INTERACTIVE PARALLEL POST-PROCESSING OF SIMULATION RESULTS ON UNSTRUCTURED GRIDS

Florian Niebling
INTERACTIVE PARALLEL POST-PROCESSING OF SIMULATION RESULTS ON UNSTRUCTURED GRIDS

von der Fakultät Energie-, Verfahrens- und Biotechnik der Universität Stuttgart zur Erlangung der Würde eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Florian Niebling
aus Waiblingen

Hauptberichter: Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Michael Resch
Mitberichter: Prof. Dr.-Ing. Eberhard Göde
Tag der mündlichen Prüfung: 26. August 2013
CR-Klassifikation: I.3.2, I.6.6

ISSN 0941 - 4665 Januar 2014 HLRS-11
D93
Contents

List of Figures .................................................. viii
List of Listings .................................................. ix
List of Abbreviations ............................................ x
Chapter Outline .................................................. xiii
Abstract .......................................................... xiv
Deutsche Zusammenfassung ................................. xiv

1. Introduction .................................................. 1
   1.1. Contribution ............................................. 4

2. State of the Art and Technological Background ...... 5
   2.1. Post-processing Workflow ............................... 5
       2.1.1. Visualization Pipeline ......................... 5
       2.1.2. Rendering Pipeline ............................. 6
   2.2. Flow Simulation Data ................................... 7
       2.2.1. Grid Types ..................................... 7
       2.2.2. Data Types ................................... 9
   2.3. Flow Visualization ................................... 9
       2.3.1. Surface Extraction ............................ 10
       2.3.2. Particle Tracing ............................... 11
       2.3.3. Commonly Used Data Structures for Unstructured Grids 12
   2.4. Post-processing Resources ........................... 14
       2.4.1. Levels of Parallelism .......................... 15
       2.4.2. GPGPU Hardware .............................. 15
       2.4.3. Performance Considerations in GPGPU Computing .... 16
       2.4.4. Parallel Rendering in Scientific Computing .... 17

3. Related Work ................................................ 23
   3.1. Parallel Post-processing Infrastructure ........... 23
       3.1.1. ParaView ...................................... 24
       3.1.2. VisIt .......................................... 25
       3.1.3. Viracocha ..................................... 26
   3.2. Parallel Post-processing Algorithms ............... 27
       3.2.1. Surface Extraction ............................ 27
### Contents

3.2.2. Particle Tracing ........................................... 29
3.2.3. Texture-based Dense Flow Visualization ................. 30

4. Interactive Post-Processing in Parallel Manycore Environments ........................................... 33
  4.1. Utilizing Visualization Clusters ................................. 33
  4.2. Interactive Post-Processing in Hybrid Manycore Systems ............ 34
    4.2.1. Simulation Datasets .................................... 35
    4.2.2. Distributed Data Objects ................................. 36
    4.2.3. Dataflow Architecture .................................. 36
    4.2.4. Hybrid Parallel Post-Processing Algorithms ............... 38
  4.3. A Dataflow oriented, Parallel Post-Processing Environment ........ 39
    4.3.1. Employing Different Levels of Parallelism ............... 40
    4.3.2. Parallel Runtime Environment ............................ 41
    4.3.3. Components of the Post-Processing Environment .......... 44

5. Particle Tracing .................................................. 49
  5.1. Introduction .................................................. 49
  5.2. Parallel Particle Tracing on GPGPUs .......................... 50
    5.2.1. Initial Cell Location .................................... 51
    5.2.2. Tetrahedral Decomposition ................................ 51
    5.2.3. Numerical Integration .................................... 52
    5.2.4. Neighbor Search ......................................... 53
    5.2.5. Parallelization .......................................... 53
    5.2.6. Particle Rendering ....................................... 54
  5.3. Massive Particle Tracing for Erosion Prediction in Turbomachinery .................. 56
    5.3.1. Tracking Massive Particles in Unstructured Grids .......... 56
    5.3.2. Collision Detection ..................................... 59
    5.3.3. Tabakoff and Grant Erosion Model ........................ 59
    5.3.4. Visualization of Erosion .................................. 61
  5.4. Performance .................................................. 62

6. Surface Extraction ............................................... 63
  6.1. Extraction of Surfaces from Unstructured Grids ................ 63
    6.1.1. Surface Extraction Algorithm ............................. 64
    6.1.2. Parallel Manycore Implementation ......................... 65
  6.2. Texture-based Feature Visualization .......................... 70
    6.2.1. Parallel Line Integral Convolution on Manycore Devices: FragmentLIC .... 70
    6.2.2. Performance Considerations on GPGPUs ..................... 74
7. Summary and Conclusion 77
   7.1. Contributions ............................................. 77
   7.2. Conclusions ................................................. 78
   7.3. Outlook ..................................................... 79
      7.3.1. Massively Parallel Post-Processing Algorithms .... 79
      7.3.2. Reliability and Fault Tolerance in HPC Algorithms ... 80
      7.3.3. Massively Parallel Rendering of Large Post-Processing
              Results ................................................. 80

A. Unstructured Grids 81
   A.1. Data Structures ........................................... 81
   A.2. Decomposition of Grid Elements into Tetrahedra .......... 82
   A.3. Marching Elements for Surface Extraction ............... 82

B. GLSL Shaders 85
   B.1. LIC on GPU ............................................... 85
   B.2. Particle Rendering ........................................ 86

C. Datasets 87
   C.1. Francis Turbine Runner .................................... 87
   C.2. Coal Combustion Chamber ................................ 88
   C.3. Data center ............................................... 89

Bibliography 91

Index 103
## List of Figures

1. Virtual Product Development (VPD) workflow ........................................ 1
2. Visualization of radial water turbine runner CFD simulation data ........ 2
3. Visualization Pipeline ............................................................................. 6
4. Simplified rendering pipeline ................................................................. 7
5. Structured grid types ............................................................................. 8
6. Unstructured grid types ......................................................................... 9
7. Marching tetrahedra triangle lookup table .............................................. 11
8. Space partitioning using Cell-Tree .......................................................... 13
9. Comparison of isosurface extraction performance using Cell
   Tree, Interval Tree and traditional implementation .................................. 14
10. NVIDIA GPU thread hierarchy and memory layout .............................. 16
11. Coalesced memory access patterns on different NVIDIA GPU
    generations ............................................................................................ 17
12. Performance of global memory access patterns on different
    NVIDIA GPU generations .................................................................... 18
13. Sort-last rendering .................................................................................. 20
14. *Binary Swap* parallel composition process ......................................... 21
15. Architecture of a parallel post-processing environment ........................ 39
16. Different levels of parallelism. ................................................................. 42
17. Sequence diagram of the creation of a shared memory object
    during execution of a module partition .................................................. 45
18. Description of the *vistle* file format header ........................................ 47
19. Performance of sort-last rendering on 1024 nodes of CRAY XE6
    HERMIT .................................................................................................. 48
20. Decomposition of neighboring hexahedral cells .................................... 52
21. GLSL particle rendering using GL_POINTS .......................................... 54
22. Massively parallel particle rendering on the GPU using GLSL
    shaders ................................................................................................... 55
23. Erosion in a radial water turbine runner ............................................... 56
24. Erosion model of Tabakoff/Grant .......................................................... 60
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.</td>
<td>Particle paths and isosurfaces of particle density for several eroded segments</td>
<td>61</td>
</tr>
<tr>
<td>26.</td>
<td>Performance of CPU and GPU implementation of tracing of massless and massive particles</td>
<td>62</td>
</tr>
<tr>
<td>27.</td>
<td>Extracted surfaces showing pressure distribution in a radial water turbine CFD simulation</td>
<td>64</td>
</tr>
<tr>
<td>28.</td>
<td>Performance of isosurface computation on CPU/GPU</td>
<td>65</td>
</tr>
<tr>
<td>29.</td>
<td>Parallel GPGPU surface extraction</td>
<td>66</td>
</tr>
<tr>
<td>30.</td>
<td>Comparison between face- and vertex normals</td>
<td>67</td>
</tr>
<tr>
<td>31.</td>
<td>Surface extraction runtime performance on different generation GPUs</td>
<td>68</td>
</tr>
<tr>
<td>32.</td>
<td>Visualization of temperature distribution in a coal combustion chamber simulation using surface extraction methods</td>
<td>69</td>
</tr>
<tr>
<td>33.</td>
<td>Visualization of vortices in data center air cooling, including additional background geometry</td>
<td>71</td>
</tr>
<tr>
<td>34.</td>
<td>First LIC render pass showing triangle boundaries, world coordinates of fragments and USG element index per fragment</td>
<td>72</td>
</tr>
<tr>
<td>35.</td>
<td>Second LIC pass: CUDA-based image post-processing applying tracing of streamlines to compute convolution integrals per fragment</td>
<td>73</td>
</tr>
<tr>
<td>36.</td>
<td>Visualization of flow at the inlet of a radial water turbine</td>
<td>74</td>
</tr>
<tr>
<td>37.</td>
<td>Performance of FragmentLIC depending on the size of the generated image, using a fixed number of triangles extracted from the datacenter dataset</td>
<td>75</td>
</tr>
<tr>
<td>38.</td>
<td>Performance of FragmentLIC depending on the number of triangles extracted from the datacenter dataset, using a fixed number of fragments (1.048.576)</td>
<td>75</td>
</tr>
<tr>
<td>39.</td>
<td>Data structures allowing for coordinate-sharing in USGs</td>
<td>81</td>
</tr>
<tr>
<td>40.</td>
<td>Decomposition of USG elements into tetrahedra</td>
<td>82</td>
</tr>
<tr>
<td>41.</td>
<td>Marching tetrahedra triangle lookup table</td>
<td>83</td>
</tr>
<tr>
<td>42.</td>
<td>Marching pyramids triangle lookup table</td>
<td>83</td>
</tr>
<tr>
<td>43.</td>
<td>Marching prisms triangle lookup table</td>
<td>83</td>
</tr>
<tr>
<td>44.</td>
<td>Original marching cubes triangle lookup table</td>
<td>83</td>
</tr>
<tr>
<td>45.</td>
<td>Geometry of the radial water turbine runner</td>
<td>87</td>
</tr>
<tr>
<td>46.</td>
<td>Surface grid of the high resolution CFD model</td>
<td>88</td>
</tr>
<tr>
<td>47.</td>
<td>Numerical simulation of combustion in a coal power plant</td>
<td>88</td>
</tr>
<tr>
<td>48.</td>
<td>Numerical simulation of airflow in a data center</td>
<td>89</td>
</tr>
</tbody>
</table>
Listings

B.1. GLSL vertex shader for first LIC render pass . . . . . . . . . 85
B.2. GLSL fragment shader for first LIC render pass . . . . . . . . 85
B.3. GLSL vertex shader for particle rendering using GL_POINTS . 86
B.4. GLSL fragment shader for particle rendering using GL_POINTS 86
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AABB</td>
<td>Axis-aligned Bounding Box</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AUFLIC</td>
<td>Accelerated Unsteady Flow LIC</td>
</tr>
<tr>
<td>BSP</td>
<td>Block-synchronous Parallel Program</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CFD</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DDA</td>
<td>Digital Differential Analyzer</td>
</tr>
<tr>
<td>FBU</td>
<td>Frequency Based Replacement</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL Shading Language</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Computing on Graphics Processing Units</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>IBFV</td>
<td>Image-based Flow Visualization</td>
</tr>
<tr>
<td>IBFVS</td>
<td>Image-based Flow Visualization for Surfaces</td>
</tr>
<tr>
<td>ISA</td>
<td>Image Space Advection</td>
</tr>
<tr>
<td>KD tree</td>
<td>K-Dimensional tree</td>
</tr>
<tr>
<td>LIC</td>
<td>Line Integral Convolution</td>
</tr>
<tr>
<td>LFU</td>
<td>Least Frequently Used</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>ms</td>
<td>millisecond(s)</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>PVM</td>
<td>Parallel Virtual Machine</td>
</tr>
<tr>
<td>RGB</td>
<td>Red-Green-Blue</td>
</tr>
<tr>
<td>RGBA</td>
<td>Red-Green-Blue-Alpha</td>
</tr>
<tr>
<td>RK4</td>
<td>Runge-Kutta of 4-th order</td>
</tr>
<tr>
<td>RK5</td>
<td>Runge-Kutta of 5-th order</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction, Multiple Threads</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
</tr>
<tr>
<td>TCL</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLT</td>
<td>Triangle Lookup Table</td>
</tr>
<tr>
<td>UFLIC</td>
<td>Unsteady Flow LIC</td>
</tr>
<tr>
<td>USG</td>
<td>Unstructured Grid</td>
</tr>
<tr>
<td>VBO</td>
<td>Vertex Buffer Object</td>
</tr>
<tr>
<td>VPD</td>
<td>Virtual Product Development</td>
</tr>
<tr>
<td>VTK</td>
<td>Visualization Toolkit</td>
</tr>
</tbody>
</table>
Abstract and Chapter Outline

Chapter Outline

This thesis is structured as follows. Chapter 1 states the purpose and context of this work. Chapter 2 describes the foundations and technologies on which this work is built. Chapter 3 reviews the state of the art and work related to parallel post-processing of fluid simulation data in the field of High Performance Computing (HPC).

Chapter 4 states the analysis of the problem statement and introduces a framework that allows for interactive parallel post-processing of simulation data containing large unstructured grids.

Chapter 5 presents methods for the tracing of both massless and massive particles using the software architecture described in chapter 4. Chapter 6 discusses techniques for massively parallel surface extraction from unstructured grids (USGs). Texture-based visualization methods are proposed that benefit from accelerators such as GPGPU devices in parallel environments.

Chapter 7 concludes this thesis by giving a summary of the work as well as an outlook on future developments. The appendices give additional information on datastructures, shader programs and the simulation datasets that have been used to evaluate the methods developed during the establishment of this work.
Abstract

Numerical simulations and the assessment of their results are constantly gaining importance in product design and optimization workflows in many different fields of engineering. The availability of massively parallel manycore computing resources enables simulations to be executed with accuracies posing very high requirements on the methods for interactive post-processing of the simulation results. A traditional post-processing of such large-scale simulation datasets on single workstations is often no longer possible due to the limited resources such as main memory, the low number of compute cores and the available network bandwidth to the simulation cluster.

