation events are the main interface between simulation and visualization. Additionally, the visualization can change certain parameters of the subblocks through events which it sends out to the simulation kernel.

5.4.1.4 Communication Limitations

A general multi-user approach for d^{3}FACT insight should allow the users to collaborate and interact cooperatively on one or more simulation runs at the same time. This directly leads to the demand of an efficient communication mechanism for the message transmission from the simulation kernel to the connected visualization modules and back. The transmission of the events between simulation and visualization needs to be regulated by a central instance. In the original implementation, the communication is processed by the simulation kernel itself. Especially for complex simulation models and multiple connected users, this leads to a significant slow-down of the simulation run, since a direct connection had to be established for all users. If the user, moreover, wants to switch between several distributed simulation kernels, this could only be done with an unacceptable overhead for the initial information processing, which could lead to an intermediate stop of the simulation run. This could be caused for example by a newly connected user, who demands all current data from a running MFS kernel. In order to avoid such peaks of communication and to reduce the growing amount of messages new ways to orchestrate the complex system needed to be found. Computational steering of IS/VR offers powerful techniques to fulfill this task.

5.4.2 Computational Steering in d^{3}FACT insight

Figure 5.19 shows the basic idea behind the integration of CS into the distributed MFS d^{3}FACT insight. Instead of a direct connection between every visualization and every

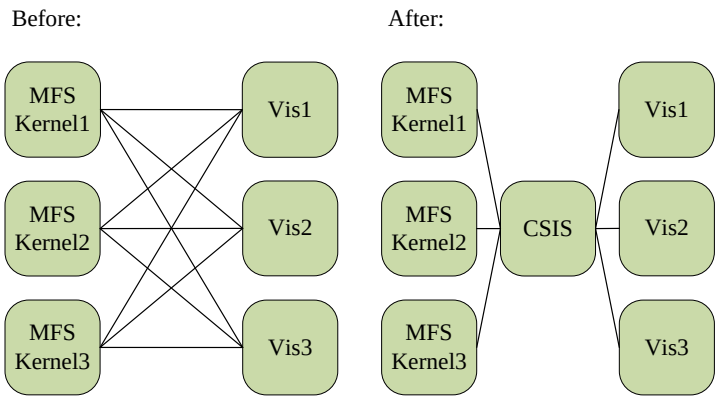

Figure 5.19: Replacing direct communication links by a centralized CS component.

MFS kernel, a central CS instance is established which orchestrates the whole message transmission. Thereby, virtually every component can exchange data with any other component. Events are exchanged by changing the values of external parameters. These are initially made public through the publish/subscribe system, where clients can subscribe to selected parameters. The following describes the functions of the main components (MFS kernel and visualization), the interfaces and the data exchange itself.

5.4.2.1 Adapting the MFS Kernel and the Visualization

Instead of sending the initialization events directly to the connected visualizations, the MFS kernel publishes them once through the P/S service of the CSIS framework. Thereby, all basic information about the used subblocks, e.g. their 3D-representative, their position and their attributes including all actual parameter values is announced. Afterwards, the simulation kernel waits for an external signal to start the simulation run. Visualizations that connect to the CSIS framework can now request a list of all connected simulation kernels and subscribe to their events.

During the actual simulation the kernel sends animation events for each subblock to the CSIS framework in every step of the global clock. The organization of this event exchange is described in section 5.4.2.3. The visualizations, however, can subscribe to these events and receive their updates synchronously. These messages are then used to animate the representations of the tokens in the three dimensional model. In addition to these volatile variables the visualizations can also subscribe to the parameters of an MFS kernel or selected subblocks. They are only updated through publish/subscribe messages when they are changed.

The visualization itself provides several interaction mechanisms. For example, by selecting a special machine or element from the 3D-environment, all of its properties are subscribed to automatically and presented to the user in an additional panel of the visualization client. From then on, every update is propagated to the client, so that the user always has all actual data about this selected element, e.g. a machine or

a forklift. If admitted, the user is able to change parameters in the properties-panel, which are transmitted back to the kernel through the CS framework and might change the behavior and thereby the animation in the 3D-view.

The 3D-client also allows tabbed displaying of multiple properties-panel (e.g. for multiple MFS kernels). Additionally, the client allows to switch between different simulation runs, which are computed simultaneously on a computer cluster. By switching the simulation run, the correspondent data is transferred to the client, including all parameters and parameter updates. Further details about the assignment of simulations and visualization are given in section 5.4.2.3. Besides switching between several simulations, an aggregated view of two or more simulations can be displayed to directly compare the computed results. To support the user in distinguishing between the different simulations, corresponding elements of each simulation, e.g. work pieces that travel through the model, can be colored uniquely.

5.4.2.2 Interfaces

To publicly announce its building blocks, a MFS Kernel uses the P/S Interface of the CSIS steering library. For every block, one message is send to the CSIS server containing details like available input and output parameters. With this information the CSIS service generates an internal list of all connected kernels and their corresponding building blocks. This list can be acquired by potential clients and allows them to choose to which kernel and which of its subblocks a connection should be established. This is done by the client's P/S interface which subscribes to every parameter according to the user's selection. After this step the CSIS server invokes an update of the global map and connects input and output parameters according to the subscriptions of the newly connected visualization.

Currently the system only works demand-driven. That means a visualization has to query for newly added kernels or for the whole list of available kernels. Another option would be to announce the new arrival of a kernel to all clients and offer the user possibilities to integrate that kernel into the running visualization. Before the actual visualization is started, the client offers a GUI to select to which MFS kernel(s) and which of its building blocks it should connect to. This information is acquired through the P/S Interface of the CSIS system. This interface is also used to query for new MFS kernels when the visualization is already running. The result of this query again is displayed in the clients GUI and shows all currently available MFS kernels and their subblocks.

5.4.2.3 Data Exchange

The actual data exchange between a subblock in the simulation and its representation in the visualization is handled by instances of an extension of the CSIS steering library called moderators. This extension was introduced to represent the hierarchical structure of the building blocks and subblocks in d^{3}FACT insight. There is one central

moderator for each simulation- and visualization component. Each moderator gets a unique name for identification. Additionally, one sub-moderator is assigned to each subblock of the model. The sub-moderator offers the local data of the subblock to the central moderator. For unique identification, each sub-moderator also has a unique name - the name of the subblock. The moderators at the simulation collect all animation messages which trigger the animation of the tokens (e.g. packets on an assembly line) in the visualizations. These messages contain a processing time-interval, simulation outputs and configurable parameters (properties) and are always bound to one specific instance of a subblock. The submoderator at the visualization (of one subblock) offers the complementary interface to receive simulation data and for changing simulation parameters.

The inputs and outputs of a subblock within a simulation look like this:

```
Inputs:
<simulator-name>.<subblock-name>.property1
<simulator-name>.<subblock-name>.property2
...
Outputs:
<simulator-name>.<subblock-name>.animate_token
<simulator-name>.<subblock-name>.animate_start
<simulator-name>.<subblock-name>.animate_stop
<simulator-name>.<subblock-name>.output1
<simulator-name>.<subblock-name>.output2
...
```

The labels `property` and `output` are only placeholders and are replaced by subblock specific names like `throughput` (output) or `manufacturing-time` (input). This depends on the functionality of the subblock.

The moderator and its sub-moderators of the visualization are able to receive the output of multiple simulations. They also allow the parameterization of a single subblock or a set of different instances of the same block in different simulation kernels. To support multiple inputs from different kernels, the input names are extended by an identifier as follows:

```
Inputs:
<vis-name>.<subblock-name>.<instance-id>.animate_token
<vis-name>.<subblock-name>.<instance-id>.animate_start
<vis-name>.<subblock-name>.<instance-id>.animate_stop
<vis-name>.<subblock-name>.<instance-id>.output1
<vis-name>.<subblock-name>.<instance-id>.output2
...
```

This set of inputs is defined with different values for the `instance-id`, according to the numbers of simulation instances, which are observed. The values of outputs correspond to the simulation inputs, but can be mapped to one or multiple inputs of different simulation kernels:

```
Output:
<vis-name>.<subblock-name>.property1
<vis-name>.<subblock-name>.property2
```

This naming is for example used to evaluate one material flow model under different conditions, by starting several runs with different initial parameters (e.g. variable processing or failure rates for selected machines). Each of these different scenarios is executed in a separate simulation kernel. Through the utilization of the CSIS framework, the kernels can be executed in a distributed fashion for example on a hybrid cluster. The simulations run on the cluster's compute nodes and send their output directly to the connected visualization nodes where the user can watch and steer the simulations. This allows for the efficient analysis of material flow simulation by aggregation of simulation results from different kernels and direct comparison between two (or more) simulations runs with different initial parameters.

For the initial generation of the moderators (the extensions of the CSIS steering library), the components (simulation and visualization) parse the XML description of the MFS model. Each instantiation of a subblock creates a corresponding sub-moderator. The sub-moderator assigns the input- and output data to the central moderator. For the scenario depicted in figure 5.18, the list of prefixes for moderated subblocks of the simulation kernel `KernelScenario1` will look like this:

```
KernelScenario1.Source_1.<variables>
KernelScenario1.Conv_1.<variables>
KernelScenario1.Saw_1.<variables>
KernelScenario1.Conv_2.<variables>
KernelScenario1.Conv_3.<variables>
KernelScenario1.Stock_1.<variables>
KernelScenario1.Stock_2.<variables>
```

The connection between a subblock in simulation and its representation in visualization is established through the CSIS steering library itself. If the visualization of a subblock `VisA1.Saw1.1` (including ID for Kernel) subscribes for `KernelScenario1.Saw1`, the CSIS server will generate the map entries, in order to pair the corresponding inputs and outputs of both components and establishes the continuous data exchange.

5.4.3 Example Scenario

A small example scenario shall help to understand the potential benefits. Imagine a discussion about the acquisition of a new machine for sawing within an existing factory layout. Figure 5.18 shows an exemplary setup of a production line including a sawing machine. With existing MFS systems it was only possible to simulate models like that with different initial parameters (such as conveyor speeds, capacities, failure rate etc.) sequentially, by starting new simulation runs with new parameters for every possible permutation. With the proposed CS-enhanced system one can start several simulation runs simultaneously and see the results in one combined visualization on a hybrid

cluster system. This is achieved by replacing the fixed mappings of the parameters between simulation and visualization in traditional MFS systems through a dynamic mapping with a communication server as described in section 5.4.2.3. The new solution also offers the possibility to couple the material flow simulation with advanced visualization setups such as a stereoscopic projection, in order to allow an immersive view on several variants. Additionally, a 2D frontend for a simulation expert's laptop can be connected to the same simulation runs, in order to allow interactive variation of the simulation's parameters and by that, the iterative refinement of the simulation model itself. Through the flexibility of the CSIS system, it is possible to start additional simulation scenarios, for example for alternative machines and thus allow a fast switching between the simulation runs for an efficient discussion, which machine is best to buy. Figure 5.20 shows the schematic setup of such a scenario: The parameters of two different MFS kernels are dynamically mapped to three visualization instances, of which two are connected to a stereoscopic display wall and the other one to a supervisor's display. Again, the extended models of computational steering for IS/VR can be found in this example.

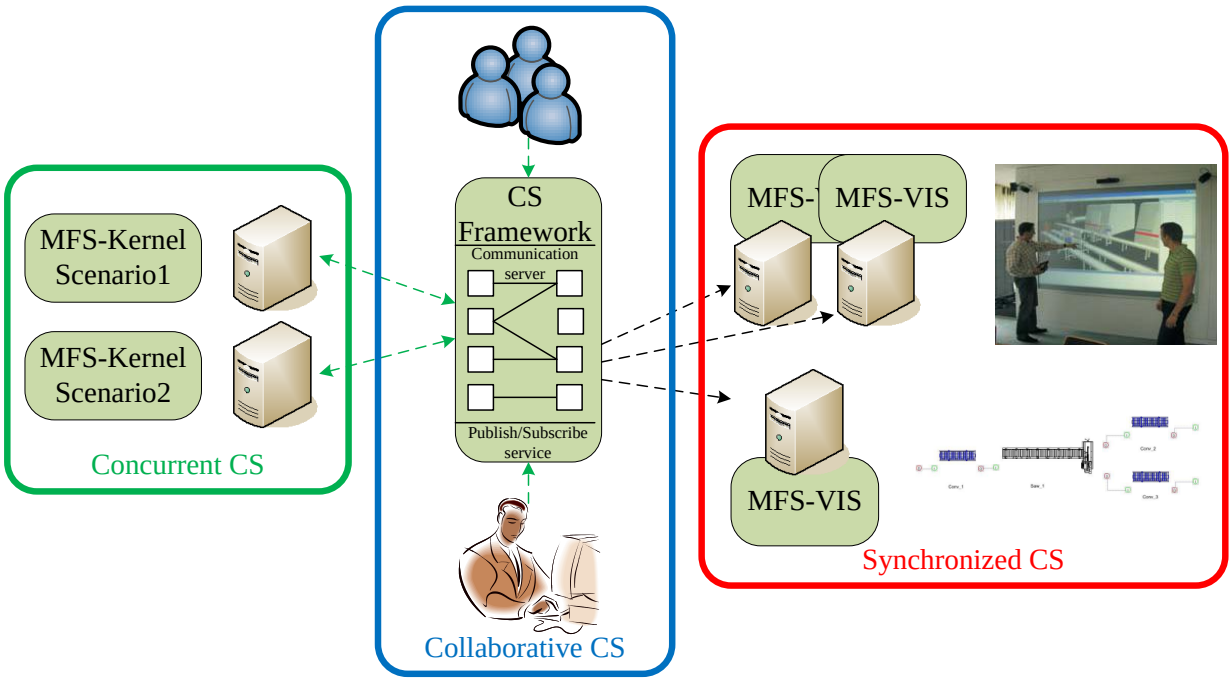

Figure 5.20: Computational Steering of the interactive material flow simulation d³FACT insight.

5.4.4 CS-Enhanced d³FACT insight - Conclusion

The example scenario above just shows the basic new features of the CS-enhanced system, but helps to understand what becomes possible now. For every day use, the

system offers good flexibility and scalability and allows to do more extensive simulations than before. Mainly the possibility to do comparative simulations in real-time is a big benefit for the users of such simulations, since it allows them to compare different scenarios online. For the practical usage of the CS enhanced material flow simulation, the system needs to be adapted to the user's needs and enhanced by methods for security and reliability. By extending the CSIS framework by the hierarchical moderators, a 1 to 1 mapping of the data model of d^{3}FACT insight could be achieved and thereby only few changes in code had to be made to make use of the hybrid cluster as a powerful computing resource.

5.5 Conclusion - Computational Steering of IS/VR

As described in chapter 4 the computational steering of IS/VR on hybrid clusters differs in many ways from traditional CS of HPC simulations. Thus, it was necessary to introduce a novel approach that specifically addresses the main needs of IS/VR applications: interactivity and flexibility. The framework that was presented in this chapter fulfills the requirements which were outlined in section 4.3.

- It serves as a flexible basis to connect and orchestrate arbitrary amounts of distributed components of IS/VR applications. This was achieved by developing models for different usage scenarios and designing a client-server based system that combines a centralized communication server and a flexible Publish/Subscribe system for the data exchange.

- Transparency was achieved by providing the developers with a slim API which encapsulates the basic functionalities of the framework and allows for a comfortable integration into existing applications. The developers and the users do not need to care about data transfer and the connection between the components, they just have to declare the external data and dynamical define the connections between their components to create the scenario they desire.

- All components of the IS/VR application (input, visualization, simulation and additionally audio) can be integrated the same way, by using the frameworks API. This even allows the coupling of different IS/VR applications if exchangeable data is available.

- The whole system bases on freely available software and utilizes well known and approved mechanisms for data exchange, synchronization and group formation (e.g. TCP-based communication, global clock synchronization and publish/subscribe message passing). The prototypical implementation proved the concept and will be freely available over the web site of the PC^2 [65].

Since performance measurement or benchmarking is hardly possible for systems that aim to allow the connection of arbitrary, distributed components, it stands to reason to

evaluate the framework with practical examples. This was done in section 5.3 and 5.4 and the results show that by utilizing CS to IS/VR applications it is, for the first time, possible to flexibly and dynamically build various distributed scenarios of an existing application on hybrid clusters. This enables the users and developers to broadly extend the functionalities of their application (for example through comparative simulation runs or multiple user simulations) without the need to completely rewrite or redesign their application.

6

Remote Visualization for IS/VR

With today's ever increasing computational power and the possibilities to simulate and visualize more and more complex systems, it is obvious that efficient technologies are needed to make the results of such simulations and visualizations available for a broad audience. Remote rendering and remote visualization (RV) are techniques to fulfill this need for visual data. Since the early 90s, researchers work on the topic of transporting the rendered images or visual data to the clients of remote users to let them analyze and work with this data. There are many systems that focus on several different aspects of remote rendering / visualization. Some allow comfortable administration of remote servers and others focus on interactively showing remote users images that where rendered on powerful workstations (for a classification and exemplary systems see section 3.2). But all of those systems have to deal with vast amounts of data that has to be transferred. This is because visual data is mostly pixel or voxel based. Therefore even single images which are displayed for only a fraction of a second consist of one or more Megabytes of data (e.g. 5.76 MB for UXGA 1600 x 1200 RGB with 3 bytes per pixel). To provide a smooth animation, 20 or more frames[1] are needed every second. All have to get from the GPU of the server over the network, to the client and finally to the display of the user. There are already many approaches to minimize the amount of data that needs to be transferred, for example through prefetching certain data to the client or by introducing level of detail mechanisms and compression of all kind (see for example [37]). But there is still no solution on how to enable users to interact with a remote system as if it was local.

