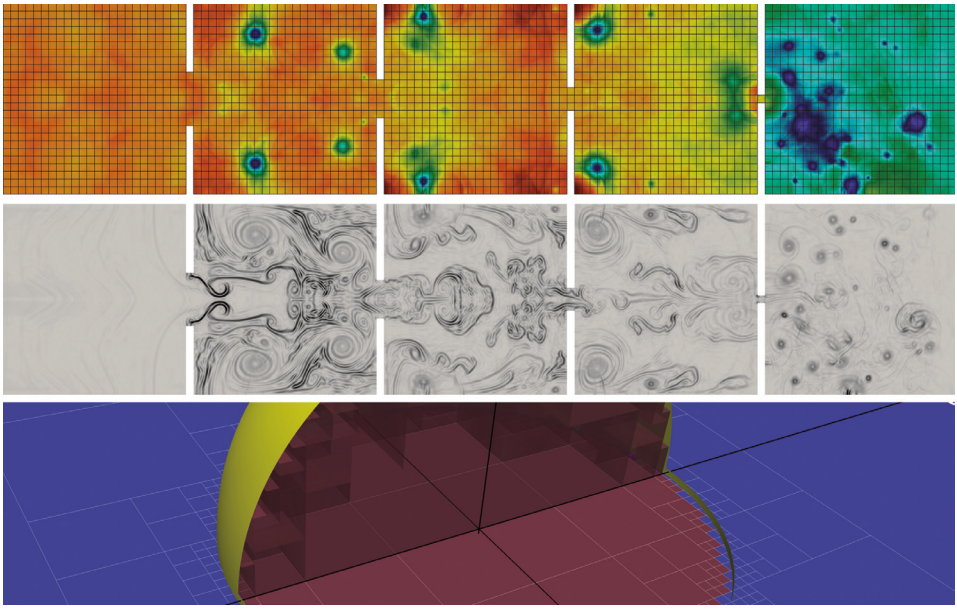
Simulation Techniques in Siegen
Vol. 1

Harald Klimach

Parallel Multi-Scale-Simulations with Octrees and Coupled Applications



universi
UNIVERSITÄTSVERLAG SIEGEN

Parallel Multi-Scale Simulations with Octrees and Coupled Applications

Harald Klimach

Simulation Techniques in Siegen / STS

Edited by Sabine Roller and Harald Klimach

Vol. 1 (2016)

Parallel Multi-Scale Simulations with Octrees and Coupled Applications

Parallele Multi-Skalen Simulationen mit Octrees und gekoppelte Anwendungen

Von der Fakultät für Maschinenwesen der
Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
genehmigte Dissertation

vorgelegt von

Harald Klimach

Berichter: Univ.-Prof. Dr.-Ing. Sabine Roller
Univ.-Prof. Marek Behr Ph.D.
Univ.-Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael Resch

Tag der mündlichen Prüfung: 28.01.2016

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at **http://dnb.d-nb.de**.

D 82 (Diss. RWTH Aachen University, 2016)

Simulation Techniques in Siegen Vol. 1 / STS Vol. 1 (2016)
Editors: Sabine Roller and Harald Klimach

1st edition univers*i* – Universitätsverlag Siegen 2016

Printing: UniPrint, Universität Siegen
printed on wood- and acid-free paper

univers*i* – Universitätsverlag Siegen
Am Eichenhang 50
57076 Siegen
Germany
info@universi.uni-siegen.de
www.uni-siegen.de/universi

ISBN: 978-3-936533-82-8

Meinen lieben Eltern Barbara und Kurt.

"Wenn ein Mensch ein Loch sieht, hat er das Bestreben, es auszufüllen. Dabei fällt er meistens hinein."

Kurt Tucholsky

Danksagung

Die vorliegende Dissertation ist über einen langen Zeitraum entstanden und im Laufe dieser Zeit durfte ich vielen Menschen begegnen, die mir auf die eine oder andere Weise beim Erstellen dieser Arbeit weitergeholfen haben.