In this work, concepts and solutions are presented that enable interactive post-processing of large-scale datasets generated by fluid dynamic simulations on unstructured grids through the use of parallel manycore environments. A software architecture the parallel post-processing and visualization, as well as specific optimizations of frequently used methods for post-processing are introduced that enable the interactive use of parallel manycore resources.

The implementation of the methods and algorithms is based on existing manycore devices in the form of programmable graphics hardware, which are no longer solemnly usable for computer graphics applications, but are getting increasingly interesting for general purpose computing. It will be shown, that methods for visualization of fluid simulation data such as the interactive computation of cut-surfaces or particle traces is made possible even for large-scale unstructured data. Additionally, an algorithm for the dense texture-based visualization of flow fields will be introduced that combines the presented methods for the extraction of cut-surfaces, isosurfaces and particle tracing. This algorithm for line integral convolution enables the interactive post-processing of flow fields on partitioned and distributed unstructured grids.

The methods introduced in this thesis are evaluated using several large-scale simulation datasets from different fields of engineering in scientific and industrial applications.
Abstract

Deutsche Zusammenfassung


In dieser Arbeit werden Konzepte und Lösungen vorgestellt, mit denen eine interaktive Nachbearbeitung großer Datenmengen aus Strömungssimulationen auf unstrukturierten Gittern durch den Einsatz paralleler Manycore Rechenressourcen ermöglicht wird. Hierbei wird sowohl eine Softwarearchitektur für die parallele Nachbearbeitung und Visualisierung vorgestellt, als auch konkrete Optimierungen häufig genutzter Methoden der Nachbearbeitung für den interaktiven Einsatz in parallelen Manycore Umgebungen beschrieben.

Die Evaluierung der vorgestellten Verfahren erfolgt mittels der Nachbearbeitung großer Simulationsdatensätze aus Forschung und industrieller Nutzung in verschiedenen ingenieurwissenschaftlichen Anwendungsgebieten.
Chapter 1.

Introduction

The purpose of computing is insight, not numbers.

(Richard Hamming)

Numerical simulations executed on high performance computing (HPC) resources have become an integral part of scientific engineering workflows. They replace or add to model tests that may be infeasible due to their high costs, prohibitive time requirements or even health hazards. In virtual product development (VPD) and digital prototyping, simulations enable engineers to carry out more product revisions during the development cycle, offering potential for cost reductions, faster product development and product improvement.

Figure 1.: Virtual Product Development (VPD) workflow
Chapter 1. Introduction

To interpret results from large scale numerical simulations, methods supporting extensive analysis of the data are indispensable. Exploration of datasets in the domain of scientific engineering, used for feasibility studies, early identification of errors or design optimization, is a highly interactive process. As a direct evaluation of numerical datasets is impracticable, methods for post-processing of simulation results are essential. During post-processing, multi-dimensional raw data is transformed into visual information. Many different methods have been developed to allow for the analysis of characteristics of data generated by computational fluid dynamics (CFD) simulations.

Indirect visualization, the focus of this work, relies upon the construction or extraction of geometric primitives such as points, lines or triangular meshes, and the mapping of simulation data to these primitives using glyphs or colors. Often used indirect visualization approaches include the extraction of isosurfaces, slices and geometry created by particle tracing methods. Using a combination of multiple post-processing algorithms on a single dataset allows for different representations of characteristics of the underlying data.

![Visualization of radial water turbine runner CFD simulation data](Dataset C.1)

The exploration of multi-dimensional datasets is massively enhanced by low latency, interactive post-processing algorithms. Direct manipulation requires that the user have some immediate visual feedback reflecting the user’s control motions, i.e. a delay of less than 100ms [8]. This can be achieved through
employing massively parallel compute resources not only for running the CFD simulation itself, but also for interactive post-processing during analysis of the data. Such a post-processing cluster typically consists of multiple nodes containing several multi-core CPUs and one or more graphics processing units (GPUs) each. Current generation GPUs themselves are highly parallel devices consisting of hundreds comparably simple processor cores that can be used for rendering as well as for general purpose processing.

A software architecture used for interactive post-processing of large amounts of simulation data on parallel hybrid visualization clusters has to consider several necessities:

1. **Memory architecture**: Simulation data and data-structures used for post-processing have to be shared between different CPU- and GPU processors and between different algorithms. Data has to be decomposed to reduce the number of data elements to be processed by each of the processor cores during execution of a post-processing algorithm allowing for interactive response times.

2. **Preserving locality of data**: Moving data between different hosts or between host and GPU device memory has to be avoided during interactive post-processing of the dataset.

3. **GPGPU computing**: To reduce latency and to address bandwidth limitations between host and device memory, GPGPU computing, i.e. post-processing of simulation data on general purpose graphics processing units, has to be used to prepare the final data for rendering.

4. **Parallel rendering**: For large amounts of post-processed data, copying of graphics primitives to a single GPU for rendering limits the speed of the post-processing environment. Parallel rendering methods have to be used to employ multiple GPUs in the cluster for rendering of graphics primitives and composition of the resulting images.
1.1. Contribution

1. A model and a framework for parallel post-processing allowing for the execution of interactive post-processing algorithms on massively parallel visualization clusters.

2. Parallel surface extraction methods for unstructured grids (USGs) and the mapping of flow fields to these surfaces without resampling or surface-projection of the flow field using line integral convolution (LIC).

Chapter 2.

State of the Art and Technological Background

The data represent an abstraction of reality in the sense that certain properties and characteristics of the real objects are ignored because they are peripheral and irrelevant to the particular problem. An abstraction is thereby also a simplification of the facts.

(Niklaus Wirth)

2.1. Post-processing Workflow

Workflows describing the post-processing of simulation data are used to formalize knowledge about the extraction of information from these datasets. Albeit workflows used in the analysis of Computational Fluid Dynamics (CFD) simulations are usually dependent on the type of simulation performed.

2.1.1. Visualization Pipeline

The Visualization Pipeline describes an abstraction to post-processing workflows from data acquisition to the display of extracted information in the form
of images. The goal of the transformations on the simulation data, is “to convert the information (...) to a format amenable to understanding by the human perceptual system while maintaining the integrity of the information” [30]. For the

![Visualization pipeline](image)

Figure 3.: Visualization pipeline (adapted from Haber et al. [30])

purpose of interactive post-processing of simulation data, operations in the Visualization Pipeline can be grouped regarding the user interactions they permit. While data acquisition and analysis, acquiring simulation data and smoothing, interpolating or correcting require very few interactions to be performed by the post-processing engineer, the filtering and mapping of data depend on expert knowledge and is highly interactive as shown in Figure 3. Filtering, the selection of portions of data to be visualized, and mapping, the forming of geometric primitives from focus data, therefor are interesting targets for optimization of throughput and latency in an interactive system.

### 2.1.2. Rendering Pipeline

The rendering pipeline closely resembles the operations implemented by GPU hardware for the generation of images from primitives by using rasterization. Over the last years, the fixed-function rendering pipeline shown in Figure 4 has been largely replaced by the possibility for the programmer to implement specific portions of the rendering pipeline using programmable operations called shaders.

In addition to effects such as per fragment lighting that have not been implemented in the fixed function pipeline, shaders provide the basic possibility to use the GPU for computations other than rendering. Being replaced in recent
2.2. Flow Simulation Data

Flow simulation data are represented by data elements mapped to specific locations $x$ within an $n$-dimensional spatial domain $D \subseteq \mathbb{R}^n$ at timesteps $t \in I \subseteq \mathbb{R}$.

The topology of a dataset is considered to be the relationship of discrete data samples that are invariant with respect to geometric transformations, whereas the geometry is a specification of the topology in space [41].

2.2.1. Grid Types

Depending on the topology of the data elements in a flow field computed by a CFD solver, the results can be categorized into mesh-less, structured, and unstructured data.
2.2.1.1. Structured Grids

Structured grids have an implicit structure given by the location of the data in memory. Their structure is expressed as multi dimensional arrays. In practical applications, cells in structured grids are limited to quadrilaterals in 2D and hexahedra in 3D.

![Structured grid types](image)

Figure 5.: Structured grid types

Structured grids can be further categorized according to specific characteristics. Cartesian grids are equidistant and orthogonal in every dimension, whereas general rectilinear grids do not have to be equidistant. Curvilinear grids can be transformed from curvilinear physical space (P-space) to a rectilinear grid in computational space (C-space).

2.2.1.2. Unstructured Grids

In unstructured grids (USGs), each vertex position has to be stored separately. The topology, i.e. the connectivity between these vertices must be explicitly defined. In transient simulations, locations of vertices as well as the topology need not be static between different timesteps. In contrast to simulation data on structured grids, proximity of data in world coordinates generally does not have to correlate with proximity in memory in neither dimension, which makes optimization of memory accesses difficult for the general case. Since there is no implicit structure inherent in the topology, neighbor relationship between cells has to be explicitly stored. USGs typically consist of an array of elements, an array specifying connectivity, and an array of points (see Figure 39). Using
the information stored in the connectivity array, cells are built from points as 2D or 3D elements such as lines, triangles, tetrahedra or hexahedra. Indirect addressing can be used to allow for the sharing of vertices between neighboring cells. One of the significant advantages of using unstructured grids in numerical simulations is the possibility to perform local refinements in a certain region, without affecting the grid point distribution outside that region [34].

2.2.2. Data Types

Data generated by CFD simulations normally is given at discrete locations either per vertex or per cell of the computational grid. The reconstruction of continuous functions that are examined in this work is performed by linear interpolation of the available surrounding data elements, effectively producing piecewise linear functions. The dimensionality of the data ranges from scalar values such as pressure or temperature, 1st order tensor data such as velocity of the flow field, to higher order tensor data such as stress and momentum.

2.3. Flow Visualization

The principal idea in flow visualization is that data has a natural spatial representation. Interactive flow visualization techniques are designed to aid the user
in the exploration and analysis of information contained in a dataset, and to provide a basis for confirmation and presentation of results. Visualization is usually an iterative process and a user may generate additional data through interactive exploration of visualised data to gain better understanding of the nature of the information [63].

### 2.3.1. Surface Extraction

Surface extraction techniques to generate cut-surfaces or isosurfaces are one of the most commonly used post-processing techniques [25]. The extraction of meaningful surfaces from large-scale datasets allows for a substantial reduction in the number of data elements to be displayed and comprehended. The visualization of scalar values on extracted surfaces is easily done by using color-coding techniques. Vector values such as flow fields are therefore often reduced to scalar values, e.g. the magnitude of velocity vectors. The most common techniques for the extraction of surfaces from structured and unstructured data are based on the marching cubes family of algorithms.

#### 2.3.1.1. Marching Cubes

_Marching cubes_ has originally been created for the extraction of surfaces from structured grids [54]. The basic idea of the marching cubes algorithm is that for a surface to be located inside a cell of a grid, at least two of the vertices of a cell have to be on opposing sides of the surface. The first step in surface extraction in a single cell is a classification of vertices to be either on the _inside_ or on the _outside_ of the surface to be extracted. For isosurfaces, i.e. surfaces through points of equal value regarding a given data field, this classification of vertices happens by comparing the data mapped at these vertices to a certain reference value, called isovalue. When extracting geometric cut-surfaces such as cut-planes, cut-cylinders or cut-spheres, the vertices are classified according to their distance to the specified surface. The classification of all vertices in a cell is then used to extract a number of triangles approximating the surface inside the cell by consulting a _triangle lookup table_ (TLT). Due to symmetry,
the 256 different possible configurations of vertex values in a hexahedron can be reduced to 15 entries in the TLT as can be seen in Figure 44. The specific vertex positions of the extracted triangles can be calculated by interpolation of vertex positions along the edges of the grid elements.

In the original implementation, cracks can form in the resulting mesh. In consecutive work [11], ambiguous cases in the original TLT table are identified and handled separately, [64] show how these ambiguous cases can be processed without referring to the configuration of neighboring cells.

![Figure 7.: Marching tetrahedra triangle lookup table](image)

The original *marching cubes* algorithm has since been extended to extract surfaces from tetrahedral grids (see Figure 7) and unstructured grids containing e.g. tetrahedra (Figure 41), pyramids (Figure 42), prisms (Figure 43) or cubes (Figure 44).

### 2.3.2. Particle Tracing

Particle tracing to generate animated particles, stream lines, path lines, streak lines, time lines or stream surfaces are an important tool in the visualization of flow fields. The rendering of animated particles and streak lines are especially interesting methods for the extraction of information from flow fields, since they correspond to experimental analysis of fluid dynamics such as smoke, steam or dye injection into a flow field. Texture-based approaches to visualizing data generated by particle tracing have the advantage to be able to make larger amounts of data conceivable at once compared to geometric methods.
2.3.3. Commonly Used Data Structures for Unstructured Grids

From a post-processing point of view, structured grids have many advantages regarding the complexities of the algorithms used to handle their data structures. Problems that are easily solvable in structured grids, e.g. finding a cell given a location in the spatial domain or finding neighboring elements given a specific cell, are computationally much harder to solve in unstructured grids. Since many post-processing algorithms need to solve these problems repeatedly, additional data structures have to be used that speed up the computation of these tasks.

2.3.3.1. Indexing and Vertex Sharing

In unstructured grids, topology and vertex locations have to be explicitly stored. Since most vertices belong to more than one grid element, the storage size of an USG can be vastly reduced if these vertex locations are stored once and are then referenced instead of having to be stored multiple times as can be seen in Figure 39. Unfortunately, this reduction in storage size does not automatically lead to a decrease in processing time. Even if an algorithm has to access fewer data items because of coordinate sharing, the memory access patterns of these indexed data structures are not beneficial for the streaming multiprocessors of today’s manycore hardware due to repeated indirect addressing.

2.3.3.2. Neighbor Lists

A mapping from grid cells to neighbors is necessary to vastly decrease the computational complexity of advection in particle tracing algorithms. As there is no implicit neighbor information in USGs, these information has to be received from the numerical solver, or if that is not available, pre-computed for each USG. Either way, these neighbor lists use up large amounts of memory and increase the number of data accesses needed during the execution of post-processing algorithms.
2.3. Flow Visualization

2.3.3. Space Partitioning Trees

Space partitioning trees can be used to accelerate the finding of a cell containing a certain coordinate in the mesh. Commonly, KD trees [22] are used to store spatial subdivisions of coordinates in K-dimensional space. Since CFD cells span a certain extend in the computational mesh, not all points inside a certain cell can be assumed to be located in the same leaf of a space partitioning of vertices based on axis-aligned splitting planes. To efficiently locate a CFD cell given a certain point in 3D space, the KD tree datastructure has to be extended to be able to store a bounding interval hierarchy.