As described before, the remote access to universal hybrid cluster systems is of great importance since they often reside in surroundings that are inaccessible for the users. In order to utilize remote visualization for IS/VR on hybrid clusters it is extremely important that a certain level of interactivity and responsiveness (see section 5.1.5) is guaranteed. Additionally, the quality of the remote frames needs to be close to the

[1]The European PAL standard specifies 25 Fps, the American NTSC standard even 30 Fps.

	slow compression for high resolutions	reading data from the GPU fast	multiple render targets
faster host to graphics hardware interfaces	no	++	+
programmable graphics hardware	++	+	no
new framebuffer concepts	no	+	++

Figure 6.1: Problems of RV for IS/VR, possible solutions and their impact on the problems.

original and thus heavy compression is not an option. The available bandwidth of typical, external cluster connections is high but not unlimited (about 1-5 Mbit/s per user), which is why the compression methods need to be adapted to this perquisite. And, last but not least, a framework is needed that is flexible enough to adapt to the needs of a hybrid cluster system. That is, for example, supporting the aggregation of multiple frames from multiple visualization nodes or support multiple users at a time. None of the existing systems presented in section 3.2 completely fulfill these needs. Many of the system simply are not designed for strict interactivity and are excluded because of high latencies (e.g. 200+ ms for most of the systems for remote administration). In contrast, the systems optimized for remote 3D-applications mostly are not flexible enough or require too much bandwidth. Thus, the following section points out three main problems that need to be solved to efficiently make use of remote visualization in combination with IS/VR. Thereafter, a platform is introduced which allows for the testing and implementation of solutions to the problems described in section 6.1 and several key technologies and their application are described. Amongst others, new parallel compression techniques that make use of the computing power of modern GPUs to achieve a fast and high quality image compression are presented in section 6.2. Section 6.4 finally describes the prototypical realization of those methods and the framework itself. Finally, section 6.5 manifests the findings with benchmarks and a comparison to an existing system.

6.1 Limitations of Remote Visualization for IS/VR

The table in figure 6.1 shows the three main limitations that hinder the effective utilization of RV for IS/VR. Additionally, possible solutions and their impact on the limitations are marked in the table. The following section describes the problems and possible solutions in detail. The main aim is to minimize the overhead in time that is

needed to compress, transfer and decompress the rendered frames to allow low latencies and undisturbed interaction. Additionally, the quality of the images should not suffer from high compression rates and we assume fast internet or ethernet connections with a bandwidth of at least 1 MB/s to achieve a decent quality of service also for high resolutions (XGA and above).

6.1.1 Slow Compression for High Resolutions

The main problem of RV is that image data gets very big the higher the resolution is (e.g. 5.76 MB for 1 frame in UXGA RGB). So it is obvious that some kind of compression is needed to transfer more than 20 frames per second over the network. The difficult tasks now are to find the right compression technique (there are a lot for still and moving images) and further to find the right balance between quality, compression time and size of the compressed frames. When it comes to choose the right compression technique the main factor is compression speed which directly influences the complexity of the algorithm. A very complex algorithm that produces great compression rates but takes a lot of time to be computed is useless for the utilization in RV. Therefore, simple and effective compression methods are needed. The simplest approaches are lossless counting or comparison algorithms like for example Run Length Coding[2] (see [71].). For certain scenarios (frames with only few colors, big unicolored areas, uniform background as for example in CAD), these algorithms generate great results in unbeatable time. But for other scenarios of RV, such as the transmission of real videos, they are not suited well and can even produce negative compression rates. This is where lossy compression techniques like the popular JPEG compression have their advantages. They achieve quite high compression rates independent of the input i.a. by filtering information which is not or only hardly visible for the human eye. Additional techniques like downsampling and dictionary based compression optimize the compression rates. But the more complex the algorithms get the more time is consumed. As for example the JPEG algorithm in the popular libjpeg implementation takes about 40 ms to compress an image of XGA resolution (1024x768 pixels) on a modern dual core CPU (see section 6.5 for details). That already limits the achievable frame rate of a RV system to 25 FPS without taking grabbing, transmission and decompression of the frame into account. For higher resolutions this rate drops dramatically (about 90 ms for UXGA which translates to 11 FPS).

Since it is not likely that new algorithms for still image compression improve the compression/complexity ratio dramatically, the only chance to realize interactive frame rates for high resolutions is to speed up the computation of the compression algorithms. At the first glance, the CPU seems to be the right choice for that kind of computation since it is supposedly the most powerful component in a computer. But,

[2]Consecutive, similar symbols are encoded as the symbol followed by the length of the row

since graphics hardware became more and more sophisticated and universally programmable through their shader units, it might be a solution to this problem to out source the task of compression to the GPU. This approach is discussed in section 6.2.4.

Another alternative would be video compression methods like the famous MPEG coding [54]. They produce really good compression rates for image streams / videos, but have one major disadvantage for the utilization in RV for IS/VR: Nearly all of them base on JPEG or similar still- image compression and need information of the previous and in some cases of the following frames to achieve compression. This introduces latency, which is in almost all cases bigger than just utilizing JPEG compression. Since the main goal of the RV framework for IS/VR is interactivity, the video compression methods are left out in the following. However, if there are approaches that are fast enough to be applied to this problem in the near future, they can easily be integrated in the framework presented below.

6.1.2 Reading Image Data From the Graphics Hardware

Another issue that limited using RV for IS/VR in terms of speed for a long time was that the rendered images could only be read back from the graphics hardware very slowly. That originated from the initial, asynchronous design of the AGP-Bus which was the standard interface between host system and graphics hardware for nearly a decade. It was optimized to transfer data like textures and meshes to the graphics hardware but not the other way round. However, the introduction of PCI Express for Graphics (PEG) cleared the way to efficiently download rendered images from the graphics card. Additionally, through several extensions to the graphic APIs (OpenGL [35] and Direct3D [52]), which allow access to dedicated memory areas on the graphics card, the grabbing of graphical data from the cards becomes easier and more flexible.

6.1.3 Rendering to Multiple Targets

This limitation is important for the efficient usage of RV on hybrid clusters. Since cluster computers are only rarely used exclusively by one user, one has to make sure that it can be shared among several users. The same holds for the visualization components in a hybrid cluster. Thus, the RV system needs to provide its services to more than one client, too. This can be done by enabling multiple render targets, which is possible since the introduction of the aforementioned extensions to the graphics API (frame and pixel buffer objects in OpenGL [31]). With these extensions and the programmability of the graphics hardware one can achieve a flexible and scalable system for RV on hybrid cluster systems.

6.1.4 Programmable Graphics Hardware and GPGPU

The description of the limitations of RV for IS/VR showed that one possible solution for some problems could be programmable GPUs. However, this is a highly unspecific

term and needs some clarification and explanation. Programmable GPUs for professional graphics solutions were first introduced in the early 2000s by companies like SGI [73]. Those GPUs were very powerful at that time, but also very expensive and rather proprietary. In 2001 the first programmable consumer graphics hardware (Nvidia's Geforce 3 [58]) appeared and loosened the strict pipelining concept that was used for consumer-grade graphics processing until that time. Through programmable vertex and fragment processors it became possible to manipulate geometries and even single pixels during their processing in the graphics pipeline. This allowed for example to enhance the visual output of VR environments without adding more geometry information (e.g. surface generation through bump mapping [10]). As the graphics hardware became more and more powerful over the years (mainly driven by demanding and complex computer games), developers started to use it for other purposes than graphics processing. This development is known as GPGPU (General-Purpose computation on GPUs, see [24]) and aims to use commodity graphics hardware as powerful coprocessors for certain applications. In the early stages of GPGPU one had to use standard graphic APIs such as OpenGL to describe the general purpose problem to be solved. Data had to be encoded in textures and the programs were written in special shader languages (CG [57] or GLSL[38]). This was often complicated and sometimes impossible, since those APIs and data formats are highly optimized for graphics processing. But the graphics hardware manufacturers reacted and designed APIs that offered access to the powerful graphics hardware in a more universal way. The two driving companies (ATI [1] and NVIDIA [59]) both introduced such (proprietary) APIs around the same time. They share the goal to allow the usage of a GPU as a massive multiprocessor after the SIMD (Single Instruction Multiple Data) principle. That means that a large amount of data is processed in parallel by one instruction, or in other words many threads with identical instructions process different data simultaneously.

For applications that do many similar and independent operations on large data sets this approach can significantly increase the performance. One example for such an application is image processing, where very often operations are performed on single pixels or small groups of pixels. In section 6.2.4 two GPU-based image compression methods are described that make use of the GPGPU API CUDA [60] by NVIDIA , which is also briefly introduced in section 2.3.

6.2 Invire - A Concept for an Interactive Remote Visualization System

In this section, the concept for a new RV system called Invire (INteractive REmote VIsualization) is introduced. It belongs (according to the classification in 3.2) to the third class of remote rendering systems and focuses on high interactivity, maximized performance, quality visualization results and scalability. The system will be a framework to implement, test, evaluate and improve techniques that help to overcome the

problems and limitations described before and will allow the successful and flexible usage of remote visualization for IS/VR applications on hybrid cluster systems. Invire is designed to be the basis for several compression, grabbing and transfer modules and offers the basic functionalities for data transfer and remote interaction. On this platform enhanced compression and grabbing modules are developed and benchmarked against each other and existing systems. This helps to find out which suits best for different tasks and to proof the need for new technologies in this field.

The following sections introduce the architecture and basic components of Invire as well as different approaches to the grabbing, compression and transfer of the rendered frames. The prototypical implementation as well as the benchmarking results are presented in the last two sections of this chapter.

6.2.1 The Architecture

The overall architecture of Invire (as shown in figure 6.2) is kept simple to maximize the remote visualization performance. The system follows the client server concept, which means that the server side handles the grabbing, compression and transmission of the rendered frames and the client side receives, decompresses and displays the data. The server side offers a so called *Invire Plugin* which can easily be integrated into existing OpenGL applications (non-transparent integration). This allows the passing of parameters between the host application and Invire, for example to allow a dynamic adaption of the applications resolution or advanced remote interaction features. A transparent integration of the framework is possible through mechanisms such as preloaded libraries. Chromium [28] for example uses this mechanism to replace the standard OpenGL library by a stub library which intercepts all OpenGL calls. These can be used, amongst other things, to get access to the rendered frames. In both cases of integration, the current OpenGL context is grabbed into a local or graphics card memory and then passed to the *Compression* facility. This part of the software is implemented as modular collection of compression algorithms that can be exchanged arbitrarily. This allows to compare the different methods and eventually combine them to achieve higher compression rates. After the compression of a rendered frame, it is passed to the *TCPServer* where a header is generated. The header contains information about the image size, compression, resolution, etc.. Afterwards, the header followed by the compressed data is send to the *Invire Client*. It receives the compressed frame and passes it to the *Decompression* facility together with the information from the header. After its decompression, the frame is displayed by the client. A separate interaction component registers remote input from the client, serializes, transmits and passes it to the remote application through the server.

6.2.2 Data Transfer

In order to transfer the data (compressed frames) a TCP socket connection is established between server and client. This socket remains active until either client or server

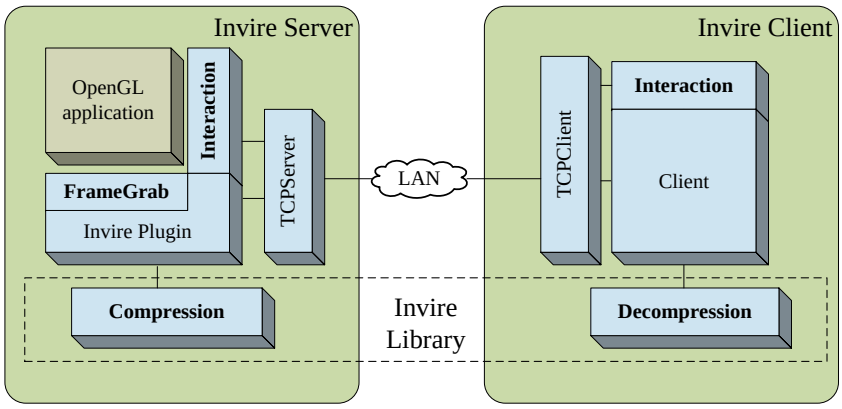

Figure 6.2: The basic architecture of the Invire framework

cancels the transfer. The advantage of TCP sockets is that the correct order and the integrity of the image data is guaranteed. After a socket connection is established, the protocol overhead is minimal and allows a good utilization of the available bandwidth. Additionally, a second socket is established to transmit the events of the interaction component. On the one hand, this helps to prevent additional delay caused by the simultaneous usage of only one socket, and on the other hand logically encapsulates frame transmission and event handling.

The Invire server is designed in a way that it can open multiple sockets for multiple users. This is realized by dynamic socket allocation and the initialization of a new thread for each new client. Additionally the client is prepared to receive data from multiple servers to allow a composition of multiple frames from different servers to for example generate one high resolution image.

6.2.3 Image Readback

To be able to process and transfer the rendered images, they need to be read back from the graphics card to system memory. This can be done by using the graphics API standard calls such as OpenGL's `glReadPixels()`. This function copies the rendered image to local memory pixel by pixel and saves it in a predefined format (e.g. RGBA with 4 bytes per pixel). This method is quite fast and optimized for the transfer of data from the graphics hardware to the host system. However, it is not flexible enough to support multiple render targets (only the main framebuffer can be read back) or copying data to other areas in graphics memory (e.g. for postprocessing by the GPU). These requirements for RV of IS/VR can only be fulfilled by using a more generic approach. In OpenGL Standard 2.1 an extension called Pixel Buffer Objects [31] was introduced which allows the easy and universal access to graphics memory for pixel based objects (e.g. frames or textures). By using these extensions for reading back the rendered frame, one can achieve the desired flexibility and still benefit from the optimized code an official extension provides. More implementation details are presented

in section 6.4.

6.2.4 Compression

As described in section 6.1.1 it is important to find the right compression method for different applications. Since Invire is designed to be a framework that allows the remote usage of arbitrary OpenGL applications and has the goal to provide a basis to improve techniques for RV, a flexible and extendible model for the integration of compression algorithms was needed. This flexibility is achieved by encapsulating all compression related classes in one library and defining universal interfaces for all compression methods. That means, amongst others, that it is possible to execute compression in two ways:

CPU-based: If the user (or the system) chooses a CPU-based compression, the rendered image is grabbed to the memory of the host system in RGB format. This raw image data (i.e. a pointer to its memory location) is passed to the selected compression algorithm. Now the CPU can run the data through the algorithm and again stores the result in local memory.

GPU-based: If a GPU-based compression is selected, the frame is grabbed to a memory location on the graphics card. Then the compression method is invoked by the CPU with a reference to that memory location. The compute intensive parts of the algorithms are encapsulated in so called kernels (for implementation details see section 6.4), which are launched on the GPU and process the data in graphics memory. After that step, the compressed data is read back from the graphics card and ready to be send over the network.

Both methods can also be used in a double (or more) buffered fashion. That means that there are two (or more) dedicated memory areas either in host or graphics memory that hold previous uncompressed frames for comparative compression methods.

The decompression works accordingly, i.e. the compressed data is received and stored either on host or graphics memory. Thereafter, either the CPU processes and passes it to the graphics hardware to display it or the compressed data is directly passed to the GPU, where it is decompressed and directly displayed.

Through common base classes, the library ensures that every compression method offers a compression and decompression function and follows the interface definitions. A special frame object encapsulates all information (header) and data of one frame. It is also possible to implement compression methods both CPU and GPU-based to, for example, compare compression times or allow the decompression on hardware that does not support GPU-based computations. In the next sections, three established compression methods (Run Length Encoding, Difference Compression and JPEG still image compression) are briefly described and two of them (Difference and JPEG) are presented as GPU-based, parallel compression methods. The selected algorithms represent groups of compression algorithms with comparable efficiency and complexity.

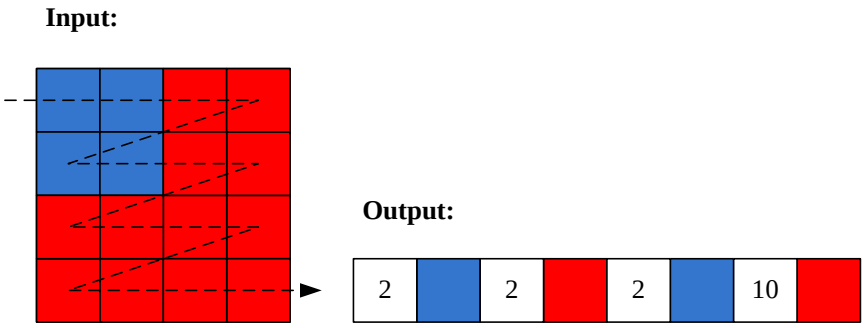

Figure 6.3: Example for Run Length Encoding of an RGB array.

RLE and Difference compression are lossless methods whereas JPEG is a sophisticated, lossy algorithm. This selection had to be done to cover a possibly wide area of compression techniques in the scope of this thesis. As described in chapter 7 there are a lot more possible and promising compression techniques which can be integrated into Invire in further projects.

6.2.4.1 Run Length Encoding

The Run Length Encoding (RLE) is a very simple but in some cases quite effective method to compress arbitrary data. Its basic functionality is shown in figure 6.3. The run-length algorithm goes over the input array (in this case a sequence of RGB coded pixel values), counts consecutive pixels with same color values and writes the sum followed by the actual color information into a new array. The decompression is done accordingly, by writing the amount of pixels with the same color in a new array consecutively. The RLE algorithm can encode and decode n pixels in $O(n)$ time. It is hardly suited for parallel execution since it is heavily dependent on dependencies and partition might revoke the compression gains.