An erster Stelle danke ich meiner geliebten Frau Astrid, die mir immer den Rücken frei gehalten hat. Ohne sie hätte ich nicht die Kraft gefunden diese Arbeit zu meistern. Auch meiner Betreuerin Frau Prof. Sabine Roller danke ich aufs herzlichste, sie hat immer daran geglaubt, dass ich diese Arbeit tatsächlich fertigstellen kann. Sie hat mich den ganzen Weg begleitet und es mir so ermöglicht diese Dissertation zu verfassen. Ich bedanke mich auch besonders bei Herrn Prof. Claus-Dieter Munz, durch den ich mit dem Gebiet der Numerik in Kontakt gekommen bin und der mir die ersten Schritte auf diesem Gebiet ermöglicht hat. Herrn Prof. Michael Resch habe ich meine erste Anstellung und eine wunderbare Zeit am Höchstleistungsrechenzentrum in Stuttgart zu verdanken. Deshalb freue ich mich sehr, dass er sich auch der Berichterstattung zu dieser Arbeit angenommen hat. Am HLRS konnte ich wertvolle Erfahrungen im High-Performance Computing erwerben, und ich möchte mich bei allen Kollegen aus dieser Zeit für die herzliche und offene Zusammenarbeit bedanken. Ganz besonders möchte ich mich bei Herrn Uwe Küster bedanken, der stets ein offenes Ohr und einen guten Rat für mich zur Hand hatte und von dem man alles lernen kann, insbesondere über Fortran. Herzlicher Dank auch an Rolf Rabenseifner, dessen Büro ich lange teilen durfte und mir oft bei Fragen rund um MPI behilflich war und ist. Ebenso danke ich Thomas Bönisch für die Gelegenheit mit Partnern aus ganz Europa so eng zusammenarbeiten zu können. Ich danke auch Jens Utzmann, der die Grundlage für diese Arbeit gelegt hat und mit dem ich mich intensiv und lange austauschen durfte.

Ebenso danke ich meinen Kollegen an der German Research School for Simulation Sciences in Aachen, mit denen ich in neue Gebiete aufbrechen durfte und neue Ideen ausprobieren konnte. Ich danke Herrn Prof. Marek Behr für seine Leitung der Einrichtung, und dass er die Berichterstattung für diese Arbeit angenommen hat. Manuel Hasert möchte ich danken, dass er uns allen in der Arbeitsgruppe mit seinem hohen Einsatz und viel Verständnis zur Seite gestanden hat und wir so auch gemeinsam schwierige Phasen, die wohl jede Softwareentwicklung mit sich bringt, überstehen konnten. Auch für seine Kommentare und Korrekturen zu

dieser Arbeit bin ich ihm zu tiefem Dank verpflichtet. Bei Christian Siebert bedanke ich mich für die vielen anregenden Gespräche und die wertvollen Ratschläge zur parallelen Programmierung. Ihm habe ich viele gute Ideen zu verdanken. Er hat immer völlig uneigennützig seine scharfsichtigen Einsichten mit mir geteilt. Jens Zudrop gilt mein ganz besonderer Dank. Durch sein enormes Arbeitspensum und seine tiefen Einsichten, die er in seine Arbeit mitgebracht hat, haben sich uns ganz neue Möglichkeiten und Wege eröffnet. Nur dank aller meiner Kollegen an der German Research School konnte der zweite Teil dieser Dissertation so entstehen wie sie heute vorliegt.

Auch meinen Kollegen an der Universität Siegen gilt natürlich mein ganz herzlicher Dank, insbesondere Kannan Masilamani und Daniel Harlacher, die wesentlich die Erstellung unseres Gittergenerators betrieben haben. Ich hatte das unendliche Glück, stets in einer wohlwollenden Atmosphäre mit freundlichen und kompetenten Kollegen an dieser Dissertation arbeiten zu dürfen.