Figure 8.: Space partitioning using Cell-Tree

This extension is called Cell-Tree [23], storing bounding boxes of elements at the leaves of the tree instead of vertices as can be seen in Figure 8. As bounding boxes of elements in an unstructured grid generally overlap, leaves in a Cell-Tree may contain more than a single grid element. This leads to additional storage and runtime overhead depending on the depth of the subdivision, i.e. the maximum number of elements that are allowed to be contained in the leafs.

2.3.3.4. Data Partitioning Trees

Instead of spatial subdivisions, post-processing algorithms such as isosurface computation can also benefit from data partitioning trees, e.g. Interval-Trees. Data partitioning trees partition computational meshes according to the data val-
ues at the vertices or cells, which means a separate partitioning of the mesh is needed for each parameter array.

![Comparison of isosurface extraction performance using Cell Tree, Interval Tree and traditional implementation](image.png)

Figure 9.: Comparison of isosurface extraction performance using Cell Tree, Interval Tree and traditional implementation

Analysis of the runtime performance of isosurface extraction in a real-world dataset shows vast improvements when extracting small surfaces as shown in Figure 9. When more elements are sliced by the surface, at some point the overhead from processing the tree datastructure outweighs the performance gained from reducing the number of elements that need to be processed.

### 2.4. Post-processing Resources

Workstations have been largely replaced by clusters for interactive post-processing and rendering of large-scale simulation results. Nowadays, simulation datasets are getting even too large to be moved to a remote cluster after simulation. This is the reason why integrated simulation and post-processing clusters that share the same application network and filesystems are being increasingly necessary. Post-processing clusters immensely benefit from the huge processing power and high memory bandwidth of manycore devices such as GPGPUs.
2.4. Post-processing Resources

2.4.1. Levels of Parallelism

In High Performance Computing (HPC), different levels of parallelism are distinguished concerning both software and hardware architectures.

- Instruction level parallelism: CPUs can issue several instructions in parallel to different execution units inside a single CPU.

- Loop- or data level parallelism: Software method of issuing several blocks of data to different compute cores or shared memory processors in parallel. On CPUs, this is often implemented using OpenMP.

- Function- or task level parallelism: Distributed compute processes on different compute nodes. Message passing between different partitions, typically implemented using MPI or PVM.

Interactive parallel applications often try to leverage parallelism on multiple levels to achieve scalable low-latency, high throughput computations.

2.4.2. GPGPU Hardware

The rendering pipeline was extended by programmable shaders to provide more realistic rendering than what was possible using the fixed-function pipeline. Programmable fragment shader units especially enabled the development of per-pixel lighting instead of per-vertex lighting. The GPU’s shader units developed into multi-purpose, high throughput compute cores that were also usable for general purpose computing.

The compute elements in current generation GPGPU are built from streaming multiprocessors\(^1\). These multiprocessors contain compute cores that work in a SIMD fashion. Aside from the local memory of the compute cores, multiprocessors contain shared memory that gives compute cores collaborative access to data. Compute cores in separate multiprocessors may exchange data by using the global memory area of the device. NVIDIA represents the GPGPU hardware architecture in their CUDA API (see Figure 10).

\(^1\)NVIDIA terminology
Chapter 2. State of the Art and Technological Background

Figure 10.: NVIDIA GPU thread hierarchy and memory layout [67]

A thread represents a single chain of execution in a compute core. Blocks combine multiple threads inside a streaming multiprocessor that share a single execution unit and are able to communicate via shared memory. Blocks are scheduled to free streaming multiprocessors independently and thus are only allowed to communicate by using global memory. A compute kernel is represented by a grid that is forming a 1D or 2D organization of thread blocks.

2.4.3. Performance Considerations in GPGPU Computing

2.4.3.1. Diverging Threads

Since compute cores in a streaming multiprocessor share a single execution unit, the SIMD programming model has to be followed. Diverging threads are implemented by executing both branches serially, while disabling the compute cores that do not participate in the particular branch. This serialization of threads considerably limits the performance of kernels containing diverging execution paths.


2.4. Memory Throughput

In GPGPU stream programming devices, high global memory throughput is enabled by coalesced memory access. In current generation NVIDIA GPUs, global memory loads and stores by threads of a warp\(^2\) can be coalesced into one memory transaction if certain access requirements are met [66] (see Figure 11).

In practical post-processing applications, the design of data structures such as vertices as structures of arrays (SoA) or arrays of structures (AoS) [83], can lead to vast differences in performance of algorithms performing large amounts of memory accesses (see Figure 12).

2.4.4. Parallel Rendering in Scientific Computing

Data obtained from post-processing of numerical simulation results typically consist of a large number of geometric primitives. These primitives, generally huge amounts of triangle data, can be rendered by GPUs using either the fixed function pipeline or programmable shaders. Because of memory limitations of single workstations or cluster nodes as well as the complexity of the

---

\(^2\)A group of threads inside a block being concurrently executed by a streaming multiprocessor
computations that need to be performed, efficient parallel post-processing of large scale simulation data is essential. To reduce latency of user interactions within the visualization environment, post-processing and rendering have to be tightly integrated and able to exploit locality of data. Parallel rendering of distributed graphics primitives offers the possibility to maintain interactive rendering speeds when visualizing these large-scale, partitioned simulation data.

2.4.4.1. Display Hardware

There are different rendering environments that make it possible for individuals or small teams of engineers to collaborate on exploring large-scale numerical simulation data. These environments have different requirements for the remote rendering system. In general, it should be possible to combine various rendering environments, even of different types, to allow sharing of a post-processing session by multiple, geographically separated users.

An immersive digital environment is an artificial, interactive, computer-created scene within which users can immerse themselves. Because of cost-efficiency, these installations are often a central resource in the company, university or research institution, in many cases featuring a high bandwidth and low latency network connection to the data center. Virtual Reality environments such as large tiled display walls [61] or CAVEs [15] need to make high-resolution, very low
latency, high bandwidth stereo image renderings available to the users. Since these installations also often feature tracking equipment for user interactions as well as viewpoint changing, the rendering needs to be updated much more often compared to desktop environments, often continuously.

### 2.4.4.2. Parallel Rendering Paradigms

Parallel rendering techniques can be categorized based on at which point in the rendering pipeline graphics primitives are distributed or assigned to the parallel rendering clients. Molnar et al. [59] identify three classes of parallel rendering algorithms:

- **Sort-first** algorithms sort graphics primitives during geometry processing. Renderer partitions are responsible for a specific rectangle in screen-space. Graphics primitives can be assigned to renderer partitions by evaluating the screen-projection of the graphics primitives’ bounding box.

- **Sort-middle** algorithms redistribute graphics primitives between geometry processing and rasterization. After geometry processing, primitives transformed to screen coordinates are distributed to the partition responsible for rasterization of the particular primitive.

- **Sort-last** algorithms assign arbitrary geometry to each partition as seen in Figure 13. Rendered pixels are then combined by one or more *compositors*. The composition is done by sorting pixels according to their distance from the screen projection plane by consulting the depth buffers of the rendered images.

Sort-first parallel rendering is a suitable strategy when either the bandwidth requirements for redistribution of geometric primitives is relatively small, or when rendering times are dominated by the limited fill-rate of the GPUs (e.g. for volume rendering).

The most simple sort-last composition solution is the serial approach which combines and merges all intermediate images on the destination rendering unit.
responsible for the final display. Several other parallelization schemes for software composition have been proposed. Most notably, these include *Direct Send* [82], *Binary Tree* [79], *Binary Swap* [53] and *Parallel Pipeline* [50], among which *Binary Swap* is the most commonly used algorithm [20].

![Figure 13: Sort-last rendering](image)

*Binary Swap* algorithms distribute the composition of parts of images to all nodes in the cluster. A naïve approach for parallel merging of the partial images is to do binary composition. By pairing up processors in order of composition, each disjoint pair produces a new subimage. Thus after the first stage, the task is reduced to composition \( n/2 \) subimages. Using half the number of original processors they are then paired up for the next level of composition. Continuing similarly, after \( \log_2 n \) stages, the final image is obtained. One problem with the above method is that during the composition process many processors become idle. At the top of the tree, only one processor is active.

To exploit the compute power of all nodes in the cluster, the key idea in *Binary Swap* is that, at each composition stage, the two processors involved in a composite operation split the image plane into two pieces and each processor takes responsibility for one of the two pieces. In the early phases of the *Binary Swap* algorithm, each processor is responsible for a large portion of the image area as can be seen in Figure 14. In later phases of the algorithm, the processors...
are responsible for a smaller and smaller portion of the image area. At the top of the tree, all processors have complete information for a small rectangle of the image. The final image can be constructed by sending the subimage tiles to the display node or a remote workstation. The bandwidth requirements for *Binary Swap* composition are similar to the *Direct Send* algorithm. Fortunately, the communication patterns are much more suitable for cluster networks that are not fully connected, since larger image tiles are distributed in early stages of the composition tree where directly neighbouring cluster nodes communicate. There are further developments of *Binary Swap* such as 2-3 Swap [97] that eliminate the need to use power-of-two number of nodes. 2-3 Swap along with enhancements such as improved scanline methods, Sorted Pipeline Image Composition [71], RLE encoding and the use of bounding boxes [84], make for a promising approach towards even very large numbers of rendering nodes in future high performance cluster systems.
2.4.4.3. Parallel Rendering Software Systems

In addition to WireGL [36] [37], its successor Chromium [38] [39] and the CGLX library [18] that were mainly developed to accelerate existing software by using distributed rendering, complex frameworks such as Equalizer [19] have been developed to simplify the development of parallel graphical applications.

Several post-processing environments feature their own parallel rendering infrastructure such as AVS/Express [51] or ParaView, which contains the IceT library [60].
Chapter 3.

Related Work

A huge gap exists between what we know is possible with today’s machines and what we have so far been able to finish.

(Donald Knuth)

In this chapter, the state of the art in parallel post-processing environments as well as parallel algorithms for the visualization of flow fields are reviewed. The review shows parallel methods for surface extraction, particle tracing and texture-based dense flow visualization with a focus on execution on parallel manycore devices such as programmable GPUs.

3.1. Parallel Post-processing Infrastructure

Modular dataflow architectures dominate today’s general purpose post-processing and visualization software environments. Traditional, mostly serial packages such as IRIS Explorer [92], Data Explorer (now OpenDX) and COVISE [94] aim at allowing the user to specify their post-processing workflow by providing configurable, combinable and thus reusable modules. For the interactive processing of large-scale datasets however, distributed data access and parallel processing mechanisms are needed.
Chapter 3. Related Work

Many of the existing parallel post-processing architectures are based on the Visualization ToolKit (VTK) [77]. VTK provides methods that allow for the implementation of parallel algorithms using message passing over MPI, shared memory and socket connections. VTK also allows for the definition of parallel dataflow pipelines.

3.1.1. ParaView

ParaView is a dataflow-oriented post-processing and visualization system based on VTK. ParaView uses VTK’s data-parallel mode, where identical pipelines are created in every process, and each process is assigned a different piece of the data set [49]. Parallel algorithms operate on VTK objects each containing parts of the distributed simulation data.

The ParaView software architecture is split into several parts. A client process providing a graphical user interface (GUI) and several server processes that are able to exploit remote parallel processing resources. The server processes are separated into a root process and optional slave processes running on attached cluster nodes. User interactions are performed by sending commands from the client to the root process, which are then broadcast to the slave processes. These commands are implemented as TCL scripts which are interpreted by the server processes. Commands are also used to build visualization pipelines called the programme graph. Modules in the programme graph receive partitioned data objects for collaborative parallel processing.

Modules can be classified into sources, filters and mappers [77]. Data storage and rendering of post-processed simulation data is separated into multiple components. This allows for the usage of ParaView in heterogeneous cluster environments where not all compute nodes also contain GPUs [32]. Since post-processing operations most often result in a vast reduction of datasize, the number of rendering servers typically needs to be much smaller than those used for post-processing. ParaView supports server side rendering as well as the transfer of geometry from the servers to the client containing the GUI.
3.1. Parallel Post-processing Infrastructure

3.1.1. Parallelism at the Task Level

ParaView achieves parallelism at the task level by executing identical pipelines in parallel, and merging the distributed results in the renderer using `vtkParallelRenderManager`.

3.1.1.2. Parallelism at the Algorithm Level

Many VTK filters support loop-level parallelism using OpenMP. VTK itself is not thread-safe, so data objects have to be created in the main thread. Changing data using multiple threads has to be protected by mutual exclusion, adding communication overhead to parallel computations at the algorithm level.

3.1.2. VisIt

VisIt is a parallel data-flow oriented visualization system that makes heavy use of VTK functionality in its components, but is independent of VTK’s data flow network.

The VisIt data flow network consists of two different types: `data objects` and `components` [12]. Components can be categorized into sources, e.g. file readers, filters that transform input data into output data, and sinks such as render components that consume data objects.

The pipeline, a collection of components, is executed in a demand-driven fashion that starts with a pull operation at the end of the dataflow graph. A sink generates an update request that propagates through the pipeline to the source. The data generated by the source is then processed by filter components and finally consumed by the sink. Parallelism is achieved by executing identical flow networks on different cluster nodes, distinguished only by the different chunks of data that are generated or processed during the different stages in the pipeline.
Chapter 3. Related Work

One main distinctive feature of VisIt is its notion of contracts that enable components to perform optimizations dependent on the properties of the pipeline components [13]. These contracts allow components to specify how they impact a pipeline, serving as an interface for communication between components detached from the processing of data objects. A contract is created at a sink and travels upwards through the pipeline along with update requests, allowing components to change or augment a given contract along the way. In a contract, components can then react to requirements requested by consumers of their output objects, as well as demand that their input objects will fulfill certain characteristics. Contracts can also be used to limit the amount of data that has to be read from disk, e.g. when spatial extends are communicated by a filter component to the reader component.