6.2.4.2 Difference Compression with Index

Another basic technique for lossless image compression is the difference compression as shown in figure 6.4. The current and the last frame are compared pixel wise and only the pixels that are different are stored. Additionally, the position of the pixels that changed is needed to decompress the current frame. This can be done most efficiently by an index which maps one bit to every single pixel of a frame. If the bit is 1 the pixel has been changed and the saved pixel value at the position *#of preceding 1s in index* is needed to update the pixel of the last frame. This requires $O(n)$ time to compress and decompress n pixels.

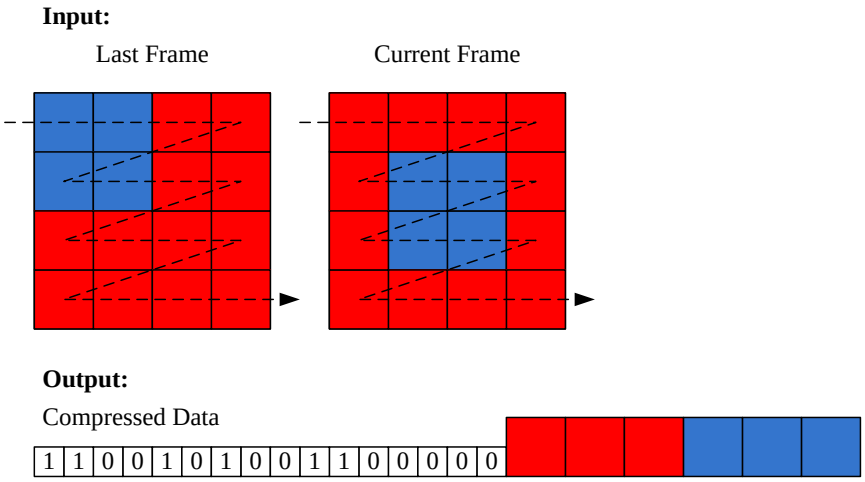

Figure 6.4: Sequential Difference encoding algorithm with index generation.

6.2.4.3 Parallel Difference Compression with Index

The difference compression with index method described before is very suitable for parallel (SIMD) execution. Especially creating the index, as well as copying the pixel data is independent for every pixel and can be computed simultaneously. The only problem is that the amount of pixels (n) is most likely higher than the available threads (k) of a multiprocessor. Therefore each image must be split into $m = \frac{n}{k}$ blocks which can be computed on a multiprocessor.

Figure 6.5 schematically shows the basic sequence of the algorithm. The compression algorithm is divided into three main steps. In the first step, the index is generated by simply comparing corresponding pixels of the last and the current frame. Each thread processes one pixel of a block and writes only the changed pixels to a new memory location (array) with the size of a block (4 pixels in the example). It takes $O(\frac{n}{k}) = O(m)$ time to compute the index and copy the pixel data to this array. When all threads have finished, a parallel compaction method (the second step) is invoked. The basic functionality of this compaction method is shown in figure 6.6. This algorithm requires $O(\log k)$ time to compute the amount of empty spaces (i.e. memory locations that do not hold a changed pixel value) to its left for every item of the local result array. This is done by doing $\log k$ steps and in every step s adding the amount of empty spaces in c_i and $c_{(i-2^x)}$ in parallel. After $\log k$ steps the pixels can be stored in a locally compacted array by calculating their new index with $x_i - c_i$. This algorithm needs to run for all m blocks. Its overall runtime is $O(m * \log k)$. With this information, the pixels can now be stored in a local block without empty spaces. The number of changed pixels is also stored for each block. After all blocks have finished the second step, the information about the changed pixels is used to compute the absolute position in global memory for each blocks partial result (3. step). This is also done in parallel, based on a scan algorithm which sums up all items to the left of the current value, in $O(\log m)$ time.

Basically, it works similar to the algorithm used for local stream compaction. However, instead of adding the amount of empty spaces it adds the numbers of changed pixels in iterative steps. Finally the partial results of the blocks are copied to the calculated memory locations and, together with the index and optionally an array of the numbers of changed pixels in each block, form the final compressed frame. The whole algorithm

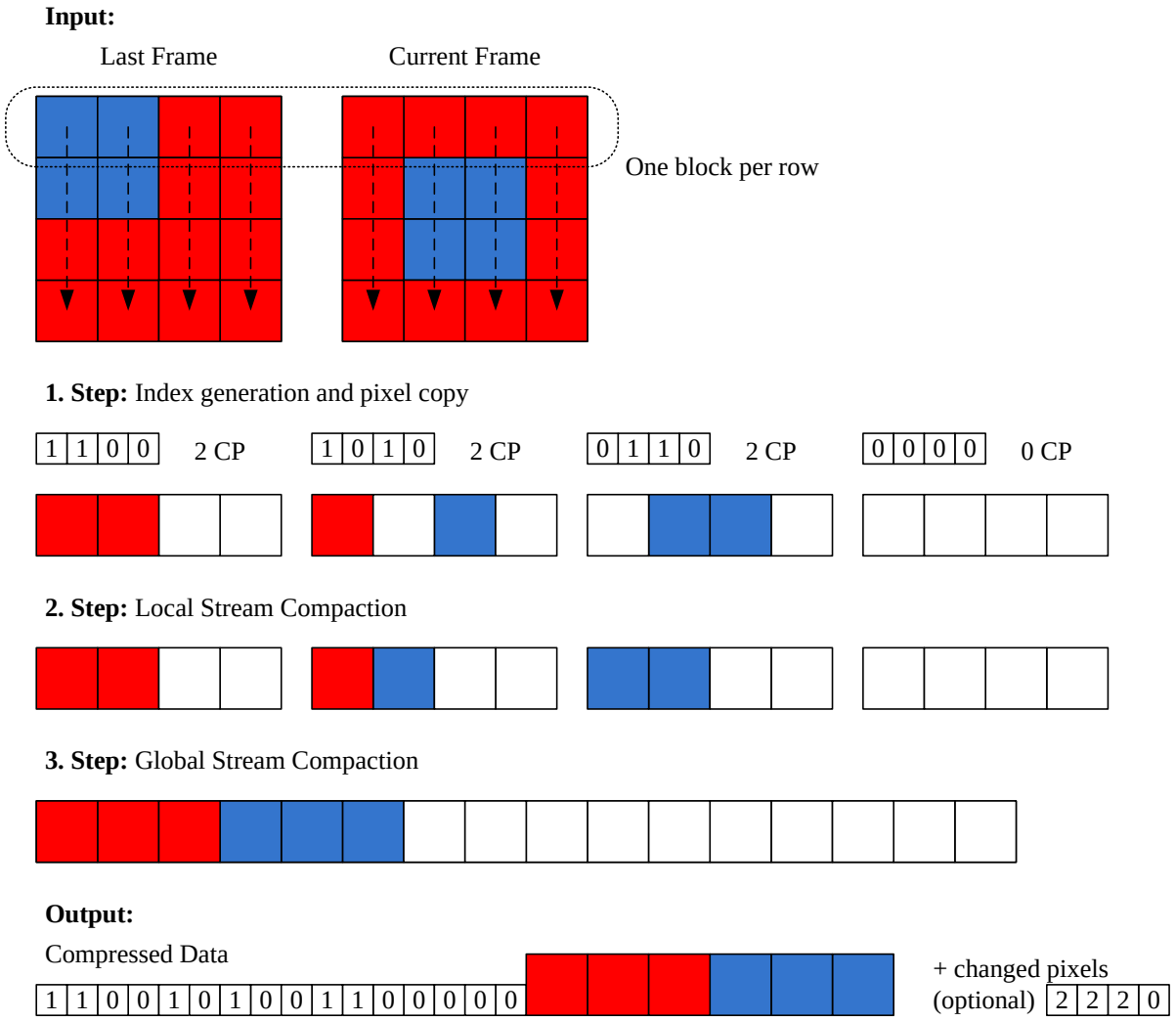

Figure 6.5: The steps of the parallel difference encoding algorithm with index generation, using 4 blocks with 4 threads and one block per line of the 4x4 frame.

can be run in

$$O(m + m * \log k + \log m) = O(m * \log k) = O(n * \frac{\log k}{k})$$

time in parallel with n = number of pixels, k = number of threads, m = number of blocks and $n = m * k$. I.e. depending on the amount of available threads the parallel

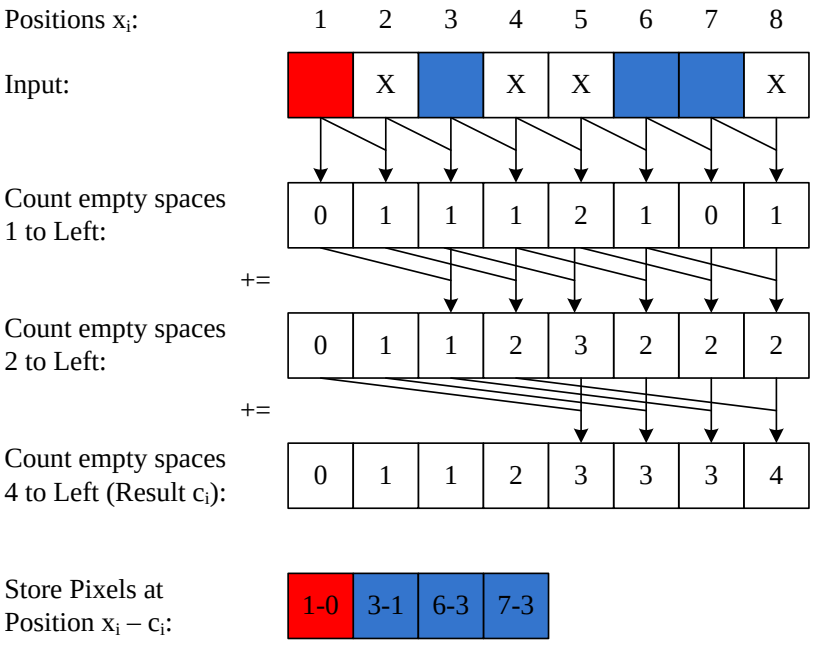

Figure 6.6: Parallel, local stream compaction on an array with 8 fields.

algorithm in the worst case performs equally to the sequential one ($O(n)$) with $k = 1$ threads and in best case achieves $O(\log n)$ with $k = n$ threads.

6.2.4.4 JPEG

The most common format for still image compression is the JPEG standard [66]. It uses several attributes of human vision to eliminate unnecessary or minor information from the images and combines them with traditional compression algorithms. Since it delivers high compression rates with only moderately complex algorithms, it fits very well for the application in RV. Additionally, many of its compute intensive parts are well suited for parallelization since they can be calculated independently for single pixels or small groups of pixels. Figure 6.2.4.4 shows the main steps of a JPEG conform compression. The first step is the **color conversion** of the commonly used RGB format into the YCbCr format. This allows for efficient **downsampling**[3] of the raw image data. In the following step, the **Discrete Cosine Transformation** is used to transform the data of 8x8 pixel blocks from the spacial domain to the frequency domain. Since nearly all pictures have some kind of patterns or areas of similar colors, low frequencies are dominant in the frequency domain, whereas many higher frequency will be zero. This effect is amplified by the next compression step, the **Quantization**, where

[3]Downsampling in this case means to reduce the resolution of the chrominance components to achieve an initial compression. The human vision is much more sensitive to luminance than it is to chrominance variations, therefore one can achieve high compression rates with only minimal loss of quality. E.g. 4:1 for the Cb and Cr components (YCbCr 4:2:0).

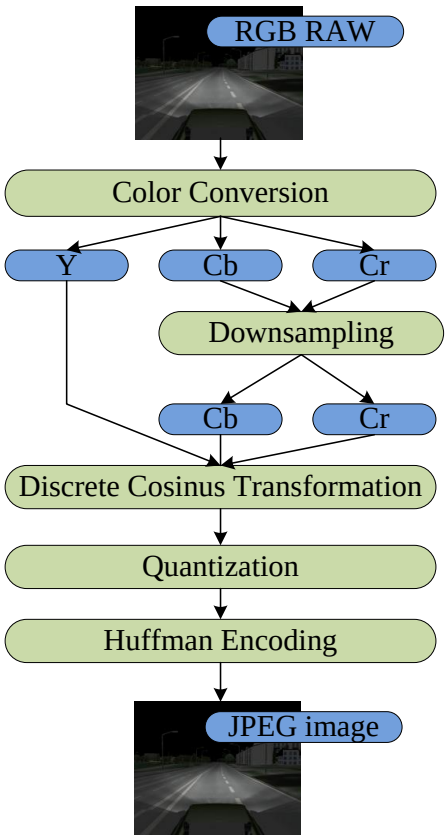

Figure 6.7: Steps of the JPEG compression.

the computed DCT coefficients are divided by a quantization value. Different quantization values are specified for different frequencies, making use of the fact that the human eye can distinguish changes in the lower frequencies a lot better than changes in high frequencies. Through **Quantization** (i.e. dividing all coefficients by constants) many of the high frequencies in the 8x8 blocks turn zero. The subsequently following Huffman encoding (a basic dictionary based compression method) makes use of the many resulting zeros to achieve the actual compression. For more insight on the algorithms JPEG uses, please see [66]. A widely used open source implementation of the JPEG standard is available as libjpeg [29].

6.2.4.5 Parallel JPEG Compression

The parallelization of the JPEG compression is done in steps according to the model in figure 6.2.4.4. The **color conversion** is the first step of the algorithm and can be computed independently for every single pixel by a fixed formula (constants may vary depending on which YCbCr standard is used). With k = amount of threads available, this step can be computed in $O(\frac{n}{k})$ time. The second step that **downsamples** the Cb and Cr components of 4 pixels can also be computed independently for a group of 4

values. Thus, with k threads available, this computation can be done in $O(\frac{n}{k})$. The third and most complex step of the algorithm is the **Discrete Cosine Transformation** (DCT). The JPEG standard determines the following two-dimensional DCT formula for the transformation of the 8x8 pixel blocks into DCT coefficients G_{ij}:

$$G_{ij} = \frac{1}{4} C_i C_j \sum_{x=0}^{7} \sum_{y=0}^{7} p_{xy} \cos\left(\frac{(2x+1)i\pi}{16}\right) \cos\left(\frac{(2x+1)j\pi}{16}\right)$$

$$\text{where } C_f = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } f = 0 \\ 1 & \text{for } f > 0 \end{cases} \text{ and } 0 \leqslant i, j \leqslant 7.$$

Most JPEG implementations use a simplified method that bases on the computation and combination of one-dimensional DCTs. In a first substep the 1D DCTs are computed for the rows of a pixel block and in the second substep these results are used to compute the 1D DCT of the columns. Both computations follow the following formula to compute the 1D-DCT coefficients S_i:

$$S_i = \frac{C_i}{2} \sum_{x=0}^{7} \cos\left(\frac{(2x+1)i\pi}{16}\right)$$

$$C_i = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } i = 0 \\ 1 & \text{for } i > 0 \end{cases} \text{ and } 0 \leqslant i \leqslant 7.$$

Through splitting up the 2D DCT one achieves a better parallelizability and less complex computations. Through optimization it is possible to further reduce the amount of necessary computing operations as shown in [3]. Since the computation of the rows and columns are independent but the computation of the columns bases on the results of the rows computation, a maximum of 8 threads can be used to compute the DCT for 1 block. This step takes $O(\frac{n}{b^2} * b) = O(\frac{n}{b}$ time with $b = k$ modulo c and $c! = 0$ as block dimension ($b = 8$ in this case). The next step, **Quantization**, performs a simple division on each pixel in the 8x8 block, thus every pixel can be processed by one thread. With k threads available, this step can be computed in $O(\frac{n}{k})$ time. The quantization table for the pixel blocks in JPEG are constant and additional scaling factors can be applied beforehand. The **Huffman Coding** of the resulting DCT coefficients cannot be executed in parallel since a dynamic tree needs to be generated where all results depend on the results of their predecessor. There are approaches to parallelize Huffman encoding, but they work only under certain assumptions, which are contradictive to the JPEG standard (see for example [27] and [11] pp. 263f). Therefore this step of the algorithm is performed sequentially in $O(n)$ time. All in all the first four steps of the JPEG algorithm can now be computed in $O(\frac{n}{k})$ time in parallel, only Huffman encoding takes $O(n)$ time. In practical application the compute intensive steps (3 and 4) benefit well from the parallelization, whereas the Huffman encoding is already

optimized for fast compression of the DCT coefficients through lookup tables in the libjpeg, therefore overall speed improvements are very good for this combination of sequential and parallel execution as shown in section 6.5.

6.3 Remote Visualization on Hybrid Clusters

In addition to the remote visualization scenarios described in section 3.2, hybrid cluster systems pose additional demands to RV systems. The most obvious one is that clusters use distributed computing to increase performance. The same holds for clusters or parts of clusters specialized on visualization. In order to efficiently use RV on those clusters the RV systems need to be capable to process and compose subimages from different nodes of the cluster. That means for example that if a complex scene is split up in a sort-last manner, so that each node of a 4 node visualization cluster processes a tile of an image, the RV server runs on each node and sends the result to one client, where it then is composed to the final frame. Such functionality implies open and flexible platforms like the one proposed in this thesis. Invire can easily be extended to serve further special needs of hybrid clusters as it was designed as a modular system that only uses open source software. All parts were designed with a focus on IS/VR application on hybrid cluster systems.

6.4 Prototype - The Invire Framework

The Remote Visualization framework was implemented from scratch and offers a basic platform for grabbing, compressing and transferring rendered images, optimized for the usage in hybrid cluster surroundings. Hence, the implementation of the framework as well as the realization of selected modules are described in the following. For a better understanding, a quick introduction of the GPGPU API CUDA was given in section 2.3. It is intensely used for the realization of the parallel compression algorithms.

The prototypical implementation is structured into 3 components: The Invire plugin which encapsulates all server functionalities and offers different interfaces to the OpenGL application that is remotely visualized; the Invire client, which offers all client side functionalities like displaying the remote rendered images and accepting input from the remote user; and the Invire library which encapsulates all common classes such as those for compression and those containing file formats etc.. The realization of these three components is described in the following. The main focus lies on the implementation of the classes in the invire library since they also contain the newly developed compression methods.