Ich danke meinen Eltern Barbara und Kurt, die es mir ermöglicht haben dieses Studium zu absolvieren und mich seit jeher auf meinem Weg unterstützt haben. Ebenfalls danke ich meinen Schwiegereltern Erika und Heinz Fandel, die uns in vielem das Leben einfacher gemacht haben. Meinen Schwestern und Brüdern danke ich für ihre vielen Anregungen und insbesondere meinem Bruder Martin für das Korrekturlesen der Dissertation.

Harald Klimach

Zusammenfassung

Simulationen physikalischer Gegebenheiten müssen oft das Zusammenspiel vieler Phänomene und Skalen berücksichtigen. Bei aeroakustischen Problemen zum Beispiel, muss beides berücksichtigt werden, sowohl die Strömung, die den Lärm erzeugt, als auch der Transport der Schallwellen. In dieser Arbeit werden numerische Ansätze für solche Probleme auf großen, verteilt parallelen Rechensystemen untersucht. Das Kopplungsframework *KOP* wird, soweit wie möglich, parallelisiert und ein neues Framework (*APES*) wird entwickelt um fundamentale Beschränkungen der Skalierbarkeit zu überwinden. In beiden Implementierungen werden Verfahren hoher Ordnung eingesetzt, da diese eine hochauflösende Simulation mit weniger Freiheitsgraden ermöglichen, als Verfahren niedrigerer Ordnung. Diese Eigenschaft von Verfahren hoher Ordnung ist ein wichtiger Vorteil auf modernen Supercomputersystemen, da der Speicher, der benötigt wird um die Freiheitsgrade abzubilden eine knappe Ressource darstellt. Die vorgestellten Methoden ermöglichen die transiente Simulation von Mehrskalenproblemen, allerdings werden für detaillierte Simulationen noch immer große Mengen an Rechenressourcen benötigt. Im Rahmen dieser Arbeit wird deshalb ein Fokus auf die effiziente Nutzung moderner Rechensysteme gelegt.

KOP verwendet diskrete Punkte um die Kopplung zu realisieren. Dies erlaubt die Interaktion zwischen Gebieten mit unterschiedlicher Diskretisierung aber auch verschiedenen Gleichungen. Beide Implementierungen verwenden eine explizite Zeitintegration um die zeitabhängigen Simulationen aufzulösen. Das Kopplungsframework erlaubt von Gebiet zu Gebiet variierende Zeitschritte. Diverse Erhaltungsgleichungen, von linearisierten Euler Gleichungen und den Maxwell Gleichung, bis hin zu den Navier-Stokes Gleichungen, können mit den vorgestellten Verfahren gelöst werden. Ein vollständig verteilter Kopplungsmechanismus wird im Rahmen der Arbeit entwickelt, der es ermöglicht, diese Gleichungen in großen Simulationen gekoppelt zu verwenden.

APES erlaubt die Verwendung spektraler Diskretisierungen. Dazu bringt es eine eigene Toolchain mit, die eine skalierbare Ausführung der gesamten Simulation sicherstellt. Insbesondere, umfasst dies auch einen Gittergenerator, der Geometrien mit Polynomen hoher Ordnung darstellen kann. Das robuste Verfahren, das hier zum Einsatz kommt ermöglicht es, ingenieurtechnische Fragestellungen auch mit solchen Verfahren hoher Ordnung in Angriff zu nehmen.

Abstract

Physical simulations often require the consideration of many phenomena and scales. For example in aeroacoustic problems, both, the flow generating the noise and the sound wave propagation needs to be considered. This work investigates numerical approaches to such problems on large distributed and parallel computing systems. The coupling framework *KOP* is parallelized as far as possible and to overcome fundamental scalability limits a new framework *APES* is developed. Both implementations utilize high-order discretizations, as these allow for accurate simulations with less degrees of freedoms than lower order methods. This property of high-order methods is an important feature for modern supercomputing systems, as memory to represent degrees of freedom in a simulation is a scarce resource. The presented methods enable the transient simulation of multi-scale setups but detailed resolutions still require large amounts of computational resources. A focus is put on the efficient utilization of modern computing systems to address this need. Besides the scalability of the implementations, the importance of single core optimization and vectorization is illustrated.