VisIt supports different models for execution of parallel processing. Streaming can be used for out of core processing where only one domain is processed per execution of the pipeline, given post-processing algorithms that do not have to employ collective communication. In grouping mode, the pipeline is executed once with each component processing all domains before proceeding to the next one. This means that using streaming, adaptive load-balancing can be performed since domains can be dynamically assigned to processor cores or post-processing nodes.

VisIt supports both rendering at the client for small numbers of polygons, as well as remote parallel rendering based on the sort-last algorithm (see Section 2.4.4).

3.1.3. Viracocha

Viracocha is a parallel extension to the visualization library ViSTA FlowLib [76]. Its main focus lies on the interactive post-processing of large-scale CFD datasets in virtual environments [25]. Viracocha employs a master/worker model to distribute computations to parallel post-processing nodes. Interactions are sent from ViSTA to the Viracocha scheduler which uses worker processes to
3.2. Parallel Post-processing Algorithms

do parallel post-processing. After completion, the master worker collects partial results and sends the complete result back to ViSTA FlowLib. Partial results generated by worker processes can also be streamed directly to the rendering component.

Throughout the system, an optional streaming multi-resolution approach can be followed that allows for the extraction of base data from the lowest resolution dataset available [74]. The approximate data is then being refined by progressively streaming results generated from post-processing on higher resolution grids. Due to the generated overhead by generating multiple results on different scales, the total computation time will be increased when streaming is enabled.

The parallel processing is based on an MPI enabled processing backend, that is coupled to the ViSTA rendering frontend by using a single TCP connection. The hybrid parallel processing backend relies on distributed processing of decomposed multi-block datasets [95]. The usage of a single TCP connection between the Viracocha and ViSTA FlowLib components lead to interruptions where the visualization system may be stalled by newly generated data, negatively affecting real-time rendering [24].

Viracocha relies upon a central data manager at the scheduler node that coordinates data proxies at the compute nodes. The data management supports hierarchical cache levels employing different caching strategies such as LRU, LFU and FBU. These data caches can be filled using prefetching [26].

3.2. Parallel Post-processing Algorithms

3.2.1. Surface Extraction

Surface extraction methods based on the marching cubes family of algorithms have been optimized for parallel computers very early after the publication of the original algorithm. Paul Mackerras’ implementation for the experimental
distributed memory MIMD computer Fujitsu AP1000 uses domain decomposition of cartesian grids to distribute the work between the parallel processors [57]. To correctly compute normal vectors at the borders between partitions while avoiding the communication overhead between processors, *halo elements* of size 1 in each dimension are employed. To achieve primitive load-balancing, an interleaved assignment of blocks containing a low number of cells to the processors is performed.

Pascucci [68] and Reck et al. [69] use the vertex shading unit of programmable GPUs to extract isosurfaces from unstructured tetrahedral meshes. To create a quadrilateral surface patch from each tetrahedron, they send four numbered vertices through the vertex pipeline, with the actual vertex positions and function values stored as vertex attributes. The tables used for the *marching tetrahedra* algorithm that is computed by the vertex shader are supplied in constant GPU registers. The output generated by the vertex shader is always a quadrilateral. Degenerate quadrilaterals are used when the actual surface inside an individual tetrahedron is actually a triangle or when the isosurface does not intersect the tetrahedron at all. Degenerate primitives with zero area are then ignored by the rasterizer.

Buatois et al. [9] extend on this work by storing data in GPU texture memory instead of GPU registers to be able to defeat the memory limitations of the former approach. They also use indexing to support the sharing of vertices between multiple tetrahedra and thus avoiding data redundancy. Although, this approach achieves a relatively low performance because of expensive texture lookups in the vertex shader.

Klein et al. [45] follow a different approach to also make use of the fragment shaders and to be able to read back generated surface patches from the GPU. The fragment shader is used to render a single quadrilateral per tetrahedron into a floating point render target using ATI’s *SuperBuffer* extension. After applying *marching tetrahedra* in the fragment shader, the *SuperBuffer* is then bound as a vertex buffer array to render the resulting isosurface.

Martin et al. present a load-balancing method for isosurface extraction on multi-GPU clusters [58]. They suggest applying cost heuristics in a pre-processing
3.2. Parallel Post-processing Algorithms

step to pairs of isovalue and partition in a multi-block grid. Triangle count for
a range of isosurfaces in each block in the grid. During runtime, blocks are
scheduled to GPUs in the system according to the estimated runtime for load
balancing. They show that triangle count is a legitimate heuristic for expected
runtime performance of isosurface extraction using marching cubes.

Gerndt et al. present view-dependent sorting for progressive streaming during
isosurface extraction [25]. They use view-dependent BSP trees to sort blocks in
partitioned grids from front to back with respect to the viewing position.

NVIDIA provide a CUDA-based marching cubes implementation for rectilinear
grids with their CUDA SDK samples [65]. Their implementation uses textures
to store the lookup tables. CUDA kernels are used to classify voxels based on
the number of vertices that will be generated per voxel. Stream compaction
is used to skip voxels that are not cut by the isosurface. Triangles are cre-
ated using marching cubes and per voxel normals are interpolated from the field
function.

3.2.2. Particle Tracing

The advancement of GPU technology greatly enhanced the usability of ren-
dering hardware as general computing devices. Newly developed GPU fea-
tures made it possible to implement particle tracing algorithms directly on pro-
grammable GPUs:

- Floating point textures and floating point render targets can be employed
to generate floating point output from fragment shader programs.

- Texture maps are made available in the vertex units allowing for the ran-
dom access of GPU memory in vertex shader programs.

- Buffers can be interpreted both as texture maps and as vertex arrays. This
enables the passing of render targets as vertex inputs to subsequent ren-
dering passes.
These features enable Kruger et al. to achieve interactive particle tracing on uniform grids [46] by performing particle advection, geometry generation and rendering completely on the GPU without any readback to CPU accessible memory. They use floating point textures to store and access the initial seed points of particles in the RGB channels, leaving the alpha channel to designate individual particle lifetimes. A floating point texture render target is employed to store advected particle positions in the fragment shader. In the second pass, vertex texture fetches are executed for displacing particle geometry according to the positions stored in the render target of the previous pass.

Schirski et al. adapt this technique for particle tracing in unstructured tetrahedral grids [74] [75]. In their tracer, the locating of cells is implemented in two phases. A broad phase where a KD tree [5] is used on the CPU to locate a vertex close to the particle position, followed by a tetrahedral walk [44] on the GPU using barycentric coordinates. As in Kruger’s approach, particle positions are stored in the RGB channels of floating point textures, the alpha channel is used to designate the cell index in the unstructured tetrahedral mesh.

Yu et al. provide methods for scalable pathline construction on very large time-varying cartesian grids [96]. In a pre-processing step, they construct hierarchical meshes by merging neighboring cells with similar vector data into larger clusters. The larger clusters on higher levels can then be assigned to the individual compute nodes in a parallel cluster system. During particle tracing, this minimizes communication between partitions since clusters are built along the principal flow direction and streamlines can thus be expected to cross partition boundaries very infrequently.

3.2.3. Texture-based Dense Flow Visualization

In the visualization of CFD simulation data, lines and animated particles are used to provide insight into the local flow field. Arrow plots have been used as the standard method to visualize the global flow directions in dense flow fields, although it is hard for the user to reconstruct the flow from discrete samples. To provide a global, contiguous visualization of dense flow fields, texture-based
3.2. Parallel Post-processing Algorithms

methods such as Digital Differential Analyzer (DDA), Spot Noise and Line Integral Convolution (LIC) have been developed. The basic idea of these flow visualization techniques is to blur textures containing noise along the local vectors in a flow fields. As opposed to DDA convolution where complete streamlines are approximated by the tangent of the vector field, LIC [10] computes locally exact streamline advection.

FastLIC [81] improves on the original algorithm by making two observations: First, streamlines cover multiple texels, and thus large parts of streamlines are recomputed for multiple texels. Second, for a constant filter kernel similar convolution integrals occur for texels covered by the same streamline.

Parallel LIC [99] is a parallelization of FastLIC that exploits both spatial and temporal coherence. The parallelization is done in image space by subdivision of the image into several subdomains assigned to different processors. To avoid communication between partitions, the vector field data is replicated in neighboring domains.

Teitzel et al. perform LIC on unstructured triangulated 2D surfaces [85]. To avoid the mapping of complex surfaces to flat texture space, one texture per triangle of the surface mesh with a fixed number of texels per triangle are employed. After projecting the vector field to the triangular mesh, streamline computation and traditional LIC for the projected vector field is performed. Several methods for the generation of the input texture are evaluated: First, using the same white noise for each texture. Second, texture interpolation by vertex using one intensity value per vertex. Third, triangle subdivision with a fixed number of subdivision steps and a single intensity value per triangle.

Unsteady Flow LIC (UFLIC) [93] and Accelerated Unsteady Flow LIC (AUFLIC) [52] extend LIC for the visualization of unsteady flow fields. Since the traditional streamline approach supports no coherence between animation frames, they use scattering of pixel values following pathlines and combination of image values at pixels from pathlines matching the timestep in the animation. AUFLIC increases performance of UFLIC to near interactive framerates by sparse seeding of pathlines and the reuse of known pathlines. The spatial reuse is performed by correcting already computed pathlines that travel through
Chapter 3. Related Work

the pixel containing the seed point of a new pathline. Temporal flexibility allows for pathline seeds to not be released at exactly integer timesteps, but some fractional time shortly after the scattering process.

Bachthaler et al. accelerate LIC by combining hybrid parallelization strategies on GPU cluster computers [4]. After using Sort-first for image space decomposition and Sort-last for object space partitioning of the vector field, LIC is performed on projected flow fields using GPU shader programs in image space. The flow field is replicated using ghost cells at partition border for efficient particle tracing. Traditional z-sorting and blending of subimages is performed during image composition in the Sort-last phase.

Image-based Flow Visualization (IBFV) [88] and Image-based Flow Visualization for Surfaces (IBFVS) [89] is an alternative to traditional LIC-based approaches for unsteady flow. Each image is the result of warping the previous image along the flow direction, blended with background images containing white noise. Van Wijk uses CPU-based a projection of the velocity vectors at mesh vertices to image space. The texture is then advected over the mesh according to the velocity vectors stored at the projected mesh vertices. The distorted texture coordinates computed by the mesh advection are then mapped to the original 3D surface mesh vertices.

Image Space Advection (ISA) [48] distorts a regular, rectilinear mesh defined in 2D image space. The vectors in a flow field at the vertices are encoded in the RGB channels of a texture and rendered using Gouraud shading to interpolate the vectors resulting in a velocity image. The velocity vectors projected onto the image plane are then used for image-space advection of the local flow field.
Chapter 4.

Interactive Post-Processing in Parallel Manycore Environments

The secret of getting ahead is getting started. The secret of getting started is breaking your complex, overwhelming tasks into small manageable tasks, and then starting on the first one.  

(Mark Twain)

4.1. Utilizing Visualization Clusters

Workstations that have been used traditionally to post-process simulation datasets are no longer able to process the huge amount of data produced by large-scale simulations. The limiting factors are the resources available in a single workstation, e.g. the amount of memory that can be installed in a single system or the number of compute cores available. Another factor is the network bandwidth between the compute cluster and workstations of post-processing engineers. The size of raw simulation data of large-scale simulations is prohibitively large to be copied off the compute cluster to an outgoing network.

This makes it necessary for post-processing and visualization of simulation data to be performed either during runtime of the simulation on the cluster nodes themselves (in-situ visualization [56]), or on dedicated visualization nodes in
the cluster that can be utilized for interactive tasks. These visualization nodes often share a common network and parallel filesystem with the compute nodes, which eliminates the need to move data off the cluster. Although the visualization cluster nodes typically use the same general processing architecture as the compute nodes, they additionally offer manycore accelerators such as programmable GPUs to support post-processing and rendering.

4.2. Interactive Post-Processing in Hybrid Manycore Systems

To be able to execute interactive post-processing algorithms on visualization clusters, the post-processing software environment has to able to make optimal use of the available resources.

- Supporting distributed memory environments. Subsets of data have to be able to be distributed to nodes in the visualization cluster. A given decomposition of data generated during pre-processing must be allowed to be used by the post-processing environment. Data might be needed to be replicated from one cluster node to another or even between multiple interconnected clusters. Shared access to data must be possible for concurrently executing post-processing algorithms.

- Executing parallel algorithms on distributed simulation data. Algorithms should be allowed to process data both on individual partitions independently, as well as collaboratively across multiple nodes in the cluster. Post-processing algorithms need to be parametrizable for the user of the system to be able to influence various aspects of the workflow interactively, such as iso-values or normal vectors of surface extraction algorithms.

- Accessing manycore accelerators such as GPGPUs to achieve interactivity. Programmable GPUs feature high memory bandwidth and high
processing power which makes them an ideal target for interactive post-
processing algorithms. Using GPUs for post-processing also allows for
immediate rendering of processed data, avoiding the latency introduced
by copying data between host and accelerator device. Parallel render-
ing techniques (2.4.4) and composition methods have to be supported to
minimize network bandwidth requirements between the post-processing
infrastructure and user workstations.

- Making the visualization cluster resources as well as individual post-
processing workflows available to multiple remote workstations simult-
aneously allowing for collaborative distributed working.

To achieve interactivity, i.e. a delay of less than 100 ms [8] from user interac-
tion to visual feedback, close attention has to be payed to software architecture
design decisions that introduce latency.

4.2.1. Simulation Datasets

Datasets that are generated by large-scale CFD simulations on unstructured
grids consist of multiple different kinds of data. First, one or more partitioned
unstructured meshes containing 1D, 2D and 3D elements of various types, such
as points, lines, polygons, tetrahedra, prisms or hexahedra. Second, one or more
arrays of data of potentially different dimensionality that map to faces, cells or
vertices of the unstructured mesh(es). These data include computed pressure,
temperature and velocity values at the distinct positions in the mesh, as well as
data initially set as boundary conditions for the numerical solver. Additionally,
there might be geometry specifying boundaries or components that are not part
of the actual simulation run but are helpful or necessary for efficient analysis of
the resulting dataset.