6.4.1 Invire Plugin

The Invire Plugin encapsulates the server functionalities of the framework. It integrates the Invire framework into host OpenGL applications, grabs the rendered frames, initiates potential compression, sends the frames over the network and receives and passes remote input events to the host application. The involved components and their functionalities are described in the following.

6.4.1.1 Integration into Host Applications

The Invire Plugin is not a self-contained application. It can be described best as an interface between the Invire System and the OpenGL application. To fulfill this task it offers one major function which needs to be integrated into the main rendering loop of an OpenGL application: `grabFrame (int width, int height, int bpp, int compression)`. By integrating this function into the host application the developer allows Invire to grab and process every rendered frame. By setting the parameters the developer can also influence the functionality of Invire. This integration is not transparent for the user / developer and requires (little) code adaption. However, it has two advantages over a transparent integration like in for example Chromium [28] or VirtualGL [78] for testing and benchmarking purposes:

- the host application can control and influence the functionality of Invire directly. E.g. if it can predict which compression method is best for the frames it currently renders, it can automatically pass this information to Invire and thereby sets the optimal compression method.

- it is important especially for the testing and benchmarking of new techniques for remote Visualization to be able to clearly distinguish between various different time consumers (e.g. rendering, grabbing, compressing, etc.). This can only be achieved by explicitly integrating the remote rendering functionality into the host application and setting several measurement points.

For the practical usage of RV systems, however, it is vital to also provide a transparent integration. This is a major task as soon as the framework leaves the prototypical stage, but not a focus of this thesis.

The `grabFrame()` function invokes, depending on the selected compression technique, one of two grabbing functions:

Direct: This method grabs the frame by calling the `glReadPixels()` function, which copies the actual content of the graphics cards framebuffer to a specified location in host memory. This method is used for CPU-based compression algorithms and is offered by the OpenGL API.

CUDA: This method allocates memory on the graphics hardware by generating so called `PixelBufferObjects`. These can be used to universally address graphics

memory and to store arbitrary data. After the allocation, the `glReadPixels()` function is used to read the pixels to the `PixelBufferObject`. The CUDA method is used for GPU-based compression.

Grabbing can be invoked more than once to store subsequent frames for comparison based compression or multiple render targets. The result of a call of one of the grabbing functions is a `Frame` object which contains several properties of the grabbed frame (size, bits per pixel, color model etc.) and a pointer to the location in memory or the `PixelBufferObject` where the actual frame is stored. This object is passed to the appropriate compression module in the Invire library (see section 6.4.3) where it is compressed and a pointer to the result is set in the frame object. The `Frame` object is passed to the `TCPServer` where the properties (including the total size of the compressed frame in bytes) are serialized and written into a proprietary header. Finally, the header followed by the compressed frame is sent over a TCP connection to the client.

6.4.1.2 Remote Control of the Application

The third main functionality of the Invire Plugin is the reception and passing of remote input events. Those are generated through inputs to the Invire client and passed over the network as serialized events. Those events are received by an `InteractionServer` and converted into standard X-Events for the Windowing System X-Server [20]. The X-Events are passed to the application which interprets them as if they were inputs of a locally connect mouse and keyboard set. Thereby it is possible to seamlessly steer the application from a remote client.

6.4.2 Invire Client

The Invire Client is the part of the framework which is executed at the user's local computer. Its main functionality is to display the remotely rendered frames. Therefore it has to receive and decompress them. Additionally, it offers a GUI interface to control the properties of Invire (select compression method, toggle input etc.) and offers the possibility to transfer local input to the remote application. The main functions are described in the following.

6.4.2.1 Reception, Decompression and Displaying of the Frames

Upon the reception of a frame header through the `TCPClient`, a new `Frame` object is generated and enough memory is allocated to store the compressed pixel data. According to the kind of decompression, either CPU-based or GPU-based, the allocated memory area is located on host or graphics memory. Thereafter, a pointer to the `Frame` object is passed to the appropriate decompression class in the Invire library. After decompression, the raw pixel data lies either in host or in graphics memory. In case it resides in host memory, the OpenGL function `glDrawPixels()` is used to display the

received frame. If a GPU-based decompression method is applied, the raw pixel data already resides in graphics memory. Thus, this memory area can be marked as a texture which then can be mapped to a blank rectangle of the size of the output window. This is the most efficient way for displaying raw pixel data since graphics hardware is highly optimized for the mapping and rendering of textures.

6.4.2.2 Controlling the Remote Application

An optional setting allows mouse and keyboard input from within the client window to be captured and send to the remote application, which takes this input as local keyboard and mouse commands. This is done by so called glut (OpenGL utility toolkit) callbacks. Every time the mouse is moved, a button is pressed or a key on the keyboard is hit, one of the callback functions is called. These functions generate a proprietary event message containing status information, such as the position of the mouse inside the window, which mouse button is pushed or held or which key was hit. These events are serialized and sent over the `TCPClient` to the Invire plugin where they are received and interpreted as described above.

6.4.3 Invire Library

The Invire library is the component of the framework where all common classes of Invire are encapsulated. Those are the common formats such as the `Frame` objects, all compression and decompression algorithms as well as tools for benchmarking and testing. The following describes the implementation of the major component classes.

6.4.3.1 The Frame Object

The `frame` object holds all relevant information for a frame and pointers to the memory locations on host or graphics memory where the compressed or uncompressed pixels are stored. The following listing shows all of its variables including short descriptions:

bool m_newFrame: is set true if this frame is not displayed yet

int m_compression: the id of the selected compression method

int m_filesize: size in bytes before compression

int m_transsize: size in bytes after compression

int m_width: width of the frame

int m_height: height of the frame

int m_bpp: bytes per pixel

char* m_glFormat: format of the raw pixel data (e.g. RGB)

unsigned char* m_pixels: pointer to memory location where the uncompressed pixel data is stored

unsigned char* m_compressedPixels: pointer to memory location where the compressed pixel data is stored

unsigned int m_pixelBufferObject: id of the Pixel Buffer Object where the pixel data is stored for GPU-de/compression

A `frame` object is first generated when a frame is grabbed and then used for all further operations on the frames, i.e. compression, transmission and decompression. Depending on the current compression method one ore more `frame` objects reside in memory. For frame-to-frame coherent compression methods the last frame(s) are also stored to be able to use them for compression.

6.4.3.2 Compression/Decompression - General

All compression/decompression algorithms are implemented in the Invire library. To guarantee interoperability and exchangeability, all classes for compression are derived from a common `Compression` class. This ensures that all methods provide a `compress()` function which compresses a frame that is passed as an argument and a `decompress()` function which decompresses the given frame. Furthermore, all available compression methods are listed and assigned to unique IDs in this class. These IDs determine the compression/decompression methods system wide and are stored in the frame object after compression.

The implementation of the CPU-based compression algorithms is quite straightforward and follows the description given in section 6.2.4. For further details please consult the source code and the doxygen-generated documentation. The main innovation is in the implementation of the CUDA-based compression methods which are described in the following.

6.4.3.3 Compression - CUDA-Based Difference with Index Compression (DIC)

The first CUDA-based method that was implemented is the parallel difference with index method described in section 6.2.4.3. After grabbing two consecutive frames by the CUDA-based grabbing method and storing them in the memory of the graphics card, the actual compression method for the CUDA-based DIC is called. It allocates new arrays for the index, the changed pixels, the numbers of changed pixels per block and the memory indices where the changed pixels of each blocks are written to in the result array. After that a first CUDA-Kernel is called with a blocksize of 256 threads[4]. The first kernel does the following:

[4] 256 threads turned out to be the most effective block size in this scenario. More or less threads resulted in performance decrease.

1. Load 256 corresponding pixels of the last and the current frame into shared memory.

2. 8 threads in parallel write an index bit by comparing a pixel of the last frame to a pixel of the current frame. 1 stands for pixel changed, 0 stands for pixel did not change. Only 8 threads can do this step in parallel, because writing simultaneously to one char by more than one thread results in inconsistent data and unpredictable behavior. This step is repeated 8 times. The resulting 8 bytes that form the partial index are copied from shared to global memory by 8 threads.

3. To determine the memory position of each changed pixel in the local result, the local stream compaction algorithm based on the stream compaction implementation described in [26] is invoked. In the first step 256 threads determine if the pixel at their ID i and the pixel at the ID i-1 have changed and store the sum of these information in $\mathtt{c(i)}$, an array for the amount of changed pixels in shared memory. In the next $\log 256$ steps s the threads add the amount of changed pixels at $\mathtt{c(i)}$ and at ID $\mathtt{c(i} - 2^s\mathtt{)}$ and store them in $\mathtt{c(i)}$. When all steps are computed the resulting information is used to compute the position in local memory where the changed pixels are stored. This is done by the threads with IDs that belong to the changed pixels.

4. The total amount of changed pixels in that block is copied to global memory.

After the first kernel processed all $\frac{\mathtt{imagesize}}{256}$ blocks, the index generation is complete and the result array consists of locally compacted pixel rows which still need to be compacted globally to achieve the actual compression. This is done with the help of a second kernel, following the implementation described in [25]. It uses the array containing the amounts of changed pixels per block to generate a memory index for the final position of each pixel row in the result array. This is done in parallel by summing all previous values up to each position in the array and storing this sum in a new array.

This array, which now contains the absolute memory locations in the result array, is used to copy the changed pixels from the current positions to the new positions in the final result array in parallel. The compressed frame, which consists of the index and the array of the changed pixels in consecutive order[5], is copied back to host memory and sent to the Invire client where it is decompressed CPU-based or GPU-based.

6.4.3.4 Decompression - CUDA-Based Difference with Index Compression (DIC)

If the computer running the Invire client also supports the CUDA architecture, it is possible to use parallel, GPU-based decompression instead of the standard, sequential, CPU-based decompression. The parallel decompression method requires the index,

[5]Optionally, the array that stores the amount of changed pixels per block can be included to ease the parallel decoding of the frame.

the changed pixels and the array that stores the amount of changed pixels per block to reconstruct the original frame. The first step is to calculate the memory position of each pixel row belonging to a thread block of 256 threads. This is done similar to the computation for compression, by using a scan algorithm that sums the amount of changed pixels per block. After that, a second kernel is started with a blocksize of 256 threads. It uses the index to determine the absolute local position of its portion of the changed pixels, and writes a changed pixel where the index is 1 and a pixel from the last frame where the index is 0. When all blocks are done the final decompressed frame is displayed to the user through the Invire Client.

6.4.3.5 Compression - CUDA-Based JPEG

The CUDA-based JPEG compression is logically separated into two CUDA kernels. The first kernel computes the color conversion and the downsampling and the second kernel is responsible for the Discrete Cosine Transformation (DCT) and the Quantization of the DCT coefficients. This partitioning is the best compromise between maximizing the amount of threads per block and minimizing expensive read and write operations from and to global memory. The maximum amount of threads per block is determined by the amount of independent operations on a certain amount of data. It is obvious that color conversion can be done independently for every pixel, thus it is best to use the maximum amount of threads per block. The CUDA Programming Guide [61] recommends 64-256 threads per block as the best value for current hardware[6]. To keep to the JPEG standard, a blocksize of 8x8 pixels was chosen which results in 64 threads for the color conversion step. The downsampling step uses 4 pixels to compute their mean Cb and Cr values, thus it would be best to also use 64 threads per block and assign each thread to 4 pixels. However, it is important to reduce global memory access in CUDA kernels since they consume 200-300 clock cycles in contrast to 4 cycles for a memory access to shared memory. After reading and storing 64 pixels in shared memory for color conversion, it is better to reuse the results for downsampling with just 32 threads, instead of writing the results to global memory and starting a new kernel which reads them in again. Downsampling is performed for both the Cb and the Cr components, therefore 32 threads can compute that step in parallel. The following enumeration briefly describes the computation steps of the first kernel:

1. At first the 64 RGB pixel values are loaded into shared memory in parallel by 64 threads. The RGB values are stored in a one-dimensional array. To be able to compute the downsampling, two-dimensional 4x4 pixel blocks are needed and DCT and quantization require 8x8 pixel blocks. Thus, the pixels are stored in 8x8 blocks in shared memory to allow the subsequent computations on the same data structure.

2. The color conversion is implemented through three equations which compute the YCbCr values from any given RGB pixel. The equations are: Y = 0.29900 * R +

[6]This may change for new revisions of GPUs since they may have more parallel execution units.

0.58700 * G + 0.11400 * B - 128; Cb = -0.16874 * R - 0.33126 * G + 0.50000 * B; Cr = 0.50000 * R + 0.41869 * G - 0.08131 * B. The constants used in these equations are those used in the libJPEG implementation. All three equations are computed sequentially by one thread for each pixel.

3. The downsampling of the Cb and Cr components is done by 32 threads in parallel. 16 threads compute the mean of a 2x2 pixel block for the Cb and the other 16 threads on the same pixel block for the Cr values. The mean is computed by adding all 4 values and dividing the result by 4. The division is replaced by a shifting operation to optimize performance.

4. The last step is to store the downsampled YCbCr values in global memory. Since DCT and quantization is done independently on each component, they are already stored separately. The Y values are stored in consecutive 8x8 pixel blocks which are mapped to a one-dimensional array. The same holds for the Cb and the Cr components. However, the resulting 4x4 blocks of the downsampling need to be grouped to 8x8 blocks to prepare the data for DCT.

After kernel 1 has finished for all blocks the second kernel is started. Each component (Y, Cb and Cr) is processed separately by one call to kernel 2. It has 8 threads per block and does the following:

1. Loading the data is straight forward since it is already stored in the required 8x8 pixel blocks. Each of the 8 threads loads one row of a block into shared memory.

2. To avoid the complex and sequential computation of a 2D DCT it can be split into the of computation of several 1D DCTs. The computation of the DCT for each row is independent and can be done by 8 threads in parallel. Thereafter, those 8 threads compute the DCTs of the columns on the results of the proceeding step. Figure 6.8 depicts the two passes of these steps. The actual computation of the equations shown in section 6.2.4.5, which would cost 896 additions and 1024 multiplications per 8x8 pixel block, can be substituted by the following computations. This algorithm was developed by Arai et al. and is described in [3]:

   Define:
   Input i = Array of 8 values
   Sums: $s_{jk} = i(j) + i(k)$
   Differences: $d_{jk} = i(j) - i(k)$
   for: $0 \leqslant j, k \leqslant 7$.

   $m_1 = 2[(s_{07} + s_{34}) + (s_{25} + s_{16})]$
   $m_2 = (s_{07} + s_{34}) - (s_{25} + s_{16})$
   $m_3 = (s_{07} - s_{34})$

$$m_4 = d_{07}$$
$$m_5 = cos(\pi/4)[(s_{16} - s_{34}) - (s_{07} - s_{34})]$$
$$m_6 = cos(\pi/4)(d_{25} + d_{16})$$
$$m_7 = cos(3\pi/8)[(d_{16} + d_{07}) - (d_{25} + d_{34})]$$
$$m_8 = [cos(\pi/8) + cos(3\pi/8)](d_{16} + d_{07})$$
$$m_9 = [cos(\pi/8) + cos(3\pi/8)](d_{25} + d_{34})$$
$$z_1 = m_4 + m_6$$
$$z_2 = m_4 - m_6$$
$$z_3 = m_8 - m_7$$
$$z_4 = m_9 - m_7$$

These computations lead to the following equations for the 8 DCT coefficients:

$$DCT(0) = m_1$$
$$DCT(1) = m_2$$
$$DCT(2) = m_3 + m_5$$
$$DCT(3) = m_3 - m_5$$
$$DCT(4) = z_2 - z_4$$
$$DCT(5) = z_2 + z_4$$
$$DCT(6) = z_1 + z_3$$
$$DCT(7) = z_1 - z_3$$

The coefficients computed by this algorithm need to be scaled by a constant factor to receive the final coefficients. Since this factor is constant it can easily be integrated into the constant quantization tables in the next step. By using this optimized 1D DCT algorithm the 2D DCT of an 8x8 pixel block can be computed by 464 additions and 80 multiplications.

3. Quantization is a simple division by a constant for every coefficient in the 8x8 block. Quantization tables are similar for all blocks of a component. Before they are applied in kernel 2, the tables are multiplied by the scaling values of the DCT step as described before. Additionally the selected compression quality influences the values of the quantization tables. The scaled and adapted quantization tables are applied to the coefficients by 8 threads in parallel. This could also be done by 64 threads, but in order to avoid copying data back and from global memory the 8 threads of the DCT steps are reused. Again each thread computes one line of 8 DCT coefficients and multiplies them by the inverse of the corresponding value in the quantization table.

4. Finally the quantized DCT coefficients are written back to global memory in 8x8 blocks for each component.

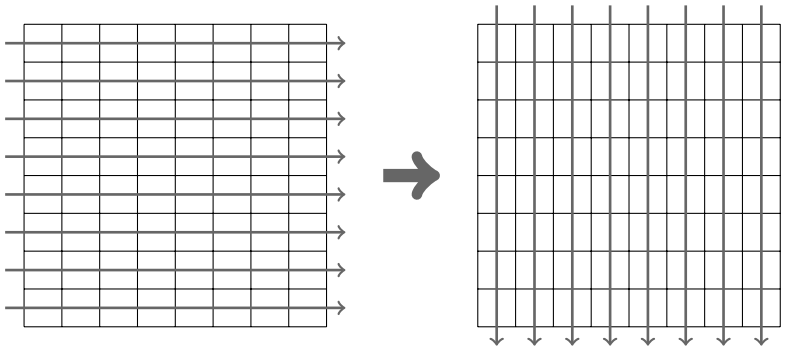

Figure 6.8: 2D DCT through separate 1D DCT computations for an 8x8 pixel block.