KOP uses discrete points to realize the coupling and allows for the interaction between domains with differing discretizations and solved equation systems. Arbitrary mesh configurations are supported and both, structured and unstructured mesh solvers are available in the framework. In both framworks explicit time integration methods are deployed to resolve the time dependent simulations. The coupling allows for a varying time step width over the participating domains by a sub-cycling method. Various conservation laws can be solved by the presented frameworks ranging from Maxwell's equations and linearized Euler equations to full compressible Navier-Stokes equations. A fully distributed coupling approach is developed that allows for coupling of those in a large-scale simulation to solve, for example, aeroacoustic problems.

APES enables high-order discretizations in the spectral regime. It involves a fully scalable toolchain for mesh-based simulations featuring a mesh generation and a post-processing tool to support the solvers. The common foundation of these tools is an Octree representation for the mesh, and this work specifically covers the generation of high-order geometry approximations in the developed mesh generator *Seeder*. This robust mechanism works for arbitrarily complex surfaces and offers a practical way to tackle engineering tasks with spectral element discretizations.

Contents

Nomenclature

Symbols

e	Energy density
ℓ	Subresolution levels
L	Octree level
$\boldsymbol{I}$	Identity matrix
$\boldsymbol{S}$	Stress tensor
p	Pressure
s	Level Offset in Octree
T	Temperature
t	Tree ID in Octree
$\vec{v}$	Velocity
$\vec{F}$	Fluxes
$\vec{P}$	Primitive variables
$\vec{U}$	State variables
$\vec{W}$	Viscous flux
$\vec{B}$	Magnetic field
$\vec{D}$	Displacement field
$\vec{E}$	Electric field
$\vec{j}$	Current density
z	Z curve value in Octree

Greek Symbols

ϵ	Permittivity
γ	Isentropic expansion coefficient
κ	Heat conductivity
λ	Bulk viscosity
μ	Permeability
μ_v	Dynamic viscosity
ρ	Density
ρ^e	Charge density

Other Symbols

$\otimes$	Dyadic product

Acronyms

ADER	Arbitrary high order using DERivatives
Aotus	Advanced options and tables in universal scripting
APES	Adaptable Poly-Engineering Simulator
Ateles	Adaptive tree based efficient and eithe equation solver
DG	Discontinuous Galerkin
Dof	Degree of freedom
EE	Euler Equations
FDM	Finite Difference Method
FVM	Finite Volume Method
GEUM	General Elemental Unstructured Mesh
HPC	High Performance Computing
IO	Input/Output
KOP	Kopplungs-Framework
LEE	Linearized Euler Equations
MpCCI	Mesh-based parallel Code Coupling Interface
MPI	Message Passing Interface
NSE	Navier-Stokes Equations
PACX	PArallel Computer eXtension
PreCICE	Precise Code Interaction Coupling Environment
TreElM	Tree based Elemental Mesh
WENO	Weighted Essentially Non-Oscillatory