4.2.1.1. Domain Decomposition

In large-scale simulations on distributed memory clusters, the simulation do-
main is partitioned spatially into chunks of data, which will be referred to as
blocks, that are processed by different compute nodes. Since the processing of data at the boundary of a block requires data from neighboring cells that might be located in a different block, the contents of these cells have to be communicated between the compute nodes participating in the simulation. To reduce communication overhead from exchanging individual data elements between partitions during access, these data elements are replicated inside the block, allowing for efficient transfer of all out-of-block neighbors at once. These out-of-block elements are called halo elements or ghost cells. The size of data per block in today’s HPC CFD simulations typically varies from several thousand elements to several million elements per core.

4.2.2. Distributed Data Objects

Without loss of generality, both the memory as well as the computational complexity of the CFD simulation solver will be higher than that of the post-processing algorithms. This allows for an existing partitioning of the data to be reused for the execution of parallel post-processing algorithms in a distributed memory cluster, even combining multiple smaller blocks in a single partition of the post-processing environment, avoiding computationally expensive repartitioning of the data.

To be able to execute collaborative parallel algorithms on distributed data objects, these objects need to contain information about which block and timestep they belong to, and which blocks are their neighbors. To limit the amount of memory needed to represent unstructured grids, data structures can be used to separate geometry from topology (see Appendix 39).

4.2.3. Dataflow Architecture

Post-processing environments allow for interactive analysis of simulation data by making modular algorithms applicable to operate on these datasets. Popular systems such as IRIS Explorer [21], IBM data explorer [55] and its successor
OpenDX [1], COVISE [94] and ParaView [77] employ data-centric views of post-processing workflows.

A dataflow graph is a directed graph in which nodes represent post-processing modules and vertices correspond to the data dependencies. Modules in the dataflow graph implement functions of one or more arguments that are represented by data connected as input streams to a given node. Post-processing modules typically produce data from their input streams that is then made available as one or more output streams. A given module may execute when data is available at all its required input ports. This strategy introduces sequencing between modules that depend on each other and allows for independent modules to execute in parallel.

Algorithms that can be instantiated as modules operate on a portion of data and transform these data to a different representation. The transformation might also be parametrized by the user to allow for an interactive analysis of the underlying data. A typical classification of post-processing algorithms follows the visualization pipeline [2.1.1]:

1. **Data acquisition**: Modules that allow for input of simulation data from different sources. These sources include databases, files and simulations that are coupled to the post-processing environment during runtime for in-situ visualization [56] of simulation data.

2. **Data analysis**: Interpolation of data to e.g. different types of grids for faster post-processing and visualization of a dataset.

3. **Filtering**: Selection of a subset of data that is of particular interest to the scientist (focus data), this typically includes cropping of data to drastically reduce the amount of data to be visualized and analysed.

4. **Mapping**: Mapping of focus data to geometric primitives such as lines and triangles. Several different mapping techniques exist according to the dimensionality of the data values, e.g. color coding for scalar data or rendering of glyphs to represent vector data.
Chapter 4. Interactive Post-Processing in Parallel Manycore Environments

5. Rendering: Rendering of post-processed data in the form of geometric primitives suitable for a given environment, e.g. powerwalls, CAVEs or desktop workstations. A renderer typically includes the possibility of user-interaction with the post-processing system, adapted to the possibilities of interaction in the given environment.

Dataflow architectures are well established for post-processing of simulation data because they offer a great deal of flexibility through the ability to arrange complex workflows by composition of smaller, relatively simple algorithms [70]. Their modularized nature also allows for ad-hoc extension by adding additional algorithms needed for advanced, sometimes domain-specific, tasks.

4.2.4. Hybrid Parallel Post-Processing Algorithms

Hybrid parallelization describes exploiting different levels of parallelism or parallelism on several different kinds of hardware architectures at the same time as will be described later on in section 4.3.1. Typically, this refers to algorithms working in a distributed shared memory environments such as today’s clusters, which consist of a large amount of shared memory multiprocessors. In this work, the term hybrid parallelism in the scope of post-processing clusters is extended to include exploiting manycore accelerator cards such as GPGPUs.

Parallel post-processing algorithms can be categorized according to the amount of communication that is necessary between the parallel partitions during execution.

- **Pleasantly parallel** algorithms require no communication with other partitions during runtime. This group of algorithms include simple surface extraction where the extracted surface, in particular the normal of the surface at each vertex in the USG, is known to every partition as will be elaborated in section 6.

- **Semi-independent** algorithms require the exchange of data with neighboring cells. This group contains e.g. more sophisticated surface extraction
4.3. A Dataflow oriented, Parallel Post-Processing Environment

Techniques such as isosurface extraction when accurate, per-vertex normals are needed in the post-processed result as will be seen in section 6. Often, communication can be avoided or performed more efficiently by using halo cells.

- Collaborative algorithms require dynamic data exchange. This may be the case when communication between partitions depends on the data values which might not be possible to predict before execution of the algorithm. This group includes particle tracing methods that will be discussed in section 5.2.5.

This categorization also extends to data-parallel post-processing algorithms.

4.3. Software Architecture of vistle\textsuperscript{1}: A Dataflow oriented, Parallel Post-Processing Environment

\textbullet\textsuperscript{1}Visualization Testing Laboratory for Exascale Computing

\textbf{Figure 15.}: Architecture of a parallel post-processing environment.

\textit{vistle} is a dataflow-oriented parallel post-processing architecture that has been built during the creation of this work. The software architecture is based upon collaborative hybrid parallel processes (see Figure 15). There are multiple possibilities to map tasks in a dataflow system to operating system primitives. The...
Chapter 4. Interactive Post-Processing in Parallel Manycore Environments

The choice which task abstraction to use influences the techniques employed to implement other integral components, such as communication and data management. The rationale for the choice of task abstraction is also influenced by considerations about performance, ease of programmability and fault isolation. In this section, decisions in the design of the vistle software architecture and its components will be presented.

4.3.1. Employing Different Levels of Parallelism

4.3.1.1. Task Parallelism

In this work, a post-processing task is defined as an independent algorithm that takes a stream of input data and executes specific instructions to produce one or more streams of output data. Task parallelism can then be defined as the concurrent execution of more than one post-processing task on different processor cores. Therefore, the ability to execute a given post-processing task depends only on the availability of its input streams. A runtime system for the execution of post-processing tasks can be modeled to contain a directed dataflow graph where the vertices are represented by tasks, and the edges are represented by data streams (see Figure 16). Due to the explicit modeling of the interdependence of tasks in the dataflow graph, the runtime system can determine which tasks may be executed according to the state of their input streams.

A post-processing task itself can be implemented to contain various subtasks that can be parallelized explicitly by the programmer without support of the underlying dataflow runtime system.

4.3.1.2. Data Parallelism

Data parallelism is the parallelization of an algorithm by distributing data across multiple processor cores and applying the same function to single data elements or subsets of data elements in parallel. This allows the algorithm to scale with
the number of data elements and processors [2]. Data parallelism can be exploited on various different scales:

- **Loop level parallelism**: On shared memory multiprocessors, a single set of operations can be executed on multiple data elements simultaneously (SIMD). Accelerators such as GPGPU devices expose methods both similar to SIMD and SMT, where the processing of each data element can be interpreted as being executed in a separate thread (SIMT).

- **Domain decomposition** refers to decomposing a data structure and solving a specific problem for the smaller subdomains. In parallel computing, subdomains can be processed simultaneously on different compute nodes. The performance of the parallel algorithm is often limited by the amount of communication that is necessary when elements in different domains cannot be processed individually. For post-processing of simulation data, the existing decomposition used by the numerical solver usually can be reused.

### 4.3.2. Parallel Runtime Environment

A runtime environment used to execute parallel post-processing algorithms must allow for both independent and collaborative processing of subdomains of data. For collaborative parallel execution, a method for communication between parallel partitions on distributed memory systems has to be supported.

*Message Passing Interface (MPI)*: MPI is a standard message passing interface for distributed memory concurrent computers [91]. MPI provides an execution environment for parallel processes as well as methods for communication between these processes. With MPI 2, a dynamic process model has been introduced. This allows for the spawning of new processes by a running application and connecting to existing processes to support client / server applications [28]. Several modes of communication between partitions of a parallel program are supported. Point-to-point communications can be executed both blocking and non-blocking. As of MPI 2, collective operations can only be performed by
blocking communication, severely limiting their use, since non-blocking communication can be used to implement overlapping of communication and computation. MPI supports the definition of process groups and intercommunication between these groups.

Enhancements to support manycore architectures are still an active research topic:

- Current implementations are not able to scale to very high numbers of cores. This is hindered e.g. by communication buffers limiting the number of connections that are possible between partitions.

- Employing very high numbers of cores reduces the time to failure due to hardware defects of the whole system. Fault tolerance in MPI will have to be improved upon the simple checkpointing / restore that is no longer applicable in large-scale applications [40].
4.3. A Dataflow oriented, Parallel Post-Processing Environment

MPI provided easy integration with shared memory parallel programming methods such as OpenMP [16].

*Parallel Virtual Machine (PVM):* Focused on portability over performance, dynamic processes in heterogeneous environments [29]. Low-level message passing. Some support for fault tolerance and recovery.

*Partitioned global address space (PGAS):* PGAS languages allow for partitioning of data while controlling locality. Current implementations, e.g. Unified Parallel C (UPC) or Co-Array Fortran are based on active messages [90] implemented on top of GASNet [7]. Implementations of GASNet for high performance networks either already exist or are currently being developed. A huge benefit of PGAS languages is the ease of programming due to globally accessible data structures and implicit communication. However, PGAS languages still have not been widely adopted in the HPC community. According to Shan et al. this is mainly due to “lack of direct hardware support, a mature developer environment and lack of convincing performance results that they are superior to MPI” [78]. Another factor is the poor interoperability between PGAS languages and existing high performance numerical libraries written using OpenMP and MPI [42].

The major reasons why MPI has been chosen as parallel runtime and communication framework for *vistle* are its wide deployment on clusters, the support for all major communication networks, high performance implementations by cluster vendors, easy integration with OpenMP, and the availability of numerical libraries and of implementations of various different post-processing algorithms.
4.3.3. Components of the Post-Processing Environment

4.3.3.1. Controller

The controller handles dataflow between different post-processing modules, and is responsible for distributed memory communication between the various components in the system. To be able to control the workflow by executing runnable module partitions, it is responsible for managing the workflow graph containing connections between the modules currently running in the system. The execution of modules where objects are available on input ports can be performed according to different modes that can be specified by the post-processing module:

- **Single timestep, single block.** A partition of the module may be executed at any time a new data object partition is available. The module partition is not allowed to do blocking communication with other partitions during the processing of the input object. This mode is applicable when all computations performed by a module partition can be executed independent of the computation of the other partitions. This is the case e.g. for the extraction of surfaces from unstructured grids when generating per-face normals.

- **Single timestep, all blocks.** Execution of a module partition may not start until all blocks in a given timestep are currently available as input objects at the different partitions of this module. This mode is applicable for modules that have to perform global communication between the different partitions participating in the post-processing of a dataset. Particle tracing modules are an example that have to track particles between the different partitions of the computational grid.

- **All timesteps, all blocks.** Processing of input data by executing the module may only start if all timesteps and all blocks of the input dataset are available. The computation of pathlines are an example for a module specifying this execution mode.
4.3. A Dataflow oriented, Parallel Post-Processing Environment

The type of execution needs to be registered for any given module at startup time.

![Sequence diagram of the creation of a shared memory object during execution of a module partition](image)

**Figure 17.:** Sequence diagram of the creation of a shared memory object during execution of a module partition

### 4.3.3.2. Data Management

Data in parallel applications has to be made accessible to the partitions participating in the data processing. For shared memory systems such as nodes in a compute cluster containing multiple processors and compute cores, many different APIs exist to provide shared access to data. In *vistle*, a system to create, access and identify objects in shared memory (Shm) based on the Boost [17] interprocess library has been developed, providing an abstraction of SysV and POSIX shared memory and memory mapped files.

Supported object types in shared memory are unstructured grids, points, lines, triangles, polygons, textures, and arrays of floating point or integer values of different dimensionality. In addition, user defined types can be stored in shared memory by the developer of post-processing modules for easy extensibility.

To avoid global communication when creating or accessing shared memory objects, every cluster node contains an independent partition of the data manager. Access to data by partitions of post-processing modules is always performed by contacting the local data manager (see Figure 17). Data objects are then replicated from other data manager partitions when necessary.
4.3.3.3. GPU Data

To be able to share data on limited GPU memory resources by different post-processing algorithms, vistle contains a memory manager for GPU objects. This memory manager is used to replicate objects from host memory to GPU memory, automatically transforming datastructures to a memory layout that is optimal for parallel access by post-processing algorithms executed on streaming multiprocessors.

Due to the implementation of data management in today’s GPU drivers, CPU objects such as textures, VBOs or Renderobjects, can only be accessed without difficulty from the operating system process in which they were allocated. Therefore, GPU data management and post-processing algorithms in vistle are implemented and executed inside the local partition of the renderer.

4.3.3.4. Parallel Modules

Modules implement operations on data, post-processing algorithms and rendering. These modules can employ different levels of parallelism making use of shared memory multiprocessing (OpenMP) and distributed memory using message passing (MPI). Module contain input ports and output ports that can be connected to ports of other modules in the dataflow graph. When a module generates a data object at one of its output ports, the controller will send a message to the connected modules depending on the module’s execution mode (see Section 4.3.3.1).

According to the execution mode, different forms of communication may be used inside a module:

- **Single timestep, single block.** The module partition is not allowed to do blocking communication with other partitions during the processing of the input object, since computation on other partitions may or may not be executed at the same time.
4.3. A Dataflow oriented, Parallel Post-Processing Environment

- **Single timestep, all blocks and All timesteps, all blocks.** Global communication may be performed, since the other partitions are guaranteed to be executed in the same sequence.

**I/O** I/O modules function as sources or sinks in the dataflow graph, performing reading or writing of simulation datasets from or to filesystems. To be efficient in a distributed environment, I/O modules have to support simultaneous, communication-free parallel input and output of blocks of data on different compute nodes.

The *vistle* file format (see Figure 18) enables the I/O of parallel regions of data by supplying meta-data linked to the objects in a file. The object-oriented metadata in the file header serves both as a specification of the layout of the file, as well as a description of the stored objects. Utilizing information about the dataflow network obtained from the controller, an I/O module is enabled to read distinct partial objects from the specified file in different partitions of the module. This decomposition of objects then provides a distribution of work amongst the various post-processing modules in the dataflow graph.