The last step of the JPEG compression is the Huffman encoding which makes use of the many zeros in lower frequencies, resulting from DCT and quantization. This step, as described in section 6.2.4.5, can not be computed efficiently in parallel. Thus, the highly optimized sequential version of the libjpeg implementation is used to compute this step. As a side effect, it is possible to use the libjpeg data structure to automatically generate a JPEG conform header. After initializing the library and allocating a `jpeg_compress_struct`, the function `jpeg_write_coefficients` is used to pass the precomputed coefficients to the Huffman encoding facility of the libjpeg. The result after calling `jpeg_finish_compress` is a standard conform JPEG image in a specific memory location. For more implementation details please refer to [41].

The decompression on the client is implemented by using standard libjpeg functions. A CUDA-based decompression is also available and mainly consists of the inverse steps of the encoding (i.e. Huffman decoding, inverse DCT and color conversion from YCbCr to RGB). However, the main focus of this implementation was to show the feasibility of a fast parallel JPEG encoding to achieve high frame rates for RV. Decoding is quite fast in the sequential case already since Huffman Decoding is faster and the quantization step is omitted, thus the parallel decompression does not achieve a significant performance increase (as shown in section 6.5).

6.5 Benchmarking the Remote Visualization Framework with Practical Examples

The Invire framework was designed to evaluate, test and compare algorithms for image compression and frame grabbing for remote rendering. Thus, after introducing and implementing new grabbing and compression techniques, this section deals with various comparative benchmarks of the compression algorithms and performance measurements of the whole system. The Invire framework is also compared to the most commonly used system in the class of Server-Side Remote Rendering for 3D applica-

tion, called VirtualGL, which is briefly introduced in section 6.5.1. In addition to the pure performance benchmarking, a quality assessment for the JPEG-based compression methods was conducted and is described in section 6.5.6, using the SSIM index, which is briefly introduced in section 6.5.2. All benchmarks were carried out on two different sample applications and with varying parameters (e.g. frame resolution, JPEG quality, bandwidth limitations) that influence the performance and the quality of the remotely rendered frames. The hardware that was used to perform the benchmarks, as well as the sample applications, which represent two classes of applications, are described in section 6.5.3.

6.5.1 The Reference Framework VirtualGL

VirtualGL [78] is a very sophisticated and renowned framework for remote rendering. It offers transparent integration, fast compression as well as transmission techniques and mechanisms for the remote interaction. It can also be combined with other frameworks, such as VNC [70], to speed up the remote administration or Chromium [28] to allow for remote rendering with distributed graphics applications. VirtualGL offers different modes for compression and transmission: Besides two modes for uncompressed transmission of the rendered data (as raw RGB stream or as X11 stream), the standard compression mode is based on the JPEG still image compression. For JPEG compression VirtualGL offers two implementations. The first is the pseudo-standard libjpeg [29] implementation which is completely open source and allows decent performance. The second is called TurboJPEG [80] and is based on the Intel(R) Integrated Performance Primitives [30], a set of libraries which contain highly-optimized multimedia functions for x86 processors. According to the developers of VirtualGL, it outperforms the libjpeg implementation by a factor of 2-4. This implementation, however, is not free and cannot be compiled without the licenses for the Intel Performance Primitives. Thus, it was not considered in the overall performance benchmarks. It is included in the pure JPEG comparison benchmarks in section 6.5.5 to allow a comparison between the GPU-based and the fastest available CPU-based method. VirtualGL also offers various optimization options for all kind of applications. For a better comparability all settings were left at the standard values for benchmarking.

6.5.2 Quality Assessment with the SSIM Index

In addition to benchmarking the performance of the proposed systems and comparing it to other systems, it is also vital to make statements about the quality of the lossily compressed images. There are several metrics for the objective measurement of image quality and most of them aim to simulate selected characteristics of the ultimate judge – the human visual system (HVS). One of the first metrics introduced by Mannos et al. in [48] is the mean squared error (MSE) which is computed by averaging the squared intensity differences of distorted (e.g. through compression) and reference image pixels, along with the related quantity of peak signal-to-noise ratio (PSNR).

These methods and many successors, which tried to enhance the original metric to better simulate the characteristics of the HVS, are quite appealing because they are simple to compute, have clear physical meanings and could be reused from other signal processing applications. However there are many reports that find that the methods based on the MSE are not very well matched to perceived quality (see for example [16] and [81]). Thus, Wang et. al introduced a novel approach in 2004 [82], which takes into account that natural images are highly structured. That means that their pixels have strong dependencies, especially when they are spatially proximate. These dependencies carry strong information about the structure and the objects in a visual scene. In contrast to other approaches, which independently compare the channels of the distorted and the original image pixel-wise, Wang et al. propose a combined measurement of Luminance, Contrast and Structure components. The combination of these three components results in the so called SSIM index which indicates the objective image quality by a value between 0 and 100%. They showed that their results were much closer to results which were generated by performing subjective quality assessments by human users. This is why the SSIM metric is also used in this thesis to evaluate the quality of the remotely rendered and lossily compressed images. The goal was to achieve SSIM indices above 90% which translate to nearly invisible quality loss for the compressed images.

In addition, the SSIM index is used to classify the heterogeneity between two consecutive frames of the two sample applications. This allows for the clarification of certain differences in the benchmarking results. If the mean SSIM index of two consecutive frames is high, there is only little heterogeneity between them. That means that difference-based, lossless compression methods are much likely to achieve high compression rates for high SSIM indices. If, the other way round, the mean SSIM index of two consecutive frames is low (below 60%) it is likely that those compression techniques do not produce reasonable compression rates.

6.5.3 The Benchmarked System and the Sample Applications

In order to prove the theoretical concept and to compare the described compression algorithms, we prototypically implemented the Invire system and tested it in a sandbox environment. The server and the client part run on a computer equipped with a CUDA-ready Geforce 8800 GTS (G92) graphics card by NVIDIA. It has 12 multiprocessors and a wrap size[7] of 32. That leads to $k = 12 * 32 = 384$ threads that can run concurrently. Both computers are equipped with an Intel Core 2 Duo E4400 CPU running at 2.00 GHz and 2GB of RAM. The network connection is a 100Mbit Ethernet connected through a switch. The peak nominal bandwidth that could be achieved over this network was roughly 11,8 MB/s. Two sample applications where chosen to represent groups of applications. The following criteria played a role in the selection:

[7] Number of threads that are executed in parallel on one multiprocessor.

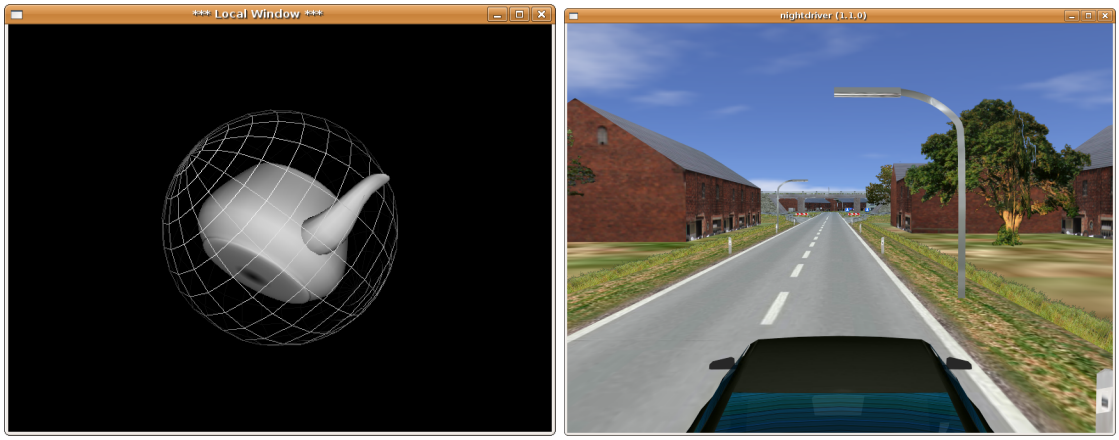

Figure 6.9: Test cases: a) simple teapot, b) Virtual Night Drive without headlight simulation.

The mean heterogeneity of two consecutive frames:
This parameter determines if an application is highly dynamic or rather static. In the field of IS/VR there exist both kinds of applications, for example rather static CAD visualizations or highly dynamic virtual worlds with rich visual effects. Thus, it was important to choose one application of each class. The heterogeneity could be measured with the help of the SSIM index.

Multicolored or grayscale frames:
Again, there are applications in IS/VR that either have the one or the other attribute, e.g. grayscale medical imaging visualizations vs. high dynamic range VR environments.

Large uniformly colored areas vs. very heterogenous multicolored areas:
As for example in CAD applications vs. highly dynamic virtual worlds with rich visual effects.

Following these criteria, two sample applications were selected:

A rotating teapot (see figure 6.9a) as a representative of the area of rather static object visualization applications. It simulates a steady rotation of a grayscaled 3d model, with one fixed light source. The background is uniformly black and not lit. This sample application has quite a low heterogeneity between two consecutive frames (mean SSIM index between two consecutive frames 89,35%), greyscale frames and at least one large uniformly colored area (the black background). These attributes describe a rather typical CAD or 3D object visualization application.

The Virtual Night Drive Simulator (VND) (see section 5.3.1 and figure 6.9b)) as a representative of the area of highly dynamic virtual worlds. The VND is specialized on simulating automotive headlights at night and uses the shaders of

the graphics card to calculate the luminance intensity per pixel. It also features a daylight view on the scene which is very detailed and feature-rich and offers dynamic lighting features. This sample application has quite a high heterogeneity between two consecutive frames (mean SSIM index between two consecutive frames 53,89%), multicolored frames and very heterogeneous multicolored areas because of the extensive use of textures. These attributes describe a rather typical VR application.

By selecting those two representatives of different application areas, the benchmarking results can give information about how well each of the compression algorithms and the overall system might perform depending on the selected application. Possibly there are algorithms that are more suited for one or the other area. There certainly are more groups of applications in IS/VR, but the ones selected represent the biggest groups with the most obvious differences. The specifications of most of the other application groups lie in between these two extremes.

6.5.4 Benchmarking and Comparing the Overall System Performance

In the first round of benchmarks the overall system performance (i.e. grabbing, compression, network transmission, decompression and displaying) is calculated and compared to the performance of the reference system VirtualGL. Both sample applications are tested in three different resolutions (640x480, 1024x768 and 1680x1050 pixels) to study the effect of the higher load for CPU/GPU and network. [79] describes performance measurements with the VirtualGL framework and gives a good description of the metrics to use: It is pointed out that simply measuring the achievable frames per second is not enough since one cannot determine if the limitation to the measured value is due to processing limitation or bandwidth limitation. Thus, in the following benchmarking diagrams both the achievable frame rate and the consumed bandwidth are shown in order to allow the comparison between network and processor limitations of each compression technique. To provide statistical correctness, the measurements were conducted for 60 seconds and a confidence interval is given for each mean value of the measured FPS. The confidence interval includes 95% of all measured values. For both RV systems, Invire and VirtualGL, all available compression methods are tested and the overall performance in frames per second as well as the required bandwidth in MB/s is depicted in the diagrams. The Invire framework includes run length, difference, difference with index, CUDA-based difference with index, JPEG and CUDA-based JPEG compression methods. Additionally the CUDA-based methods can optionally make use of CUDA-based decompression on the client. VirtualGL offers uncompressed RGB transmission as well as libjpeg based JPEG compression.

6.5.4.1 Rotating Teapot

Figure 6.10 shows the benchmarking results for the rotating teapot application in 640x480 resolution. Both uncompressed methods (Invire uncompressed and VirtualGL

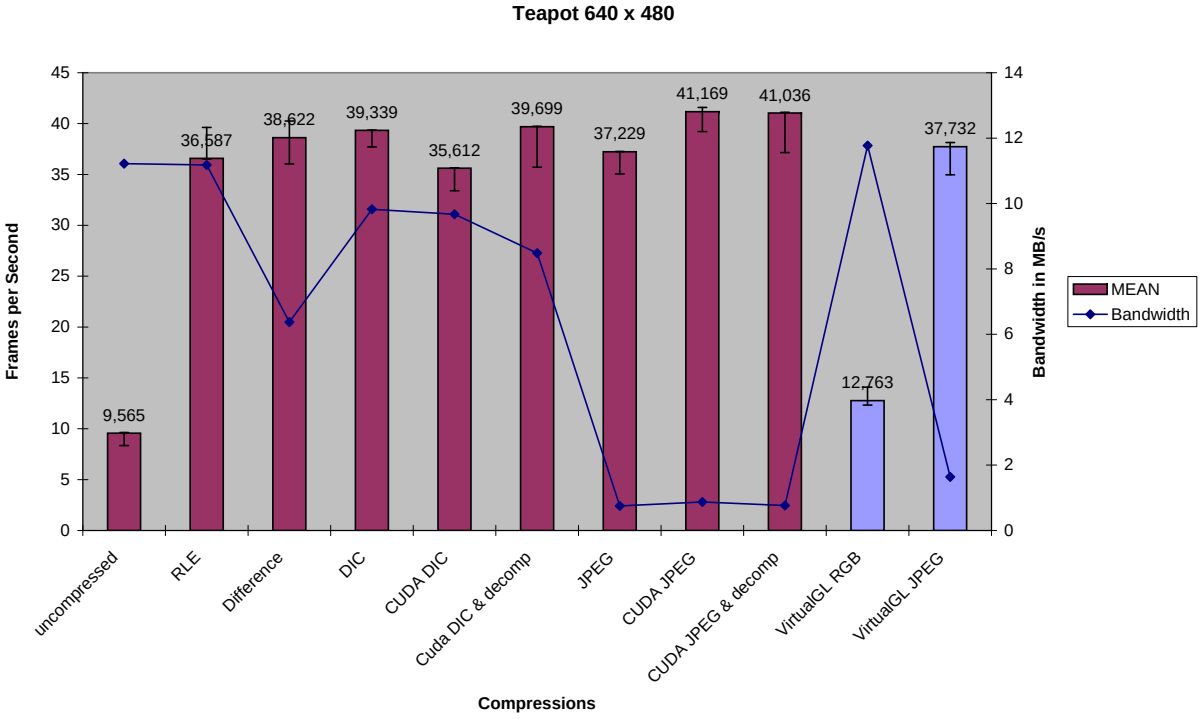

Figure 6.10: Rotating Teapot in 640 x 480 Pixels.

RGB) as well as the run length encoding algorithm are already limited by the available bandwidth. All other techniques are limited by the processor capabilities and achieve frame rates around 40 Fps. Especially the JPEG based methods achieve very good compression results, however, at the cost of slightly visible block artifacts (JPEG quality[8] set to 75 in all benchmarks). The lossless methods consume more bandwidth, but can achieve similar frame rates for this resolution. The CUDA-based algorithms slightly outperform their non-parallel equivalents but initialization overhead consumes most of the parallel speed-up for small resolutions like this. Figure 6.11 shows the benchmarking results for the rotating teapot application in 1024x768 resolution. Again both uncompressed methods and the run length encoding algorithm are already limited by the available bandwidth. Additionally, all difference based algorithms are limited by the bandwidth. However, it is visible that the simple RLE algorithm achieves a better compression and thereby results in a higher frame rate for the available bandwidth. The JPEG based methods are again limited by the processing power and reach values between 21 and 29 Fps, where the CUDA-based algorithms perform about 20% better than the CPU based methods. Again bandwidth consumption is very low for all

[8]The quality value determines the compression rate of the JPEG algorithm. It is incorporated into the values of the quantization tables which influence the amount of zeros in the high frequencies. If a low quality is set, these values are higher and eliminate more coefficients than for higher quality value. 75 represents a good compression/quality ratio.

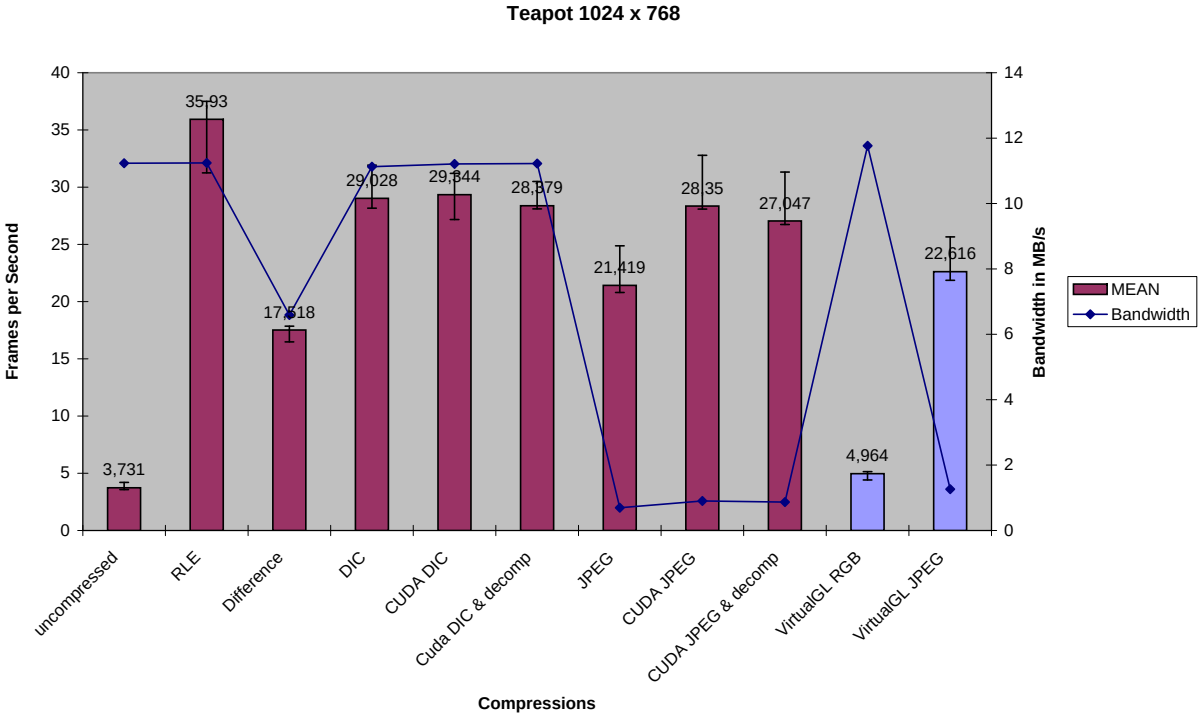

Figure 6.11: Rotating Teapot in 1024 x 768 Pixels

JPEG methods (around 1 MB/s). Overall the best result is achieved by the simple RLE method (35,93 Fps) for this scenario.
Figure 6.12 shows the benchmarking results for the rotating teapot application in 1680x1050 resolution. At this resolution only the uncompressed as well as the RLE and the CUDA based Difference with Index algorithm are limited by the bandwidth. Thus, the CUDA-based lossless methods slightly outperform the CPU-based methods with more potential for higher available bandwidths. The RLE method outperforms all other methods again, because of its great performance/ratio for this sample application. All JPEG methods are again clearly limited by processing power and the CUDA-based JPEG implementation outperforms the CPU based version by roughly 30 %.