1 Introduction

Advancing the tools available in engineering design requires increasingly the consideration of a multitude of interacting phenomena. While many problems are quite well understood when investigated in isolation, their interactions are often only coming into reach of investigation with the increasing computational power of supercomputing facilities. One of the most demanding computational tasks in engineering is the simulation of fluids in motion. Fluids play a significant role in many technical devices, and the prediction of their behavior is therefore crucial for a functional design of such devices. Phenomena occurring in flows of interest can span a wide range of scales in space and time. This is especially the case for aero-acoustic problems, where the sound is generated by a highly energetic flow on a small turbulent scale, but the sound waves are propagated over large distances, carrying only a small amount of energy. Figure 1.1 illustrates the setup. The figure shows a tiny sphere in the middle of the domain and the passing flow generates a Kármán vortex street in its wake. Those vortices produce sound waves, which we see in the shown cut through the instantaneous pressure field. They are propagated over large distances, and the far-field needs to cover this vast space. Different domains are indicated by black lines separating them from each other. In the presented work, direct numerical simulations are considered, and the specific target application is a direct aero-acoustic simulation. A uniform resolution of such problems, involving several scales is out of reach for today's numerical methods and available computing devices. Instead, the traditional approach to solving such problem classes is the separation and isolated computation of the various scales. However, this has the negative effect of neglecting tightly coupled interactions between the different simulated parts. Usually, there is just a flow of information in one direction without any feedback. For direct simulations, the goal is a minimization of modelling assumptions. While simplifications are needed and a separation into separately treated domains seems to be inevitable, the coupling considered in this work is bidirectional and well suited for direct simulations. The domain boundaries, drawn in the fluid domain are somewhat arbitrary, but in the studied field, they usually can be estimated fairly well.

The goal of this work is to develop a parallel coupling scheme for fluid domains which enables tightly coupled interactions between different scales. This scheme is designed to be as general as possible and allows the computation of multi-scale problems without much more computational effort than the separated approach. As the solution of each fluid domain in itself poses a major computational effort, the scheme has to be employable on using modern supercomputing facilities. Computational resources are increasingly diverse and distributed. Any numerical method, aiming for aggregation of those resources in a single simulation, therefore, has to be capable of dealing with this trend. Thus, the underlying principle of the coupling scheme is driven by the two issues mentioned above: Increasing complexity and distributivity of computational resources, on the one hand, the complexity of the simulations with a proper resolution of all scales by various numerical methods on the other hand.

The presented work started out with the existing serial coupling tool *KOP* and resulted in the completely new designed simulation framework *APES* that overcomes several scalability bottlenecks.

1.1 State of the Art in Coupling Techniques

There is a wide range of coupling concepts and implementations available with different goals, requirements and features. This work treats the coupling of domains where compressible flows are to be solved. For these, only explicit time integration solvers are considered here in the various domains. Thus, the focus in this short overview on contemporary coupling methods is put on the ones relevant to this kind of application.

Meshes and interfaces can take different forms, the easiest meshes to couple are those with matching nodes at the interfaces. Such setups might for example arise where different equations are to be solved in the separated spatial domains. More complicated are those configurations, where the the nodes are not matching at the mesh interfaces. This type of interfaces is typical for separations of different scales, where one spatial region needs to be better resolved than the other (Chen et al. [7]). A special case in this category are meshes with a fixed element relation. There, one mesh has an integer number of elements for each of the elements in the adjacent domain. One important mesh type with such fixed relations between individual parts are hierarchical meshes like octrees. These are also of interest for parallel processing and will be discussed in the latter part of this thesis. Various implementations to solve partial differential equations on this kind of mesh exist. Some examples of these include proposals by Flaherty et

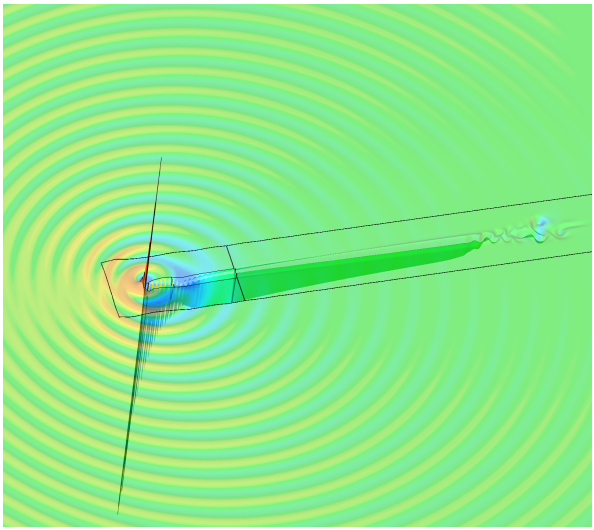


Figure 1.1: Image of an instanteneous pressure field for a sound emitting Kármán vortex street behind a sphere. A large domain is covered to capture the propagation of soundwaves, and around the sphere there are smaller domains, as indicated by the black lines.