**Post-Processing** Post-processing modules act as filters for data objects, transforming their input objects into output objects according to a given post-processing algorithm. The available post-processing modules are implementations of the cut- and isosurface algorithms presented in Section 6, as well as the
particle tracing methods introduced in Section 5. These implementations both
test independent execution of different domains in the dataflow graph for sur-
face extraction, and collaborative processing for the tracing of particles through
different domains in the CFD grid.

**Parallel Rendering**  Renderer modules provide a user interface for interac-
tion with data or post-processed results in a given graphical environment. The
parallel rendering module currently implemented in *vistle* supports sort-last ren-
dering for desktop computers on remote clusters. Along with transformation
of the viewer position, control over the visualized timesteps is provided using
mouse input. The performance of the renderer based on the parallel rendering li-
brary IceT [60] has been evaluated on the CRAY XE6 installation at HLRS (see
Figure 19). Normally, in sort-last based algorithms, the performance should be
similar inside a section ranging from \(2^n + 1\) to \(2^{n+1}\) nodes. The observable vari-
atations in the measurements stem from the node allocation that is not guaranteed
to be consecutive in the testing phase of the HERMIT installation.

![Figure 19.: Performance of sort-last rendering on 1024 nodes of CRAY XE6 HERMIT](image-url)
Chapter 5.

Particle Tracing

When you put together the science of the motions of water, remember to include under each proposition its application and use, in order that this science may not be useless. 

(Leonardo da Vinci)

Particle tracing is an established visualization method for the analysis of CFD data. By releasing particles into a computational domain and computing the paths of the particles as they are advected by the flow field, particle tracing algorithms are able to reproduce experimental results created in wind tunnels or turbomachinery testbeds.

5.1. Introduction

Time-varying flow fields are defined as vector-valued functions defined on a given domain:

$$v(x, t) : D \times I \rightarrow \mathbb{R}^n$$

(5.1)

where \( x \in D \subseteq \mathbb{R}^n \) defines spatial positions in an \( n \)-dimensional flow domain and \( t \in I \subseteq \mathbb{R} \) represents time in transient flow fields. Particle paths can be computed by the integration of the ordinary differential equation (ODE)

$$\frac{dx(t)}{dt} = v(x(t), t)$$

(5.2)
where \( x(t) \) denotes the position of the particle at time \( t \), and \( v(x, t) \) denotes the velocity-vector of the flow field at position \( x \) at time \( t \). The starting position \( x_0 \) of the particle provides the initial condition

\[
x(t_0) = x_0 \quad \text{(5.3)}
\]

Reformulation of equation 5.2 leads to its integral form

\[
x(t + \Delta t) = x(t) + \int_t^{t+\Delta t} v(x(t), t) \, dt \quad \text{(5.4)}
\]

While equation 5.4 cannot be analytically solved in general, a solution can be approximated using numerical integration. Although in CFD simulation data vector fields are given in discrete form over a computational mesh, numerical integrators treat the flow field \( v \) as a continuous function. As flow data is given with respect to individual grid cells or vertices, there is a need for interpolation to evaluate the flow field at arbitrary positions inside the computational domain.

### 5.2. Parallel Particle Tracing on GPGPUs

Algorithms for the tracing of particles by numerical integration in discrete meshes perform multiple steps. First, the cell that contains the particle at its initial position has to be determined. The velocity of the flow field at this position has to be determined by interpolating the velocity field at the vertices of the cell. Then, an integration step calculates the next position of the particle in the flow field according to the, either constant or adaptive, step size. If the particle has left the mesh element, the cell containing the new particle position has to be determined. The algorithm continues until either the particle leaves the computational mesh, or another break condition, such as a total number of integration steps, is reached.
5.2. Parallel Particle Tracing on GPGPUs

5.2.1. Initial Cell Location

The finding of cells for the location of particle positions can be accelerated by utilizing binary space partitioning data structures such as KD trees [22] (see also section 2.3.3.3). There are different ways KD trees can be used to help in initial cell location. First, vertices potentially close to a given coordinate can be found by traversing the vertex partitioning tree and then continuing the search by visiting the cell the vertex is in and its neighbors until the correct cell is found. Second, a KD tree can also be used to quickly constrain the number of cells a specific coordinate may be located in. For this purpose, leaves in the KD tree store axis-aligned bounding boxes (AABBs) of the elements in the computational mesh. The element(s) containing a specific coordinate can then be found by traversing the KD tree until a leaf is reached. Considering mesh elements cover an extent in space and are generally neither axis-aligned nor aligned to other mesh elements, splitting planes of the KD tree may slice elements, leading to the requirement for elements to be located in more than one branch of the tree. These observations motivate an extension of the KD tree datastructure called Cell-Tree [23]. Since AABBs of unstructured mesh elements can generally overlap, the leaves in the Cell-Tree are not limited to containing only a single mesh element. To limit the amount of memory a Cell-Tree uses, it is beneficial for the leaves in the tree to contain a certain maximum number of elements, and then use exact point location methods in these cells to find the particular coordinate. KD trees can be constructed on graphics hardware in realtime [98].

5.2.2. Tetrahedral Decomposition

There are multiple possibilities for interpolation in unstructured mesh elements. The first method is the transformation of irregular elements into regular elements to carry out the interpolation in transformed space. Although, the transformation of irregular element using Jacobian matrices is only an approximation and can lead to discontinuities in the flow field [43]. The second method is the decomposition of elements such as hexahedra, prisms and pyramids into multiple
Chapter 5. Particle Tracing

(a) Incongruent decomposition  
(b) Continuous decomposition

Figure 20.: Decomposition of neighboring hexahedral cells

tetrahedra and computing natural coordinates for interpolation of the flow field inside these tetrahedra [44].

It has to be noted that in general, a given decomposition of complex elements into tetrahedra is not unique. To be able to create consistent results, the decomposition of these elements has to be performed the same way every time. Because faces of complex elements may not be planar, neighboring faces with incongruent edges can create gaps between elements and will prohibit consistent velocity interpolation (see Figure 20).

The method for decomposition of complex elements into tetrahedra described by Kenwright et al. [44] can be used to do on-the fly decomposition during particle tracing. The advantage of using runtime decomposition compared to offline decomposition is that the memory requirements for an USG containing complex cells is much lower than for a tetrahedral mesh.

5.2.3. Numerical Integration

Hull et al. compare various numerical integration methods and recommend using fourth or fifth order Runge-Kutta method for calculations in which the function evaluations are simple [35]. Bogacki et al. describe an embedded Runge-Kutta 4,5 pair [6] that can be used for efficient particle tracing with adaptive stepsize by using the error between RK4 and RK5 as an indicator for stepsize doubling.
5.2. Parallel Particle Tracing on GPGPUs

5.2.4. Neighbor Search

To avoid computationally expensive cell location testing during particle advection, particles leaving individual cells can be detected as part of the tetrahedral decomposition method described in section 5.2.2. Neighbor search [31] can then be applied to locate the new cell that the particle traveled to. This standard method has also been implemented on the GPU e.g. by Schirski to trace particles through unstructured tetrahedral grids [74]. More recently, algorithms that do not require face to cell data [72] to locate the next cell in the particle path have been developed.

5.2.5. Parallelization

Acceleration of particle tracing in HPC can be performed by parallelization over data, parallelization over the particles, or hybrid approaches. Parallelization strategies depend on the amount of data to be visualized:

- For large numbers of particles and relatively small size of the dataset fitting in memory of a shared memory node or an accelerator, a parallelization over the number of particles allows for pleasantly parallel implementation achieving good scalability.

- Although for large data sets that do not fit into memory, parallel tracing in partitioned datasets has to be employed. Particles leaving a domain have to be transferred to different partitions, adding communication overhead. The scalability then depends on the smart partitioning of the data, that has to ensure that particles stay inside a single partition for long periods of time. The approach described by Yu et al. [96] employing hierarchical clustering methods to partition the flow field is an effective method towards this goal.
5.2.6. Particle Rendering

To be able to render large number of particles for the visualization of flow fields, standard methods include the rendering of billboards or textured point-sprites. Using billboards, as well as simple geometry such as quads, has the disadvantage that large numbers of vertices have to be passed through the vertex pipeline of the GPU, seriously affecting rendering performance for large numbers of particles.

Point-sprites define an alternative, where only a single vertex has to be processed for each particle. Although, texture accesses needed to render dynamically lit, rounded points that appear to be spheres viewed from the front, are expensive in terms of rendering performance. To eliminate texture accesses, shader programs can be used rendering dynamically lit “spheres” using point sprites (see Figure 21 and Appendix B.2). This is made possible by automatic generation of texture coordinates on OpenGL’s `GL_POINTS` during rendering using `GL_COORD_REPLACE` as a texture environment for the point. These texture coordinates can be used to detect the position of a fragment inside the particle to calculate appropriate lighting per fragment, or simply discarding the fragments on the outside to provide the notion of circular points. The size of the particle can be set according to the z position of the fragment, possibly augmented with an additional particle size passed as a vertex attribute when different particle sizes have to be supported.
5.2. Parallel Particle Tracing on GPGPUs

(a) Visualization of vortices in data center air cooling (particles)

(b) Vortex rope in a radial water turbine runner

Figure 22.: Massively parallel particle rendering on the GPU using GLSL shaders
5.3. Massive Particle Tracing for Erosion Prediction in Turbomachinery

Erosion is a major component causing material removal in hydraulic machinery [86]. The effects of erosion are abrasive and erosive wear caused by massive particles contained in the water flow that impact on the surface of the machinery. The erosion intensity depends on the sediment type and its characteristics such as shape, size and hardness [62]. Erosion effects may also be combined with wear caused by cavitation [87]. An optimal geometry of turbomachinery has to be developed to minimize these issues.

In this section, methods are presented that allow the engineer to interactively analyze the cause and effect of erosion during computational optimization of turbomachinery components. The implementations are realized to exploit parallel manycore devices such as GPUs, to be able to efficiently augment traditional methods for visualization of cavitation (also see Figure 27c) with the visualization of abrasive and erosive wear.

5.3.1. Tracking Massive Particles in Unstructured Grids

The advection of sediment such as quartz sand is based on the methods presented in Section 5. To be able to efficiently compute erosion caused by massive par-
5.3. Massive Particle Tracing for Erosion Prediction in Turbomachinery

ticles during the post-processing of a CFD simulation, a model using several simplifying assumptions was created:

- **Particles do not affect the flow.** Since the erosion detection is performed completely during post-processing, no backwards interaction between particles and fluid mechanics of the surrounding medium can be performed. The rationalisation behind this assumption is that sediment particles are not a major constituent of the fluid and that backwards interactions can be neglected [62].

- **Particles do not affect surface geometry.** For the same reason, the geometry of the CFD model cannot be changed by post-processing. This also means that in the model, no characteristics of the flow are affected by wear.

- **Particles do not interact with other particles.** For efficient parallel computations, particle-particle interactions are disregarded.

- **Particles are rigid and spherical.** In the model, particles do not change their properties when they collide with surface geometry. For reasons of computational simplicity, particles are approximated by spheres [14].

- **Particles emitted by abrasive wear are discarded.**

Particles in the generated computational model are influenced by gravitation, momentum transfer i.e. viscous drag, buoyancy and added mass force, and the pressure gradient force. The motion of a spherical particle in a viscous flow is derived from Basset, Boussinesq and Oseen [3]

\[
\frac{dv}{dt} = F_D + F_B + F_{VM} + F_P
\]

(5.5)

where \( F_D \) depicts the drag force acting on the particle, \( F_B \) is the buoyancy force due to gravity, \( F_{VM} \) is the virtual mass force and \( F_P \) is the pressure gradient force.
The drag force $F_D$ is proportional to the relative fluid flow velocity between the particle and the fluid. It is defined by

$$F_D = \frac{1}{2} C_D \rho F A_F |U_F - U_P|(U_F - U_P) \quad (5.6)$$

where $C_D$ is the drag coefficient, $A_F$ is the effective cross section of the particle, and $U_F$ and $U_P$ are the velocities of the fluid and of the particle respectively.

The drag coefficient is a term determined by experimental data used to quantify the drag of an object in a viscous fluid [33] [73]. It is defined by

$$C_D = \begin{cases} \frac{24}{Re} & \text{if } Re \leq 1 \\ \frac{24}{Re}(1 + 0.15 Re^{0.687}) & \text{if } 1 < Re \leq 1000 \\ 0.44 & \text{if } 1000 < Re \leq 2.5 \times 10^5 \end{cases} \quad (5.7)$$

in which the first case depicts the *Stokes Region*, the second case is a transitional region, and the third case represents *Newton’s Law Region* where the drag coefficient is constant.

The buoyancy force $F_B$ on a particle is equal to the weight of the fluid that is displaced by the particle. It is defined by

$$F_B = (m_P - m_F) g \quad (5.8)$$

with $g$ being the gravity vector, $m_P$ and $m_F$ depicting the mass of the particle and of the displaced fluid respectively.

The virtual mass force $F_{VM}$ is an additional drag force on the particle, representing the kinetic energy gained by the surrounding fluid due to an accelerated particle. It is defined by

$$F_{VM} = \frac{C_{VM}}{2} \cdot m_F \cdot \left( \frac{dU_F}{dt} - \frac{dU_P}{dt} \right) \quad (5.9)$$

where $C_{VM} = 0.5$ is a non-dimensional mass coefficient for inviscid flow around an isolated sphere [3], $m_F$ the mass of the displaced fluid, and $U_F$ and $U_P$ the velocity vectors of the fluid and the particle respectively.
According to [3], the pressure gradient force is only important if large fluids pressure gradients exist and if the particle density is smaller than or similar to the fluid density. Since particle density will be much higher in this application\(^2\), the pressure gradient force will be ignored.

### 5.3.2. Collision Detection

To compute collision detection, the algorithm for cell traversal presented in Section 5.2 and [43] is extended using *walls* generated from the CAD model of the water turbine. Whenever a particle leaves a CFD cell, the face of the element in the direction of the worst violator in terms of natural coordinates of the particle is checked. If the face that has to be intersected by the path of the particle is marked as as *wall*, the particle is reflected from the wall instead, impacting the surface geometry at the intersection point, adding erosive wear according to a erosion model.