Rotating teapot - Conclusion

The rotating teapot application is a special case in terms of remote rendering. Since it has large unicolored areas (the whole background, regions of the teapot) and very predictable and homogenous object movements, it is well suited for lossless compression methods. Especially the very simple run length encoding performs very well in all evaluated resolutions since it can achieve great compression results with just one pass over the input data. JPEG based methods achieve great compression results, but only for the cost of worse performance and lossy image compression. The CUDA-enhanced

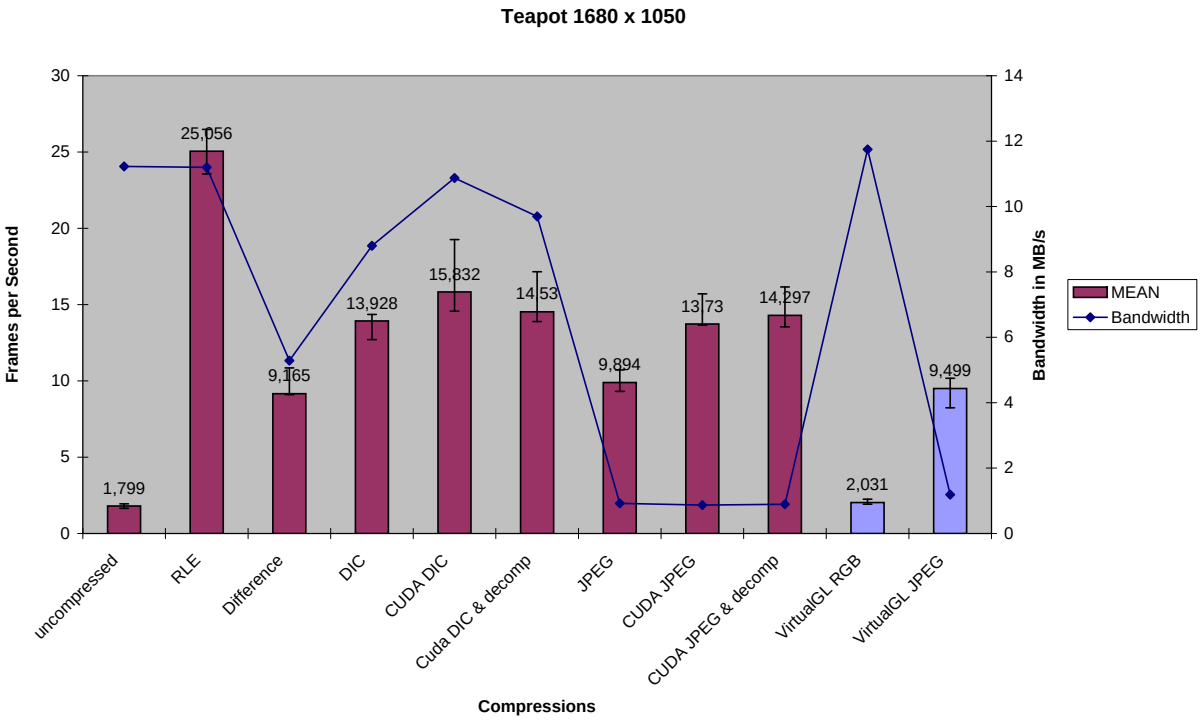

Figure 6.12: Rotating Teapot in 1680 x 1050 Pixels

algorithms improve the performance for higher resolutions (1024x768 or higher) by 20-30% with a rising tendency. The rotating teapot scenario was chosen because especially CAD and many scientific visualization applications have similar displaying patterns. Therefore it is very promising to use simple lossless compression methods for the remote rendering of such applications.

6.5.4.2 Virtual Night Driver

Figure 6.13 shows the benchmarking results for the VND application in 640x480 resolution. It becomes very clear that, as well as the uncompressed transmission, also the lossless compression methods are limited by the available bandwidth. This is a result of the very heterogeneous color distribution of the application. All lossless compression methods in Invire base either on similar pixel values in the proximity or in the following frame. However, if for example a car moves through a textured scene none of these conditions hold. Thus the compressed frames are likely to be only slightly smaller or even bigger than the original frames. This is why the lossless compression methods were not considered in the benchmarks of the higher resolutions for the VND application. Besides this, the JPEG based methods result in frame rates between 29 and 37 FPS. For Invire, the CUDA based JPEG methods perform around 20% better than the CPU-based. In comparison to VirtualGL's JPEG implementation there is no significant performance increase at this resolution. Compression rates are better for

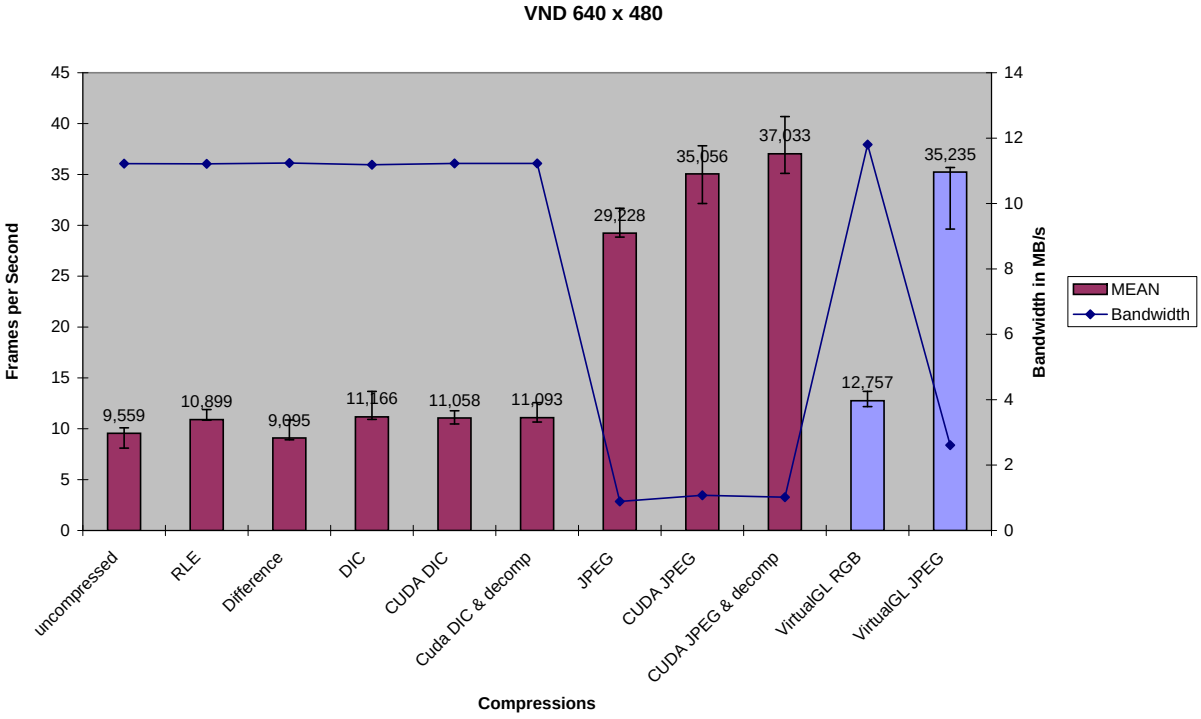

Figure 6.13: Virtual Night Drive in 640 x 480 Pixels.

Invire's JPEG algorithms (1 MB/s vs. 2.5MB/s in bandwidth usage). Figure 6.14a) shows the benchmarking results for the VND application in 1024x768 resolution. Only JPEG compression methods are considered and the CPU-based methods of Invire and VirtualGL achieve about the same results (17 Fps). Now the CUDA-based methods of Invire outperform the CPU-based one by roughly 30%. Still the compression is better for the Invire implementation. Figure 6.14b) shows the benchmarking results for the VND application in 1680x1050 resolution. For this high resolution the gab between CPU-based and CUDA-based widens further. Now the GPU-accelerated methods are about 60% faster than the CPU based methods. Bandwidth usage is nearly similar for all JPEG methods.

Virtual Night Driver - Conclusion

Applications like the VND, which make extensive use of textures, lighting and 3D models to for example simulate a very realistic surrounding, are not well suited for the lossless compression methods. There is simply too much heterogeneity in the frames as that the coding algorithms could efficiently compress differences or consecutively uniform pixels. Thus, the JPEG algorithms prove to work a lot better since the JPEG standard was initially designed to compress such images. Especially the CUDA-based implementation shows good results mainly for high resolutions. The bandwidth limitation in the 100 MBit network is not reached by far and thus there might be even more

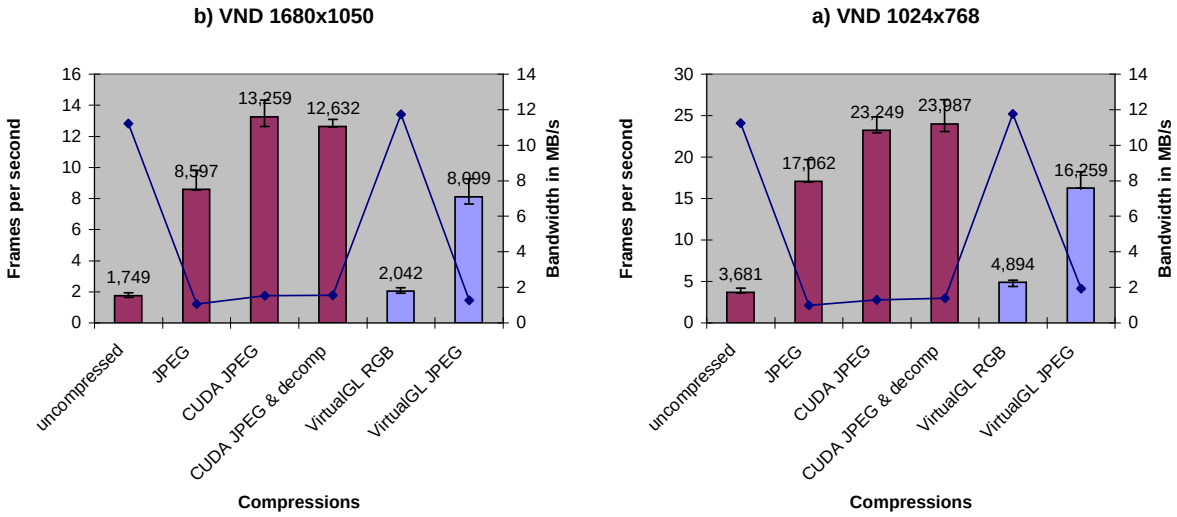

Figure 6.14: a)Virtual Night Drive in 1024 x 768 Pixels and b) Virtual Night Drive in 1680 x 1050 Pixels.

potential for optimized CUDA-based approaches.

6.5.5 Benchmarking Three Different JPEG Implementations

The results of the benchmarking of the overall system performance show that the JPEG based compression algorithms are likely to be the most universal choice for different kinds of applications. To get a deeper insight in the raw performance of the different JPEG implementations of Invire, the following benchmarks compare the pure compression times of the libjpeg [29], CUDA-based JPEG (see section 6.2.4.5) and turboJPEG [80] implementations. All three implementations were tested in three different resolutions with the two sample applications (VND and rotating teapot) described above. For each measurement 1000 samples were taken and the mean of these samples as well as the minimum and maximum of 98% of all measurements are displayed in the diagrams. For the CUDA-based method the overall mean time is composed by the mean CPU and GPU compression times. All times are measured in milliseconds.

6.5.5.1 JPEG - Compression Times

Figure 6.15 depicts the comparison of the three JPEG implementations for both applications in 640 x 480 pixels. The libjpeg implementation takes the most time for compression (about 14 ms for teapot and 18 ms for VND), the CUDA-based version is about 60% faster (9 ms teapot and 10ms VND) and turboJPEG outperforms CUDA-based JPEG by the factor 2 (4 ms teapot and 5 ms VND). For CUDA-based compression an interesting shift between GPU and CPU times for the applications can be regarded: The grayscale teapot application produces DCT coefficients that are much easier to

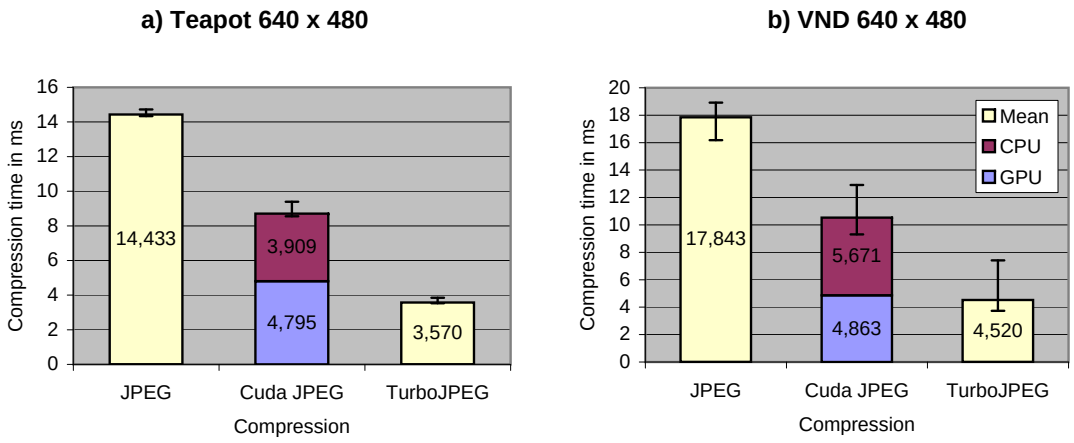

Figure 6.15: Comparison of the JPEG implementations for Teapot (a) and Virtual Night Drive (b) applications in 640 x 480 Pixels.

compress since most of the color values are 0, thus the Huffman coding, which is performed on the CPU, executes faster for this application than for the more elaborate compression of the coefficients of the VND application. Therefore, the time needed for GPU calculation is nearly constant for both applications but the time spend on the CPU varies heavily. The time for GPU computation is still higher as the total time for turboJPEG compression. Theoretically achievable framerates are about 70/56 Fps (Teapot/VND) for libjpeg, 115/95 Fps for CUDA-based JPEG and 280/221 Fps for turboJPEG.

Figure 6.16 depicts the comparison of the three JPEG implementations for both applications in 1024 x 768 pixels. The overall result is quite similar to the lower resolution: The libjpeg implementation is now outperformed by a factor 2 by CUDA-based JPEG and a factor 4 by turboJPEG. Again CPU computation time for CUDA-based JPEG is lower than GPU time for the teapot application and higher for the VND application. The time for GPU computation is now lower as the total time for turboJPEG compression. Theoretically achievable framerates are about 28/22 Fps (Teapot/VND) for libjpeg, 50/43 Fps for Cuda-based JPEG and 108/84 Fps for turboJPEG.

Figure 6.17 depicts the comparison of the three JPEG implementations for both applications in 1680 x 1050 pixels. For this high resolution, a similar performance ratio as for the 1024x768 resolution (1:2:4) can be regarded. However, libjpeg has very high variations in the VND application. This might be caused by memory or cache limitations for the high amount of colored pixels (1.76 MPixels) to process. Theoretically achievable framerates are about 14/9 Fps (Teapot/VND) for libjpeg, 25/21 Fps for CUDA-based JPEG and 57/42 Fps for turboJPEG. Thus, the libjpeg implementation already is the limiting factor for reaching more than 20 fps with this resolution.

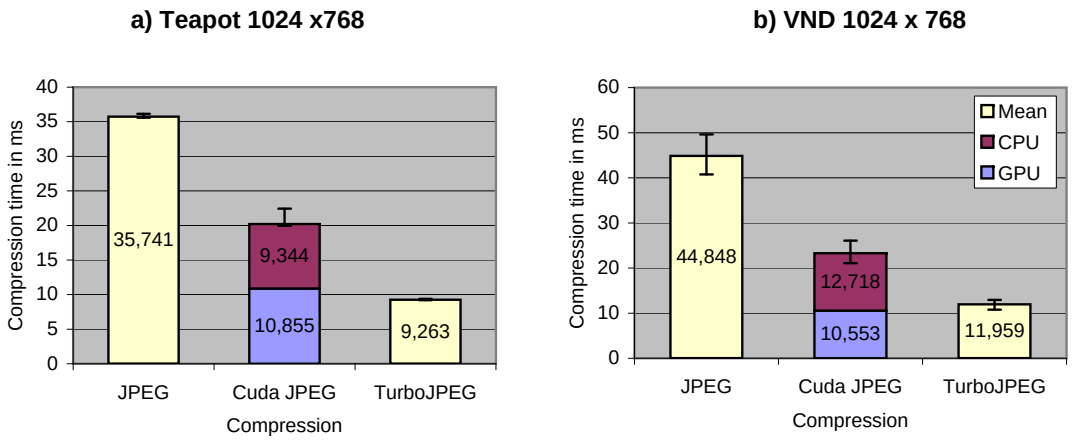

Figure 6.16: Comparison of the JPEG implementations for Teapot (a) and Virtual Night Drive (b) applications in 1024 x 768 Pixels.