al. [17] and Tu et al. [80], application frameworks like Peano [58] and dedicated libraries like Dendro [69].

The Chimera method is a different approach, using overlapping meshes to allow the combination of body-fitted structured meshes with cartesian background grids [76]. An overview on the meshing strategies for this approach is given by Meakin [50]. This coupling method is often used in combination with moving geometries, where the body-fitted meshes can move along with the structures, while the background grid is fixed. The Overture framework [32] implements this kind of coupling for moving geometries and various equation systems.

One step further in the direction of coupling arbitrary domains is taken by tools like *MpCCI* (Mesh-based parallel Code Coupling Interface [37]) developed by Fraunhofer SCAI, where different domains are mostly treated as black boxes with arbitrary surface data that is to be used for the coupling. The *MpCCI* library manages the interpolation and communication between the individual parts and takes care of steering them. A similar tool is offered by *PreCICE* [24], which is also used in *APES* to enable the coupling with third-party solvers. This approach allows specifically the coupling of commercial closed codes to cover different physical aspects in a simulation. However, it is limited to a low order interpolation and often requires a single server to provide the interpolating coupling. Such a single server concept for the coupling interfaces restricts the scalability of the approach as it imposes a bottleneck in the communication between the interacting simulations. Another restriction is the time integration method offered by these tools, which lacks the ability to deal with high-order extrapolations as for example offered by the *ADER* time integration method [79].

1.2 Approach to the Coupling Scheme

The coupling approach deployed in this thesis relies on the observation that all numerical discretizations can be mapped to point values. By taking advantage of this point representation, a coupling at domain interfaces was designed, to allow different domains in arbitrary configurations to exchange values with a high accuracy. Each domain can be discretized independently from the other domains and might make use of unstructured or structured meshes, Finite Volume (FVM), Finite Difference (FDM) or Discontinuous Galerkin (DG) methods. Those numerical schemes will be briefly discussed in Chapter 3. It allows for different equations to be solved in each domain. Thus, the computational effort can be reduced by minimizing the physical phenomena to consider in each domain. This

approach allows us in aero-acoustic simulations to restrict the computation of the full compressible Navier-Stokes equations to a small, highly resolved domain, as viscuous effects are often negligble far away from obstacles. Similarly, nonlinearities can be neglected in the far field region, where only acoustic waves are propagated any further. With a proper decomposition of the physical domain, the equations can therefore be reduced from full Navier-Stokes equations over inviscid, nonlinear Euler equations to linearized Euler equations. These equation systems are briefly introduced in Chapter 2. As the area of application are time resolved phenomena, explicit time integration schemes are used in this work and the individual domains are allowed to advance with their own time-steps. Again, this variance in the time integration allows a minimization of the computational effort.

1.2.1 Serial coupling method for heterogeneous 3D static meshes

In Chapter 3 we will see that boundary conditions for FVM can be implemented by using a set of extra elements outside the actual domain. The coupling method exploits these extra elements to implement the interface between adjacent domains. These elements, also called *ghost cells*, are set to state, such that the numerical scheme fulfills the boundary condition on the surface when using this neighboring information. In the coupling scheme they are filled with neighbor information instead. By this, the *ghost cells* provide an adapter for the interpolation from one mesh to the other. Thus, the coupling is achieved at the interfaces by exchanging the state on both sides as boundary conditions to the mutual neighbor.