### 5.3.3. Tabakoff and Grant Erosion Model

Given the velocity of a particle, the angle between the path of the particle and the surface normal of the struck geometry, as well as the material of the particle and the surface, the erosion model of Tabakoff and Grant [27] reproduces empirical erosive wear of the surface.

As implemented in ANSYS CFX-12 [3], the dimensionless erosion rate \( E \) is computed by

\[
E = f(\gamma)\left(\frac{V_P}{V_t}\right)^2 \cos^2 \gamma (1 - R_{T}^2) + f(V_{PN})
\]

(5.10)

where \( V_P \) is the particle impact velocity and \( \gamma \) is the impact angle. The impact angle function \( f(\gamma) \) is computed by

\[
f(\gamma) = \left[1 + k_2 k_{12} \sin(\gamma \frac{\pi}{2\gamma_0})\right]^2
\]

(5.11)

\(^2\)the density of quartz is 2.650
(a) Erosion caused by quartz particles on steel
(b) Erosion caused by quartz particles on aluminum walls

Figure 24.: Eroded mass depending on particle mass, angle of collision, speed of particle and wall material (erosion model of Tabakoff/Grant [27])

with $\gamma_0$ depicting the angle of maximum erosion. The other model constant $k_{12}$ and the reference velocities $V_1$, $V_2$ and $V_3$ are used to configure the model dependent on the materials used.

\begin{equation}
R_T = 1 - \frac{V_P}{V_3} \sin \gamma \tag{5.12}
\end{equation}

\begin{equation}
f(V_{PN}) = \left(\frac{V_P}{V_2} \sin \gamma\right)^4 \tag{5.13}
\end{equation}

\begin{equation}
k_2 = \begin{cases} 
1.0 & \text{if } \gamma \leq 2\gamma_0 \\
0.0 & \text{if } \gamma > 2\gamma_0 
\end{cases} \tag{5.14}
\end{equation}

Figure 24 shows a visualization of erosive wear of quartz particles on steel and aluminum walls with respect to different velocities and impact angles according to the Tabakoff and Grant erosion model.
5.3. Massive Particle Tracing for Erosion Prediction in Turbomachinery

5.3.4. Visualization of Erosion

The visualization of erosive wear combines massive particle tracing methods and the surface extraction methods introduced in Chapter 6. Using a statistical distribution of sand particles at the inlet elements of a turbomachinery simulation (also see C.1), massive particles are traced to the outlet elements of the computational mesh. When colliding with surface geometry, the eroded mass is added to the affected surface polygons accordingly. The eroded mass that is computed by the introduced erosion model can then be visualized as color coded regions mapped to the surface polygons of the CAD model.

Additional information about the origin of particles contributing to erosion of a particular segment can be visualized as boundary surfaces of particle paths as can be seen in Figure 25. Efficient extraction of the boundary surface of multiple particle paths is implemented by generating an additional particle density field in the CFD model during particle tracing. The density field can then be used to extract an isosurface containing all CFD elements through which particles contributing to erosion of an interactively indicated segment have travelled, allowing for deeper analysis of potential for optimization regarding the durability of turbomachinery components.

Figure 25.: Particle paths and isosurfaces of particle density for several eroded segments
Chapter 5. Particle Tracing

5.4. Performance

As can be seen in Figure 26, tracing of massive particles adds much less computational overhead on GPUs than on CPUs. Where massive particle tracing on the CPU yields about half the performance of massless particle tracing, the GPU achieves about 74% of the performance compared to tracing massless particles using the same dataset. Even though particle tracing in USGs is not a prime discipline of GPUs since memory accesses cannot be coalesced and code paths diverge frequently between the different threads, there is an overall speedup. The reason why massive particle tracing is able to perform much better than in the CPU implementation, is that the additional computations necessary are overshadowed by memory accesses.
Chapter 6.

Surface Extraction

Measure twice, cut once.

(Proverb)

6.1. Extraction of Surfaces from Unstructured Grids

Direct visualization of a complete, large scale simulation dataset is often impractical because of the occlusion of interesting flow features by other data. Providing tools for interactive reduction of datasets to extract meaningful data via filtering helps users to analyze large-scale flow fields. Displaying extracted surfaces is such an effective way to visualize various characteristics of fluid dynamic simulation results. Scalar fields such as pressure, or scalar data derived from fields of higher dimensionality such as the magnitude of a velocity field, can be efficiently displayed on these surfaces via color coding. To allow users for dynamic probing of data fields, low latency surface extraction and mapping mechanism have to be developed that provide immediate visual feedback to the users control motions, i.e. with a delay of less than 100 ms [8].

In this chapter, GPGPU methods for creating interactive cross-sections of data on USGs and accompanying mapping techniques for scalar and vector data are shown. These methods can be used to extract surfaces both using implicit spatial
functions such as planes or cylinders, and contour surfaces defined by the data values such as isosurfaces.

(a) Cut-Plane
(b) Cut-Cylinder
(c) Isosurface of pressure at outlet (magnitude of velocity color coded)

Figure 27.: Extracted surfaces showing pressure distribution in a radial water turbine CFD simulation (see also Appendix C.1)

6.1.1. Surface Extraction Algorithm

Marching cubes was created to construct isosurfaces from 3-dimensional rectilinear grids, originally intended to be used for the visualization of medical data [54]. Although, the enhanced family of algorithms can also be applied to the extraction of arbitrary surfaces from structured and unstructured datasets (see also Section 2.3.1.1). The basic marching cubes algorithm can be accelerated using either geometric decomposition such as KD trees, value decomposition such
as Cell-Trees (see also Section 2.3.3.3), or employing the sweeping simplices algorithm [80]).

Parallelization is usually achieved by employing a divide and conquer approach to locate the surface within each cell of the grid. Collaboration between the partitions participating in the parallel extraction is needed to achieve results similar to the serial algorithm:

- The combination of extracted partial surfaces from the distinct partitions to form the complete surface.
- The calculation of per-vertex normals for the complete surface.

Figure 28.: Performance of isosurface computation on CPU/GPU

### 6.1.2. Parallel Manycore Implementation

In this section, a method for efficient hybrid manycore extraction of surfaces on USGs is provided that is based on NVIDIA’s implementation for cartesian grids [65]. The algorithm treats unstructured grids as a stream of individual cells that can be processed independently in parallel. Given an input stream of cells,
output streams of triangles, normals and mapped data elements are provided for immediate rendering or further post-processing on the parallel GPU.

### Classification

<table>
<thead>
<tr>
<th>Mesh elements</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of generated vertices</td>
<td>9</td>
<td>0</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>Output</td>
</tr>
<tr>
<td>Classification</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Stream compaction index computation

| Number of generated vertices | 9 | 0 | 12 | 9 | 0 | 0 | 3 | Input |
| Classification | 1 | 0 | 1 | 1 | 0 | 0 | 0 | |
| Exclusive prefix sum (vertices) | 0 | 9 | 19 | 20 | 30 | 30 | 30 | Output |
| Exclusive prefix sum (classification) | 0 | 1 | 2 | 3 | 3 | 3 | 3 | |

### Compact Elements

| Number of generated vertices | 9 | 0 | 12 | 9 | 0 | 0 | 3 | Input |
| Exclusive prefix sum (classification) | 0 | 1 | 2 | 3 | 3 | 3 | 3 | |
| Element index | 0 | 2 | 3 | 6 | 6 | 6 | 6 | Output |

### Scatter triangle vertices to output stream

| Element index | 0 | 2 | 3 | 6 | 6 | 6 | 6 | Input |
| Exclusive prefix sum (vertices) | 0 | 9 | 19 | 20 | 30 | 30 | 30 | |
| Output vertices |  |  |  |  |  |  |  | Output |

Figure 29.: Parallel GPGPU surface extraction

The algorithm is split into multiple steps (see also Figure 29):

- **Classification**: All cells in the USG are classified according to how many vertices are generated by surface extraction in the individual cell. Two output streams are generated as input for the following stages. First, a stream containing the number of vertices generated in the cell, second, a classification of boolean values (0/1) if any vertices in the cell are generated at all.

- **Index computation for stream compaction**: Since it is likely that there are cells in the USG that are not cut by the surface, the output streams have to be compacted. To be able to leave out empty cells, exclusive prefix sums for the aforementioned output streams are computed to be used in the next stage.

- **Stream compaction**: Using the stream containing the number of generated vertices per cell, along with the stream containing the exclusive prefix sum...
of the boolean classification, starting indices for vertices in a compacted output stream can be computed forming an element index stream.

- Triangle generation, normal generation and data mapping: With the element index stream designating the position of triangle vertices in the surface output stream, and the stream containing the exclusive prefix sum of generated vertices, the generated triangles can be independently scattered to the output stream. During generation of the triangle output stream, further compacted streams containing per face normals and interpolated data on vertices are generated.

Using this algorithm, the actual extraction of the surface is independent per cell, allowing for a scalable pleasantly parallel execution on parallel GPGPU clusters. Stream compaction to allow for a compact representation of the resulting triangle mesh is based on building parallel prefix sums, which introduces an additional computational complexity of $O(\log_2(n))$, where $n$ designates the number of cells in the USG. Combination of the resulting surfaces can be achieved in object space by copying of triangular meshes, or in screen space using parallel rendering. Since large numbers of triangles are very data intensive, parallel rendering approaches are vastly more efficient for relatively low resolution display screens (see also Section 2.4.4).

![Comparison between face- and vertex normals](image)

(a) Per face normals  
(b) Per vertex normals

Figure 30.: Comparison between face- and vertex normals

In addition to the presented algorithm that generates per face normals, per vertex normals can be generated from the resulting triangular mesh in a post-processing
(see Figure 31). Using neighbor information in the USG, the vertex normals can be interpolated from the face normals by revisiting cell neighbors based on the element index stream. This process is computationally expensive and can be very time consuming depending on the number of generated triangles.

Since memory on existing manycore devices is scarce compared to the host-memory of current compute cluster nodes, data has to be constantly copied to the device when applying surface extraction on transient datasets. Using double-buffering, communication and computation can be overlaid as can be seen in Figure 31 without any additional overhead in the coal combustion chamber dataset.

Figure 31.: Surface extraction runtime performance on different generation GPUs

Performance analysis of the parallel manycore implementation using stream computing show various interesting results on different generation NVIDIA GPUs (see Figures 28 and 31).

- A single previous generation Quadro FX 5800 is about as fast as 16 Intel XEON CPUs applying isosurface extraction on the medium-size coal combustion chamber dataset.

- The speedup using multiple GPUs connected via MPI over a high-speed Infiniband interconnect is decreasing when more triangles are extracted because of the overhead of combining the resulting triangle meshes over the network.

- The memory intensive classification of cells where every cell and all the vertices in each cell have to be accessed is much faster on newer genera-

---

1Quadro FX 5800 chipset GT200, Quadro FX 6000 chipset GF100
6.1. Extraction of Surfaces from Unstructured Grids

tion GPUs. Although these highly irregular memory accesses needed in the processing of USGs still are a bottleneck for parallel manycore algorithms.

- Transfers of memory from host to device e.g. for the preparation of the next timestep in transient datasets is not much faster on the newer generation devices, but still can be hidden by overlaying communication and communication.

- The performance of surface extraction on manycore devices allows for interactive feedback to user interactions. Due to the mostly pleasantly parallel nature of the algorithm, meshes can be distributed to the nodes in the post-processing cluster for scalable parallel execution.

Figure 32.: Visualization of temperature distribution in a coal combustion chamber simulation (see Dataset C.2) using surface extraction methods
6.2. Texture-based Feature Visualization

Dense, texture-based visualization techniques have been introduced in Section 3.2.3. In contrast to geometric, indirect visualization methods, these techniques can be used to provide a complete coverage of large-scale vector fields [47].

Given streamlines $\sigma$ parametrized by arclength $s$ that are defined by

$$\frac{d}{ds}\sigma(s) = v(\sigma(s), t) \quad (6.1)$$

and the initial condition

$$\sigma(0) = \sigma_0 \quad (6.2)$$

the LIC algorithm computes intensities for each texel located at $x_0 = \sigma(s_0)$ by evaluating the convolution integral

$$I(x_0) = \int_{s_0-L}^{s_0+L} k(s - s_0)T(\sigma(s))ds \quad (6.3)$$

where $\sigma$ is the streamline through the center of the texel, $k$ is a given filter kernel with length $L$, and $T$ is a texture filled with white noise [81].

In this section, a method for parallel LIC is introduced, that provides fast, interactive visualizations of flow on surfaces extracted from USGs.

6.2.1. Parallel Line Integral Convolution on Manycore Devices: FragmentLIC

The presented algorithm is able to calculate convolution integrals on surfaces extracted from USGs using parallel manycore devices. It avoids several shortcomings of previous developments:

- The unstructured grid does not need to be resampled to a structured grid at any time of the algorithm. This avoids artifacts introduced by interpolation.
6.2. Texture-based Feature Visualization

Figure 33.: Visualization of vortices in data center air cooling, including additional background geometry

- The flow field does not have to be projected to the surface where LIC is applied. Instead, particle advection is performed on the original CFD grid. Projection of the vector field is only appropriate for surfaces that are close approximations of the local flow field, i.e. stream surfaces. Otherwise, aliasing artifacts are added to the visualization, possibly leading to erroneous conclusions about the underlying flow.

- Streamline computation is performed *per fragment* of the generated image, providing a decoupling of image resolution and object space. Therefore, both coarse visualizations stemming from large triangles, and duplicate computations for small objects are eliminated.

- Readback of rendered images from the GPU to CPU-accessible memory is not required, unless parallel or remote rendering is performed and copying of image data to remote hosts is essential.

The algorithm provides integrated surface extraction and computation of line integral convolution on these surfaces using multiple rendering and computation phases. The surface extraction methods described in section 6.1.2 and the particle tracing technique shown in section 5.2 are combined to provide generation of LIC visualizations at interactive framerates using programmable GPUs.
Some additional features extending the aforementioned algorithms are needed to be able to efficiently combine the techniques.

- In the first phase of the FragmentLIC algorithm, extraction of one or more surfaces is performed as depicted in section 6.1.2. The surface extraction algorithm is extended to provide, for each extracted triangle, the \textit{element index} of the cell in the USG it was extracted from.