JPEG - Conclusion

The comparison of the three JPEG variants shows that the highly optimized turboJPEG implementation is currently the fastest way to compress JPEG-conform images. It uses the Intel Performance Primitives library which "... is an extensive library of multi-core-ready, highly optimized software functions for multimedia data processing, and communications applications..." [80]. Thus it can fully harvest the power of the dual processor built in the testing machine. The CUDA-based JPEG implementation introduced in this thesis, however, partially relies on the unoptimized code of the libjpeg implementation. Especially the sequential Huffman encoding which is used by this implementation seems to be limiting the performance for high resolutions and multi-colored frames. The time spend on the GPU for doing the compute intensive steps (color conversion, downsampling, DCT and quantization) is less than the overall time used by the fast turboJPEG implementation for resolutions of 1024x768 and higher. This is a good perspective to further optimize the CUDA-based implementation and design a CUDA-supported Huffman encoding to outperform the highly optimized turboJPEG implementation. In addition, the GPU-based methods strongly relief the CPU and leave it available for other tasks. The GPU-based compression can either be done on the same card that is responsible for rendering, or on a second graphics card in the visualization node in order to share the computation and rendering load. The libjpeg implementation is by far the slowest but most feature-rich and universal implementation. It also offers an extensive documentation and greatly helps to understand the details of the JPEG standard. Anyway, it is not really suited for the application in remote visualization of IS/VR since it limits the overall system performance for high resolutions by its slow compression times.

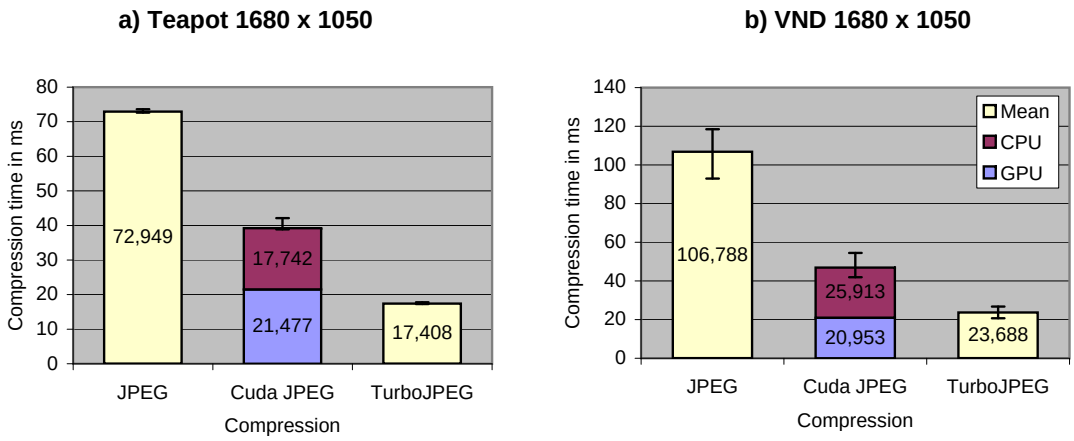

Figure 6.17: Comparison of the JPEG implementations for Teapot (a) and Virtual Night Drive (b) applications in 1680 x 1050 Pixels.

6.5.6 Quality Assessment of the Lossy Compression Methods

As described before, besides achieving reasonable performance, it is extremely important for the practical usage of RV to ensure a certain level of quality of the lossily compressed images. Thus, the following section describes quality assessments of the three lossy (JPEG) compression methods, each conducted with the two sample applications. All JPEG implementations were benchmarked with different quality settings q[9] ($q = 25$, $q = 75$ and $q = 100$), and the SSIM index as well as the compression ratio in bytes per pixel are recorded.

Figure 6.18 shows those results in two diagrams for the teapot application. First of all, it is visible that all three implementations produce nearly similar SSIM indices and compression ratios for equal JPEG quality settings. This is evident since all three versions implement the same basic algorithms (variations only in parallelization and hardware acceleration). Small deviations may result from error corrections or rounding differences between the utilized APIs / compilers (GNU/IPP/CUDA) and hardware (CPU/GPU). For the teapot application it is observable that even for very low JPEG quality settings ($q = 25$) the SSIM index is around 95% while achieving a great compression ratio (0.03 bytes per pixels in contrast to 4 bytes per pixel of the raw image). Higher quality settings result in even higher SSIM indices; the best SSIM index / compression ratio is achieved for $q = 75$, as shown in the charts.

Figure 6.19 shows the quality and compression ratio results for the VND application. They are analogue to the one for the teapot application. But, it is observable that the lowest JPEG quality setting ($q = 25$) produces frames with a SSIM index below 90%.

[9]Those settings directly influence compression quality and ratio by altering the quantization table and thereby determining the amount of DCT coefficients that are eliminated. $q = 0$: highest compression with strongly visible artifacts, $q = 100$: lowest compression, best quality, worst compression ratio.

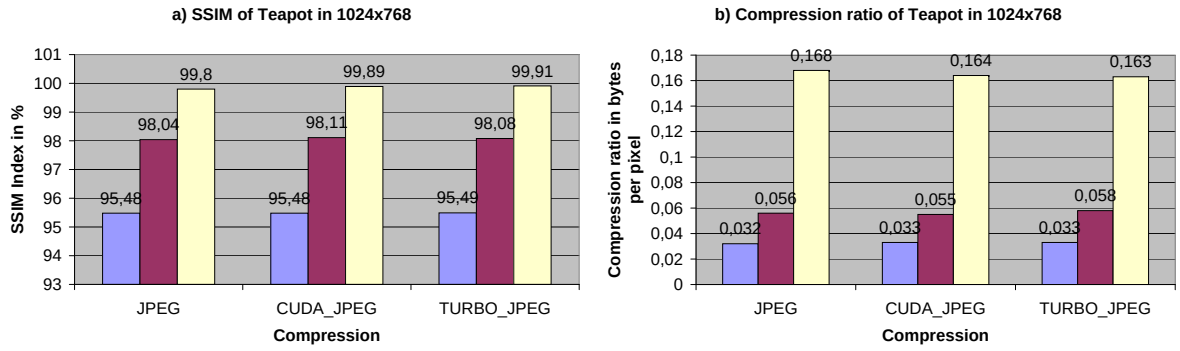

Figure 6.18: SSIM index (a) and compression ratio (b) of the three JPEG implementations for the Teapot application. The color of the bars represents the selected JPEG quality settings: Blue $q = 25$, Red $q = 75$ and White $q = 100$.

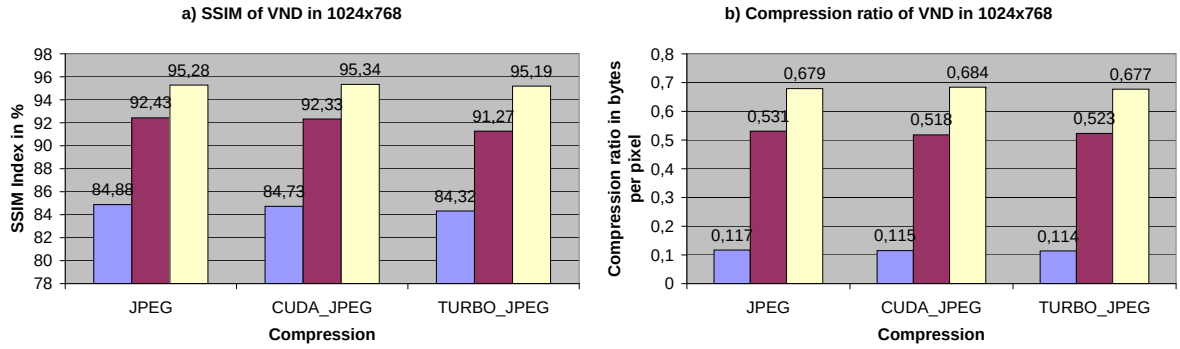

Figure 6.19: SSIM index (a) and compression ratio (b) of the three JPEG implementations for the VND application. The color of the bars represents the selected JPEG quality settings: Blue $q = 25$, Red $q = 75$ and White $q = 100$.

That means that there are visible distortions in the frames that might disturb remote users of the application. For higher q's the SSIM index improves and reaches about 92 % for $q = 75$. Admittedly, also the compression ratio rises by a factor 4 between $q = 25$ and $q = 75$. Setting $q = 100$ further improves the SSIM index a bit, but this improvement is hardly noticeable for the human eye.

The main finding of the quality measurement is that all three JPEG implementations produce very high quality compressed images, even with low JPEG quality settings. However, to ensure that the resulting frames permanently have SSIM indices above 90% q needs to be set to 75 or higher. It is planned to extend the Invire system by an automatic quality assessment component, which dynamically measures the quality of the compressed frames and adapts q to the results of this measurement. This either helps to save bandwidth or to provide the users with optimal quality.

6.5.7 Benchmarking Conclusion

The benchmarks described above surveyed several performance and quality aspects of remote rendering and showed their heavy dependency on the underlying compression algorithms. The first set of benchmarks made clear that for different applications different compression techniques (lossy and lossless) produce very different results. Lossless techniques, for example, are well suited to do high quality remote rendering of visually homogenous and slow-changing applications (e.g. CAD). Admittedly, they are not suited for visually heterogeneous applications such as fast-changing VR-applications. Lossy methods, such as the JPEG still image compression, are deployable more universally since they are mostly independent of the homogeneity of the input image. JPEG achieves great compression in interactive time at the cost of more or less visible loss of image quality, which, however, is acceptable for most IS/VR applications (as shown in section 6.5.6). This is why the second set of benchmarks analyzes the performance of JPEG compression for different implementations of the JPEG standard. The compression speed is extremely important for remote visualization since it mainly determines the achievable overall performance for a remotely rendered application. The proposed GPU-based algorithms perform well in comparison to existing approaches and at the same time relief the CPU. The other limiting factor is the available network bandwidth which was taken into account in the first set of benchmarks. Again, the JPEG-based algorithms provide good compression rates, so that even for low bandwidth networks frames of reasonable quality can be compressed and delivered in time.

6.6 Conclusion - Remote Visualization for IS/VR

Existing systems for Remote Visualization have two major disadvantages regarding their utilization for IS/VR applications: Either they lack the required interactivity by taking too long for compression or they cannot provide a decent image quality for high resolutions. Furthermore most of them cannot handle distributed input or multiple client connections. Thus, the previous chapter presented a framework which helps to find solutions for those problems and allows to deploy them in a practical environment. The framework fulfills the requirements that were pointed out in section 4.4:

- The framework was designed in a modular and open way to allow the integration of arbitrary grabbing and compression methods.
- By designing and implementing methods to access rendered frames directly on the GPU and by integrating the GPGPU API CUDA into the framework, the usage of the GPU for general purpose computing tasks was realized. Thereby, new and fast methods for image compression could be developed, benchmarked and compared to existing methods.
- A plugin component allows for the comfortable integration of the RV functionality into existing applications. Transparent integration is possible through pre-

loaded libraries (under certain circumstances: e.g. Linux OS, correct X and OpenGL versions).

- Server and client are prepared for distributed input and output. This is realized through the capability of handling several TCP connections and allowing the composition of a frame by multiple inputs.
- The whole system bases on freely available software and utilizes well known and approved mechanisms for data transfer, frame grabbing and image/video compression (e.g. TCP-based communication, Pixel Buffer Objects and JPEG standard). The prototypical implementation proved the concept and will be freely available over the web site of the PC^2 [65].

Besides the design and development of the framework for RV, the main contributions of this chapter are the parallel GPU-based image compression methods. They allow for the first time to use the powerful GPUs of the visualization nodes for image compression and outperform most of the standard methods. However, in some cases there are still traditional methods that are slightly faster than the GPU-based ones but also consume almost all of the CPU's processing power. The methods presented in this thesis greatly relief the CPU and bundle all graphics relevant functionalities on the graphics card. The CPU only handles the transfer of the compressed images and the connection between client(s) and server(s).

7

Conclusion

This chapter briefly summarizes the contributions of this thesis, draws a conclusion and outlines directions of future works.

7.1 Contributions

This thesis adds the following main contributions to the state of the art in the area of middleware for hybrid cluster systems:

New models for Computational Steering of IS/VR:
The new models, which reflect the requirements of distributed IS/VR applications, allow for the classification and realization of arbitrary IS/VR application scenarios on hybrid cluster systems. They form the basis for the design of a CS framework and specify the required data types and communication mechanisms.

A flexible framework for CS of IS/VR:
It enables the dynamic coupling of arbitrary components to realize pure or hybrid implementations of the extended CS models described before. The framework focuses on the special requirements of this field of applications: interactivity and scalability. Interactivity is achieved by the communication server's global synchronization mechanisms and the methods for the elimination or punishment of producers of latency. Scalability is provided through the Publish/Subscribe service, which allows for the dynamic and flexible creation of communication links.

A platform for Remote Visualization on Hybrid Clusters:
The platform makes use of the specific characteristics of a hybrid cluster, i.e. its visualization component, by introducing methods to use the graphics hardware for general purpose computing tasks and by utilizing the implicit parallelism for distributed visualization scenarios. It forms the basis to develop new grabbing, composition and

compression techniques to allow for the efficient, remote usage of hybrid cluster systems for interactive applications.

New Parallel Image Compression Methods:
The two presented methods both use the powerful GPUs of the clusters visualization nodes to realize the necessary, fast and high-quality image compression for the remote visualization. By parallelizing well known image compression algorithms (Difference Coding (lossless) and JPEG (lossy)) and adapting them to the special GPU hardware, substantial performance increases and significant load relieving of the CPU could be achieved. Additionally, through outsourcing the image compression to the GPU and thereby avoiding the need to readback large uncompressed frames through the limited host interface, a further source of latency could be eliminated. All these improvements help to achieve the goal of providing a seamless graphical remote interface without restrictions in latency or quality.

7.2 Conclusion

The focus of this thesis was to enable a special class of applications (IS/VR) to utilize the power of upcoming hybrid cluster systems. Interactive simulations and Virtual Reality applications were chosen because they can greatly benefit from the computational and graphical power of hybrid clusters as shown in the scenarios in chapter 5. After introducing the problem and describing the foundations, including the terminology of hybrid cluster system and IS/VR applications, a comparison between traditional High Performance Computing and IS/VR applications was conducted and the need for new subsystems was pointed out. Those subsystems were derived from existing ones for traditional HPC applications, but were thoroughly adapted to the requirements of the new field. The most important requirements, which do not play a big role in traditional HPC, are interactivity and (soft) real-time execution. Thus, the presented subsystems strongly focus on techniques to fulfill these needs.

The first subsystem, computational steering of IS/VR, allows for the synchronization and coupling of arbitrary amounts of components, to create variable distributed IS/VR applications on the fly. The models that underlie this subsystem can be applied to most existing IS/VR applications and allow new functionalities (such as high resolution visualization, comparison of simulations or multi-user scenarios) with very little programming effort. Without computational steering for IS/VR those features were only realizable by completely redesigning the whole application and handling all communication and synchronization tasks manually. The system is on the one hand flexible enough to support a broad spectrum of applications and on the other hand offers specialized functions that help to fulfill the requirements of IS/VR in terms of interactivity and immersivity. To illustrate the potential of this approach, two very different applications were ported, with only minimal changes to their original source code, from static, sequential applications to flexible, distributed systems by utilizing

the models of computational steering for IS/VR.

The second subsystem, remote visualization for IS/VR, addresses another problem that those applications have when being executed on a universal cluster system: Accessability. Since IS/VR applications heavily depend on interactivity and the visualization of the simulated worlds, a direct access to the running simulation is essential. However, clusters often reside in highly inaccessible areas and are in many cases only reachable over the network (which is sufficient for most traditional HPC applications). Newer hybrid cluster systems (especially those which include visualization capabilities) in many cases feature one or more sophisticated output devices (stereoscopic display, tiled wall or a CAVE) as main sites of visualization and interaction. Those are fixed to one location and cannot be used simultaneously by multiple users. Thus, systems for the remote visual access are needed to successfully transfer IS/VR applications from desktop PCs to high performance clusters. The system designed and developed in this thesis aims to optimize latency and quality as well as scalability of the remote visual access. None of the systems for remote visualization that currently exist share this combination of goals. Most remote access frameworks focus on remote administration tasks where latency and performance play a minor role but the consumed bandwidth is the most important feature. Systems that focus on high quality remote visualization of interactive applications, either introduce big latencies by applying sophisticated compression algorithms that consume a lot of computation time or lack the desired flexibility to allow for example multiple users to simultaneously access the system remotely. The system introduced in this thesis is a framework to test, evaluate and utilize methods that help to achieve the goal of low latency, high quality and flexible remote visualization with adequate bandwidth consumption. One promising approach is to use the powerful graphics cards of the hybrid cluster not only for rendering, but also for compression and other computing tasks to significantly speed-up the process of grabbing and encoding the rendered frames. Two exemplary methods were developed and implemented inside the presented framework. Benchmarks showed that even in a prototypical stage the methods (especially GPU-based JPEG compression) could compete with and even outperform in some cases the standard methods used in this field.