A general coupling method based on this principle for three-dimensional problems was developed in [82] by Jens Utzmann. It was specifically designed for multi-scale problems such as aero-acoustics and extends the ideas from Schwartzkopff [73] from two dimensions to three dimensions. While the previous scheme in [73] uses cell intersections to obtain the state in the *ghost cells*, the new approach in [82] deploys discrete points to establish the state within *ghost cells*. The method based on cell intersections computes exact volume fractions of overlapping cells to integrate the corresponding integral mean value in the *ghost cells*. While this is complicated but achievable in two dimensions, the cost for the computation of arbitrary volumina integration becomes prohibitively large in three dimensions. The concept of integral means is also not easily transferable to higher order schemes like the $P_N P_M$, where a more sophisticated internal representation is used in each element, including the *ghost cells*. High-order methods will also be

described in Chapter 3. They usually make use of nodal values, which are chosen at suitable integration points, like the ones for the Gauss-Legendre [48] or Gauss-Chebyshev [8] numerical integration. This nodal basis to construct the state representation in elements, is now also used for the construction of *ghost cells*. Only the state values at those integration points are needed to construct a *ghost cell* with the required accuracy. With this concept it is no longer necessary to deal with complicated volumina and their integration, and it is possible to describe the state in *ghost cells* with arbitrarily high precision. Although the numerical integration with data exclusively from discrete points poses the theoretical risk of aliasing errors, these interfaces proved to work reasonably fine in real world settings.

To account for different scales in time, a subcycling method is used, allowing one domain to perform multiple time step updates during a single time step of another domain. Overall we have a flexible coupling environment that allows the interaction of vastly different solvers. In this work this concept is developed further and adapted to the parallel and distributed computing infrastructure of modern supercomputing facilities.

1.3 Parallel Processing

As the computational effort is drastically increased by the transition from two dimensions to three, a parallelization of the code became necessary. Utzmann [82] used a preliminary parallel implementation, which allowed only relatively small meshes on large numbers of processes. The biggest simulation achieved with that implementation was conducted on 1024 processes and limited to 2 million elements in the unstructured mesh, due to memory constrains. A major issue of the preliminary implementation was also the lacking parallel restart functionality, which did not allow arbitrary restarts from given computed intervals on parallel machines. With the rising frequency of hardware failures in massively scaled super computing facilities and therefore decreasing possible running times for the simulation in a single time block, a parallel restarting facility is an essential feature for future large scale simulations. To enable the restarting with arbitrary number of processes a new file format based on a space-filling curve is introduced for the unstructured solver.

While a major focus of this work is put on the parallelization of the coupling between the fluid domains, the scalable deployment requires the consideration of all involved parts, as any non-scaling step in the simulation will eventually break the simulation. The scalable treatment of the structured solver proved to be not so much of an issue. However, with

the unstructured mesh solver some major bottlenecks are encountered and are overcome in this work as described in Chapter 4. FVM and $P_N P_M$ schemes deployed in the unstructured solver, require the construction of a neighborhood for each element. This so called stencil can span multiple elements for high-orders and overlaps into partitions of neighboring processes. The need for an efficient, parallel construction of this neighborhood led to the development of a dedicated mesh data format, and a suitable preprocessing tool. The tool is designed as a separate application, called *GEUM*, and enables the unstructured solver to run arbitrary large problems on massively parallel systems with little memory per core. While this shifts a scalability bottleneck out of the solver itself, there still remains one in the generation of the unstructured meshes. Those can be generated on special machines with large memory, but eventually this is a point of limited scalability. Therefore, the completely new developed approach in *APES* with a dedicated mesh format and infrastructure involves also the mesh generation itself.

Large scale computations not only require the reduction of the computational effort where possible, but also an efficient exploitation of the computing infrastructure. Especially, when run on massively parallel systems it is important to maximize the serial execution of the application, as any inefficiency gets multiplied with the number of executing processes. Therefore, some strategies on the serial optimization will be briefly discussed in Chapter 5.