- In the first render pass, the indices are provided as vertex attributes to a GLSL shader program along with the triangle vertices of the extracted surface (see Appendix B.1). The triangles are then rendered to a floating point RGBA buffer. In the vertex shader, the world coordinates of the triangle vertex along with the \textit{element index} are passed on to the fragment shader. The fragment shader stores the interpolated world coordinates of the fragment in the RGB channel of the render target. The alpha channel is used to store the \textit{element index}. The resulting ‘image’ in the render target consists of the world coordinates for the center of each fragment and the index of the cell in the original USG grid (see Figure 34).

- Following the first render pass is an implementation of LIC as a post-
processing of the generated floating point RGBA image using CUDA kernels. For every fragment in the 'image' containing an alpha value that is a valid cell in the USG, computation of the convolution integral through the center of this fragment is performed (see Figure 35). The starting point of the streamline computation is known to be in the grid cell referenced in the alpha buffer. Therefore, the computationally expensive point location operations described in section 5.2 can be avoided. The output of the post-processing kernels is an image where the fragments containing the rendered surface have been replaced by the intensity values resulting from the computation of the convolution integral, merged with colors designating scalar properties of the flow, e.g. the magnitude of the local flow field. Fragments containing image background, e.g. additional geometry accompanying the LIC visualization, can be copied from the floating point RGBA image. Geometry that contains transparent elements has to be rendered in an additional render pass and blended with the image generated in the post-processing pass.
6.2.2. Performance Considerations on GPGPUs

As the performance of the presented algorithm mainly depends on the number of fragments used for post-processing of the image (see Figure 37), it benefits enormously from *sort-first* parallelization by partitioning the image space, allowing multiple GPUs to process parts of the dataset in parallel.

Although no reuse of calculated streamlines is performed (since the GPGPU particle tracing implementation does not support it), the combined surface extraction and particle tracing benefits from current GPGPU architectures. Especially at close viewing distances, partitioning of the image space into 2D blocks for processing using CUDA kernels, makes it probable that fragments processed inside the same block are part of the same USG cell. The individual threads participating in the computation of streamlines in a single block are then able to exploit cached access to USG datastructures on the GPU, considerably increasing performance (see Figure 38).
6.2. Texture-based Feature Visualization

Figure 37.: Performance of FragmentLIC depending on the size of the generated image, using a fixed number of triangles extracted from the datacenter dataset

Figure 38.: Performance of FragmentLIC depending on the number of triangles extracted from the datacenter dataset, using a fixed number of fragments (1.048.576)
Chapter 7.

Summary and Conclusion

We can only see a short distance ahead, but we can see plenty there that needs to be done.

(Alan Turing)

The analysis of simulation data during the virtual product development cycle is alleviated tremendously by interactive post-processing. Traditionally, post-processing algorithms used to examine flow features in simulation data have been executed on special purpose workstations. With remarkably detailed simulation results generated on today’s highly parallel HPC clusters, even separate post-processing clusters are inadequate for interactive analysis, since these datasets frequently are even too large to be transferred between the compute cluster and the post-processing cluster in adequate time. Engineers presently often have to settle for exploration of temporal or spacial subsets of simulation data, data sampled to coarser grids, or non-interactive post-processing methods.

7.1. Contributions

To be able to interactively analyze complete, unmodified large-scale simulation data, post-processing environments have to be able to take advantage of highly parallel cluster resources.
In this work, a framework was introduced for the execution of post-processing algorithms in parallel manycore environments. The main focus was put on achieving interactive framerates during evaluation of large simulation data on unstructured grids, without the need for resampling of data to structured grids or coarser unstructured grids. *vistle*, a dataflow-oriented software infrastructure for a parallel post-processing system based on HPC mechanisms was presented, that was designed and developed during the preparation of this work. *vistle* is using a distributed data model to make use of a given domain decomposition obtained from the partitioning created during pre-processing or simulation by the numerical solver. It allows for parallel modules, implementations of post-processing algorithms, to access distributed data objects either for independent or collaborative parallel processing.

The parallel runtime environment enables *vistle modules* to utilize the various levels of parallelism existent in modern HPC cluster systems. The interactive post-processing algorithms that have been developed using the *vistle* architecture demonstrate the usage of different parallelization techniques. Parallel surface extraction algorithms have been designed and implemented using distributed stream-processing capabilities, supported on current HPC hardware using either GPGPU manycore devices or alternative accelerators. Parallel particle tracing methods using hybrid parallelization methods have been implemented both for the interactive assessment of flow fields, as well as for the detection of areas vulnerable to erosion due to sand-particles in turbomachinery. Combinations of parallel surface extraction and particle tracing to enable dense, texture-based visualization of fluid simulation data, has been shown to generate interactive framerates without the need for projection of the flow field to the extracted surface used in competing methods. The algorithms have been evaluated using several real-world datasets.

### 7.2. Conclusions

Surface extraction techniques, as well as visualization methods based on particle tracing and combinations thereof, still play a highly important role in the
analysis of fluid dynamic simulation results. Since parallel rendering and composition is achieving interactive framerates even on massively parallel cluster systems, interactive parallel post-processing is becoming feasible for the analysis of very large partitioned simulation data.

Future manycore CPUs developed for HPC systems will likely combine features found today in programmable graphics hardware, relatively simple compute cores allowing for high-throughput computations, with general purpose processing units. The availability of manycore computing resources in the form of GPGPUs makes integrated post-processing and rendering feasible even today, avoiding or hiding expensive copying of data between the different memories and address-spaces of CPUs and GPUs and thus further decreasing latency.

The applicability of exploiting large scale manycore HPC compute resources for remote post-processing while maintaining interactivity was shown. This enables engineers to analyze full-scale fluid dynamic simulation results without sacrificing accuracy due to the usage of sampling or coarsening methods.

7.3. Outlook

7.3.1. Massively Parallel Post-Processing Algorithms

Interactive parallel post-processing using domain-decomposition techniques have been demonstrated to be attainable for different algorithms required for the exploration of fluid dynamic simulation data. Stream computing on many-core devices has been shown to be a useful technique even for algorithms that necessitate reduction of data, as is often the case in post-processing methods. The approaches that have been presented might also proof to be highly beneficial for three-dimensional feature-based visualization methods that rely on the advection of massive number of particles.
7.3.2. Reliability and Fault Tolerance in HPC Algorithms

The probability of hardware malfunctions during large-scale computations rises in combination with the ever increasing number of compute cores involved. Prospective generations of HPC middleware libraries such as MPI and PVM will have to provide features to support recovering from process failure. The prevailing method of checkpointing and restarting of failed computations will likely be replaced by application focused, Algorithm Based Fault Tolerance (ABFT) techniques [40], which will allow recovery with minimum overhead. Regarding massively parallel post-processing systems, replication features currently used to support load balancing might be able to provide means for maintaining interactivity even during process deterioration or breakdown.

7.3.3. Massively Parallel Rendering of Large Post-Processing Results

Most of the current parallel post-processing and visualization systems depend on remote parallel rendering using scanline rendering, preferably hardware accelerated using dedicated GPUs. Future manycore HPC systems without specialized rasterization hardware might make rendering methods such as CPU-based ray-casting appear far more attractive. Of course, this brings up other interesting possibilities e.g. for direct rendering of isosurfaces and cut-surfaces even in USG-based simulation datasets, bypassing the need for the very expensive prior extraction of large numbers of triangles.
Appendix A.

Unstructured Grids

A.1. Data Structures

Element lists allow for the reuse of vertices in the vertex list within neighboring grid elements. Since vertex coordinates are data intensive, this sharing of coordinates drastically reduces the amount of memory needed to store an USG. In some algorithms, memory access times can be reduced since less data elements have to be accessed compared to using data structures where coordinates...
Appendix A. Unstructured Grids

are replicated. Although in other algorithms, the indirection introduced by the
element list drastically decreases performance.

A.2. Decomposition of Grid Elements into
Tetrahedra

(a) Decomposition of a pyramid
(b) Decomposition of a prism
(c) Decomposition of a hexahedron

Figure 40.: Decomposition of USG elements into tetrahedra

Complex elements in USGs can be efficiently decomposed into tetrahedra to
perform robust particle tracing techniques. Continuity between decomposition
of neighboring grid elements has to be taken into account as is shown in Figure
20 on page 52.

A.3. Marching Elements for Surface
Extraction
Figure 41.: Marching tetrahedra triangle lookup table

Figure 42.: Marching pyramids triangle lookup table

Figure 43.: Marching prisms triangle lookup table

Figure 44.: Original marching cubes lookup table [54]. For topologically correct isosurfaces see [11] and [64]
Appendix B.

GLSL Shaders

B.1. LIC on GPU

Listing B.1: GLSL vertex shader for first LIC render pass

```glsl
#version 130
#extension GL_EXT_gpu_shader4 : enable
attribute int elem; // index of CFD element from VBO
flat out int element; // index of CFD element to pass to fragment shader
out vec3 position; // world coordinates of vertex to pass to fragment shader

void main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  position = gl_Vertex.xyz; // pass world coordinates of vertex to
  // fragment shader
  element = elem; // pass CFD element index of vertex to
  // fragment shader
}
```

Listing B.2: GLSL fragment shader for first LIC render pass

```glsl
#version 130
#extension GL_EXT_gpu_shader4 : enable
flat in int element; // index of CFD element
in vec3 position; // interpolated world coordinates of fragment

void main() {
  // set RGB to interpolated world coordinates, alpha to CFD element index
  gl_FragData[0] = vec4(position, element);
}
```
Appendix B. GLSL Shaders

B.2. Particle Rendering

Listing B.3: GLSL vertex shader for particle rendering using GL_POINTS

```glsl
varying vec4 color;
varying vec3 position;

void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    position = vec3(gl_ModelViewMatrix * gl_Vertex);
    gl_PointSize = 1.0 / gl_Position.w; // size adaption of particle ‘point’
    color = vec4(gl_Color.xyz, 1.0);
}
```

Listing B.4: GLSL fragment shader for particle rendering using GL_POINTS

```glsl
// render a ‘sphere’ on a square point
// - discard fragments that exceed the sphere’s radius from the center
// - light fragments according to a normal vector on the sphere’s surface
varying vec4 color;
varying vec3 position;

// texture coordinates are generated across the fragments of the GL_POINT
uniform sampler2D tex;

const float PI = 3.14159265;

void main() {
    // normalize fragment coordinate to ([-1, 1], [-1, 1])
    float x = (gl_TexCoord[0].x - 0.5) * 2;
    float y = (gl_TexCoord[0].y - 0.5) * 2;
    vec3 light = normalize(vec3(gl_LightSource[0].position));
    vec3 ptl = normalize(light - pos);

    // if fragment distance from midpoint is > 1, discard
    // hereby rendering a disc from a quad point
    float distance = sqrt((x * x) + (y * y));
    if (distance > 1.0)
        discard;

    // calculate a normal vector on the surface of a sphere
    float v = distance * PI / 2.0;
    vec3 normal = normalize(vec3(sin(v), sin(v), cos(v)));

    // light the sphere
    float spec = max(dot(reflect(-ptl, normal), normalize(-pos)), 0.0);
    vec3 color = vec3(0.15, 0.15, 0.15); // ambient light
    color += col * dot(normal, light); // diffuse light
    color += max(0.0, 0.3 * pow(spec, 20.0)); // specular light
    gl_FragColor = vec4(color, 0.0);
}
```
Appendix C.

Datasets

C.1. Francis Turbine Runner\(^1\)

The francis turbine runner dataset uses an USG consisting of 11 channels containing 15,215,488 elements and 15,490,831 vertices each, resulting in a total amount of 167,370,368 volume elements and 170,399,141 vertices. The simulation results include per-vertex pressure and velocity data. The numerical solver used to simulate the turbine is ANSYS CFX.

Figure 45.: Geometry of the radial water turbine runner

\(^1\)Dataset courtesy of Stellba Hydro GmbH & Co KG, Heidenheim
Appendix C. Datasets

C.2. Coal Combustion Chamber

This coal combustion chamber simulation dataset uses an USG split into 32 parts. It contains 11,626,484 elements and 12,431,626 vertices in total. The simulation results include per vertex temperature, velocity and various species concentrations such as CO$_2$, O$_2$, NO$_x$, coke, coal and ash. The numerical solver used for the simulation of coal combustion is RECOM AIOLOS.

Figure 46.: Surface grid of the high resolution CFD model

Figure 47.: Numerical simulation of combustion in a coal power plant

---

2Dataset courtesy of RECOM Services GmbH, Stuttgart
C.3. Data center

This airflow simulation is one of various different design possibilities analyzed during the optimization of the data center for the installation of the NEC SX9 supercomputer at HLRS. The dataset consists of an USG containing 5,575,896 elements and 5,836,713 vertices. It contains per-vertex temperature and velocity data. The numerical solver used to simulate the air conditioning is ANSYS CFX.

Figure 48.: Numerical simulation of airflow in a data center

---

3 Dataset courtesy of High Performance Computing Center Stuttgart (HLRS)
Bibliography


[50] LEE, T.-Y., RAGHAVENDRA, C. S., AND NICHOLAS, J. B. Image composition schemes for sort-last polygon rendering on 2D mesh multicom-


[78] **Shan, H., Wright, N. J., Shalf, J., Yelick, K., Wagner, M., and Wichmann, N.** A preliminary evaluation of the hardware acceleration


Index

BSP-Tree, 13
Cell-Tree, 13, 51
Cut-Cylinder, 10, 64
Cut-Plane, 10, 64, 69
Erosion, 56, 61
GPGPU, 2, 15, 16, 47, 66, 78
Interval-Tree, 13
Isosurface, 10, 64, 65
KD-Tree, 13, 51
Line Integral Convolution, 30, 70
Manycore, 33, 34, 66, 79
Marching Cubes, 10, 64
MPI, 15, 65, 68
Octree, 13
OpenMP, 15, 65
Parallel Rendering, 17, 19, 22, 48
Parallelism, 15, 27, 38, 65
Particle Rendering, 54
Particle Tracing, 11, 29, 49, 56, 61, 78
Post-Processing, 5, 23, 27, 34, 47, 79
PVM, 15
Stream Compaction, 65, 66
Stream Processing, 37, 65
Structured Grid, 8
Surface Extraction, 10, 63, 78, 79
Tabakoff and Grant, 59
Triangle Lookup Table, 10, 82
Unstructured Grid, 8, 35, 36, 51, 56, 63, 65, 77, 81
Visualization Pipeline, 5, 24, 36