All in all, the two developed systems form a basis to successfully deploy IS/VR applications on hybrid clusters. As pointed out by Pfister and in the introduction of this thesis, there currently is powerful hardware at hand, but the problem is that it is still to complicated and complex. Thus, the conceptual design as well as the development of middleware and subsystems that help the developers and users to harvest the raw computational and graphical power of such systems are vital. This not only holds for software for clusters but for most of the software that will be developed in the near future. The paradigm shift from sequential to parallel and distributed execution confronts the developers with the same tasks as in massive parallel software development: Load balancing, communication, consistency etc.. These issues can only be handled efficiently with the right tools (subsystems/middleware) at hand.

7.3 Outlook

The subsystems presented in this thesis can be seen as a starting point for further developments. They allow the generic execution of arbitrary IS/VR applications on hybrid clusters for the first time. However there is still work to be done to practically enable all potential users to utilize the powerful features of this combination. At first, both subsystems need to be permanently deployed on the cluster system and integrated into cluster management software, so that reservation and planning of usage is possible. The prototypes need to be transferred to stable programs, which also take care of e.g. security and authorization issues. Especially the simultaneous execution of various applications on both system needs to be tested and safeguarded. In addition, universal interfaces for the components connected through computational steering are thinkable. They would allow to couple components of (nearly) arbitrary applications in the same field and possibly ease the integration of new components. Finally, further compression methods for remote visualization need to be integrated and tested through the Invire framework to have a broad selection of techniques for all possible applications.

Through this thesis and the work done under the scope of the VisSim project[1], new topics and tasks have evolved. The systems that were presented before found the basis for several bachelor and master thesis. One thesis for example deals with the development of a generic benchmark for the computational steering framework in order to evaluate the scalability of the system and its potential for further areas of applications. Besides that there are several projects that are planned to make use of the hybrid cluster systems through the developed subsystems. They reach from soccer playing robots to medical image reconstruction. Finally, the software implementations of both systems will be released under GPL license in alpha versions after this thesis has been finished.

[1] A project funded by the German government to do cooperative research in the field of distributed visualization and simulation.

Bibliography

[1] Advanced Micro Devices. Homepage of AMD/ATI. Website: `http://ati.amd.com`, 2008.

[2] ANSYS. FLUENT - CFD Flow Modeling Software and Solutions. Website: `http://www.fluent.com`, 2008.

[3] Yukihiro Arai, Takeshi Agui, and Masayuki Nakajima. A Fast DCT-SQ Scheme for Images. *Transactions of IEICE*, E71(11):1095–1097, 1988.

[4] Moshe Bar. The openMosix Project. Website: `http://openmosix.sourceforge.net`, 2008.

[5] Jochen Bauch, Rafael Radkowski, and Henning Zabel. An Explorative Approach to the Virtual Prototyping of Self-optmizing Mechatronic Systems. In *Proceedings of ProSTEP iViP Science Days 2005 - Cross Domain Engineering, Darmstadt*, 2005.

[6] Heiko Bauke and Stephan Mertens. *Cluster Computing: Praktische Einführung in das Hochleistungsrechnen auf Linux-Clustern (german)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[7] Jan Berssenbrügge. *Virtual Nightdrive – Ein Verfahren zur Darstellung der komplexen Lichtverteilungen moderner Scheinwerfersysteme im Rahmen einer virtuellen Nachtfahrt (german)*. PhD thesis, Universität Paderborn, December 2005.

[8] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for VR Application Development. In *VR01: Proceedings of the IEEE Virtual Reality conference*, pages 89–96. IEEE, 2001.

[9] Stephan Blazy, Odej Kao, and Oliver Marquardt. padfem2 - An Efficient, Comfortable Framework for Massively Parallel FEM-Applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 681–685. Springer, 2003.

[10] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Computer Graphics*, 12(3):286–292, 1978.

[11] Maxime Crochemore and Wojciech Rytter Wojciech. *Jewels of stringology*. World Scientific Publishing Co. Inc., River Edge, NJ, 2003.

[12] Wilhelm Dangelmaier, Daniel Huber, Christoph Laroqueand Mark Aufenanger, Matthias Fischer, Jens Krokowski, and Michael Kortenjahn. d^3FACT insight goes parallel - Aggregation of multiple simulations. In *SimVis 07: Proceedings of the*

17th Simulation and Visualization Conference, pages 79–88. SCS European Publishing House, 2006.

[13] Jauvane C. de Oliveira, Shervin Shirmohammadi, and Nicolas D. Georganas. Collaborative Virtual Environment Standards: A Performance Evaluation. In *DIS-RT99: Proceedings of the 3rd International Workshop on Distributed Interactive Simulation and Real-Time Applications*, page 14, Washington, DC, USA, 1999. IEEE Computer Society.

[14] Declan Delaney, Tomás Ward, and Seamus McLoone. On consistency and network latency in distributed interactive applications: a survey – part I. *Presence: Teleoperations and Virtual Environments*, 15(2):218–234, 2006.

[15] Chrilly Donninger and Ulf Lorenz. The Chess Monster Hydra. In *FPL04: Proceedings of the 14th International Conference on Field Programmable Logic and Application*, pages 927–932. Springer, 2004.

[16] M. P. Eckert and A. P. Bradley. Perceptual quality metrics applied to still image compression. *Signal Processing*, 70:177–200, 1998.

[17] Martin Eikermann. A Flexible Framework for Computational Steering of Distributed High Performance Systems. Master's thesis, Universität Paderborn, September 2006.

[18] J. Joseph Brann et al. IEEE standard for distributed interactive simulation - application protocols. *IEEE Standard 1278.1-1995*, 26 March 1996.

[19] Konrad Etschberger. *Controller Area Network*. IXXAT Automation GmbH, August 2001.

[20] X.Org Foundation. The X.Org project. Website: `http://www.x.org/wiki`, 2008.

[21] Freedesktop.org. Introduction to D-Bus. `http://www.freedesktop.org/wiki/IntroductionToDBus`, 2008.

[22] Juergen Gausemeier, Jan Berssenbruegge, and Jochen Bauch. A Virtual Reality-based Night Drive Simulator for the Evaluation of a Predictive Advanced Front Lighting System. In *ASME CIE06: Proceedings of the ASME 2006 International Design Engineering Technical Conference and Computers and Information in Engineering Conference*. ASME, 2006.

[23] Gaussian Inc. The Official Gaussian Website. `http://www.gaussian.com`, 2008.

[24] GPGPU. General-Purpose Computation Using Graphics Hardware. Website: `http://www.gpgpu.org`, 2008.

[25] M. Harris. Parallel Prefix Sum (Scan) with CUDA. Website: `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf`, 2007.

[26] D. Horn. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Stream Reduction Operations for GPGPU Applications, pages 573–583. Addison-Wesley Professional, 2005.

[27] Paul G. Howard and Jeffrey Scott Vitter. Parallel lossless image compression using Huffman and arithmetic coding. *Information Processing Letters*, 59(2):65–73, 1996.

[28] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[29] Independent JPEG Group. libjpeg (Open Source JPEG library). Website: `http://www.ijg.org`, 2008.

[30] Intel. Intel® Integrated Performance Primitives 5.3. Website: `http://www.intel.com/cd/software/products/asmo-na/eng/302910.htm`, 2008.

[31] Jeff Juliano and Jeremy Sandmel. OpenGL Extension - Frame Buffer Objects and Pixel Buffer Objects. Website: `http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt`, 2005.

[32] Roy Kalawsky. *The Science of Virtual Reality and Virtual Environments*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[33] Jonathan Kaye and David Castillo. Interactive Simulation Newsletter Vol. 1, No. 1 (April, 2003). `http://www.flashsim.com/newsletter/v1n1.html`, 2003.

[34] T. Kesavadas and Abhishek Sudhir. Computational Steering in Simulation of Manufacturing Systems. In *ICRA00: Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2654–2658, 2000.

[35] Khronos Group. The OpenGL Standard. Website: `http://www.opengl.org/`, 2008.

[36] James Arthur Kohl, Philip M. Papadopoulos, and G. A. Geist II. CUMULVS: Collaborative Infrastructure for Developing Distributed Simulations. In *PPSC97: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.

[37] David Koller, Michael Turitzin, Marc Levoy, Marco Tarini, Giuseppe Croccia, Paolo Cignoni, and Roberto Scopigno. Protected interactive 3D graphics via remote rendering. In *SIGGRAPH '04: Proceedings of the ACM SIGGRAPH conference*, pages 695–703, New York, NY, USA, 2004. ACM.

[38] Kronos Group. GLSL - The OpenGL Shading Language. Website: `http://www.opengl.org/documentation/glsl`, 2008.

[39] Christoph Laroque. *Ein mehrbenutzerfähiges Werkzeug zur Modellierung und richtungsoffenen Simulation von wahlweise objekt- und funktionsorientiert gegliederten Fertigungssystemen (german)*. PhD thesis, Universität Paderborn, June 2007.

[40] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw Hill Higher Education, 2000.

[41] Paul Hermann Lensing. GPU-basierte, verlustbehaftete Bildkompression für Remote Rendering (german). Master's thesis, Universität Paderborn, March 2008.

[42] Stefan Lietsch and Jan Berssenbruegge. Parallel, Shader-Based Visualization of Automotive Headlights. In *EGPGV07: Proceedings of the 7th Eurographics Symposium on Parallel Graphics and Visualization*. ACM, 2007.

[43] Stefan Lietsch and Paul Hermann Lensing. CUDA-based, parallel JPEG Compression for Remote Rendering. In *ISIVC08: Proceedings of the 4th International Symposium on Image/Video Communications over fixed and mobile networks*. IEEE, 2008.

[44] Stefan Lietsch and Oliver Marquardt. A CUDA-Supported Approach to Remote Rendering. In *ISVC07: Proceedings of the International Symposium on Visual Computing*, volume 4841 of *Lecture Notes in Computer Science*, pages 724–733. Springer, 2007.

[45] Stefan Lietsch, Henning Zabel, and Jan Berssenbruegge. Computational Steering of Interactive and Distributed Virtual Reality Applications. In *ASME CIE07: Proceedings of the 27th ASME Computers and Information in Engineering Conference*. ASME, 2007.

[46] Stefan Lietsch, Henning Zabel, Jan Berssenbruegge, Veit Wittenberg, and Martin Eikermann. Light Simulation in a Distributed Driving Simulator. In *ISVC06: Proceedings of the International Symposium on Visual Computing*, volume 4291 of *Lecture Notes in Computer Science*, pages 343–353. Springer, 2006.

[47] Stefan Lietsch, Henning Zabel, and Christoph Laroque. Computational Steering Of Interactive Material Flow Simulations. In *ASME CIE08: Proceedings of the 28th ASME Computers and Information in Engineering Conference*. ASME, 2008.

[48] J. L. Mannos and D. J. Sakrison. The effects of a visual fidelity criterion on the encoding of images. *IEEE Transactions on Information Theory*, IT-4:525–536, 1974.

[49] Mechdyne. CAVElib - Build a better reality. Website: `http://www.mechdyne.com/integratedSolutions/software/products/CAVELib/CAVELib.htm`, 2008.

[50] Message Passing Interface Forum. MPI: A message passing interface standard.

[51] Microsoft. Microsoft Windows Server 2003 Terminal Services. Website: `http://www.microsoft.com/windowsserver2003/technologies/terminalservices/default.mspx`, 2007.

[52] Microsoft. The Microsoft DirectX Collection. Website: `http://msdn.microsoft.com/directX`, 2008.

[53] Microsoft. Windows Compute Cluster Server 2003. Website: `http://www.microsoft.com/windowsserver2003/ccs`, 2008.

[54] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. Legall, editors. *MPEG Video Compression Standard*. Chapman & Hall, Ltd., London, UK, UK, 1996.

[55] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[56] Jurriaan D. Mulder, Jarke J. van Wijk, and Robert van Liere. A survey of computational steering environments. *Future Generation Computer Systems*, 15(1):119–129, 1999.

[57] NVIDIA. CG - The C for Graphics Language. Website: `http://developer.nvidia.com/page/cg_main.html`, 2008.

[58] NVIDIA. GeForce3 - The Infinite Effects GPU. Website: `http://www.nvidia.com/page/geforce3.html`, 2008.

[59] NVIDIA. Homepage of NVIDIA. Website: `http://www.nvidia.com`, 2008.

[60] NVIDIA. NVIDIA CUDA - Compute Unified Device Architecture. Website: `http://www.nvidia.com/object/cuda_home.html`, 2008.

[61] NVIDIA. NVIDIA CUDA - Compute Unified Device Architecture - Programming Guide v1.1. PDF: `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`, 2008.

[62] Object Management Group. Unified Modeling Language Specification, Version 1.5, March 2003.

[63] Kyoung S. Park, Yong J. Cho, Naveen K. Krishnaprasad, Chris Scharver, Michael J. Lewis, Jason Leigh, and Andrew E. Johnson. CAVERNsoft G2: a toolkit for high performance tele-immersive collaboration. In *VRST00: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 8–15, New York, NY, USA, 2000. ACM Press.

[64] Steven G. Parker and Christopher R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *SUPCOM95: Proceedings of Supercomputing*, San Diego, CA, December 1995. ACM/IEEE.

[65] PC^2. Paderborn Center for Parallel Computing. Website: `http://wwwcs.uni-paderborn.de/pc2`, 2008.

[66] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[67] Gregory F. Pfister. *In search of clusters (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[68] Marco Platzner, Sven Döhre, Markus Happe, Tobias Kenter, Ulf Lorenz, Tobias Schumacher, Andre Send, and Alexander Warkentin. The GOmputer: Accelerating GO with FPGAs. In *ERSA08: Proceedings of the 8th International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages –, Las Vegas, Nevada, USA, 2008. CSREA Press.

[69] Luc Renambot, Henri E. Bal, Desmond Germans, and Hans J. W. Spoelder. CAVEStudy: An Infrastructure for Computational Steering and Measuring in Virtual Reality Environments. *Cluster Computing*, 4(1):79–87, 2001.

[70] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[71] D. Salomon. *Data Compression: The Complete Reference, 3rd Edition*. Springer, 2004.

[72] SGI. SGI OpenGL Vizserver - Visual Area Networking. Website: `http://www.sgi.com/products/software/vizserver/`, 2007.

[73] Silicon Graphics, Inc. Homepage of Silicon Graphics, Inc. Website: `www.sgi.com`, 2008.

[74] Jens Simon. Paderborn Center for Parallel Computing - Benchmarking Center. Website: `http://wwwcs.uni-paderborn.de/pc2/about-us/staff/jens-simons-pages/benchmarkingcenter.html`, 2008.

[75] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen. GROMACS: fast, flexible, and free. *Jornal on Computational Chemistry*, 26(16):1701–1718, December 2005.

[76] Jarke J. van Wijk, Robert van Liere, and Jurriaan D. Mulder. Bringing Computational Steering to the User. In *Proceedings of the Scientific Visualization Conference 1997*, pages 304–313, 1997.

[77] Jeffrey Vetter and Karsten Schwan. High Performance Computational Steering of Physical Simulations. In *IPPS97: Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, April 1997. The Institute of Electrical and Electronics Engineers.

[78] VirtualGL. The VirtualGL Project. Website: `http://www.virtualgl.org/`, 2007.

[79] VirtualGL. A Study of the Performance ofVirtualGL 2.1 and TurboVNC 0.4. PDF: `http://www.virtualgl.org/pmwiki/uploads/About/vglperf21.pdf`, 2008.

[80] VirtualGL. TurboJPEG 1.10 - Intel IPP accelerated JPEG compression. Website: `http://sourceforge.net/project/showfiles.php?group_id=117509\&package%_id=166100`, 2008.

[81] Z. Wang, A. Bovik, and L. Lu. Why is image quality assessment so difficult. In *ICASSP02: Proceedings of the 27th IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 3313–3316. IEEE, 2002.

[82] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13:600–612, 2004.

A

Acronyms

AGP Accelerated Graphics Port

APDU Application Protocol Data Unit

API Application Programming Interface

BPP Bytes Per Pixel

CAD Computer Aided Design

CFD Computational Fluid Dynamics

CG C for Graphics

CPU Central Processing Unit

CUPS Common Unix Printing Service

CS Computational Steering

CSIS Computational Steering of Interactive Simulations

CUDA Compute Unified Device Architecture

DCT Discrete Cosine Transformation

DIA Distributed Interactive Application

DIC Difference with Index Compression

DIS Distributed Interactive Simulation

DTD Document Type Definition (XML)

FEM Finite Element Methods

FPGA Field Programmable Gate Arrays

FPS Frames Per Second

GLSL OpenGL Shading Language

GLUT OpenGL Utility Toolkit

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HAL Hardware Abstraction Layer

HCA Host Channel Adapter

HPC High Performance Computing

HVS Human Visual System

IPP Intel(R) Integrated Performance Primitives

Invire Interactive Remote Visualization

IS/VR Interactive Simulation and Virtual Reality

JPEG Joint Photographic Experts Group

MD Molecular Dynamics

MFS Material Flow Simulation

MPEG Moving Pictures Expert Group

MPI Message Passing Interface

MSE Mean Squared Error

NTSC National Television System Committee

OS Operating System

PAL Phase Alternating Line

PC Personal Computer

PC^2 Paderborn Center for Parallel Computing

PCI Peripheral Component Interconnect

P/S Publish/Subscribe

PSNR Peak Signal-to-Noise Ratio

PVM Parallel Virtual Machine

RAID Redundant Array of Inexpensive/Independent Disks

RGB(A) Red Green Blue (Alpha)

RLE Run Length Encoding

RPC Remote Procedure Call

RV Remote Visualization

SIMD Single Instruction Multiple Data

SSIM Structural Similarity

TCP Transmission Control Protocol

UDP User Datagram Protocol

UML Unified Modeling Language

(U)XGA (Ultra) Extended Graphics Array

VND Virtual Night Drive

VSD Volatile State Data

VR Virtual Reality

XML Extensible Markup Language

XSL Extensible Stylesheet Language (XML)

YCbCr Luma Chroma Blue Chroma Red Color Space