The parallel implementation of the coupling scheme itself is discussed in Chapter 6. Limits of the highly general coupling approach of *KOP* in terms of parallel scalability are explored and a concept to overcome such limitations by the design of a new scheme is presented. This new concept builds a framework on its own and is described in the later part of the thesis. It makes use of a unified mesh data structure that will be introduced in Chapter 7 and covers all necessary tools from mesh generation to post-processing. To overcome scalability limits this concept requires a restriction on the mesh data structure. Thus, instead of the completely general approach with arbitrary structured or unstructured meshes, this new framework relies on the known topology of Octrees to enable the distributed and parallel processing. One of the goals by moving to Octree meshes is the aim for runtime adaptations, another is the applicability to a wide range of engineering problems. Therefore, the framework is called *APES* as in **A**daptable **P**oly-**E**ngineering **S**imulations.

As this work is mostly concerned with the parallelization of the discussed coupling mechanism, the question arises how the parallelism is achieved. Large scale parallel systems use distributed memory and the

parallelism has to account for this setup. The de-facto standard for parallel programming on distributed memory systems in high performance computing (HPC) is the Message Passing Interface (MPI) [19]. MPI provides a standardized method to realize communication between processes on independent computing nodes connected by a network. It abstracts the underlying communication mechanisms and exposes interfaces for common communication tasks to the application. The functionality ranges from point to point communications between pairs of processes over collective operations like reductions, involving several or all processes to parallel input and output (IO).

MPI was revised and extended several times and the current version 3 of the standard introduced some interesting new features, especially non-blocking collectives are of interest for this work here. Non-blocking barriers enable the resolution of the termination problem, where it is unclear when to finish a certain task. This arises, when all processes serve some information to all other processes but it is not known in advance who will request information from whom. We will utilize this in the scalable implementation of the coupling exchange.

Another step that can be taken for the coupling of heterogeneous domains is the deployment of specialized hardware for the different domains. Such an interaction between different domains is enabled by *PACX-MPI* [22]. This special approach is covered in Section 6.3. It enables the mapping of a heterogeneous application to a heterogeneous computing environment, which might get even more relevant in the future, when a diversification of computing architectures is expected [6].

Besides the minimization of computational effort by adapted equation systems, discretizations and time integration, an important factor on modern computing architecture is the memory consumption and access. High-order methods allow for accurate solutions with few degrees of freedom and, therefore, little memory consumption. For this reason we will concentrate in this thesis on high-order numerical schemes. Especially, in the new *APES* framework the high-order representation will be the focus after a proper introduction of the mesh infrastructure. In Chapter 8 we investigate how high-order geometries can be represented in the Octree mesh representation of *TreElM*. The method developed in this work to construct these high-order representations is robust and suitable for any complex surface. This is followed by some numerical and scaling considerations in Chapter 9.

Finally, not all the ideas that arose in the course of this work have been realized yet. Therefore, Chapter 10 provides, besides a summary, a glimpse on future work that is still left to be done to advance the presented topic.

2 Considered Equation Systems

The governing equations for the direct aero-acoustic simulation are a set of conservation laws, that allow the description of compressible fluid mechanics in general and simplifications thereof. These equations are briefly revisited in this chapter. The most general description for compressible, viscous flows is provided by the Navier-Stokes equations (*NSE*). In aero-acoustic simulations the following simplifications of this general set of equations, that is expensive to compute can be made for large parts of the computational domain. First, away from obstacles and strong velocity gradients, it usually is legitimate to neglect the viscous terms in the Navier-Stokes equations. This results in the Euler equations (*EE*). Finally, in the propagation of acoustic waves, there are only small deviations from a constant mean flow, that are relevant and the non-linear Euler equations can be linearized around this mean flow in this part of the domain. The linearized Euler equations (*LEE*) offer the largest savings in computational effort, and can be used to cover vast domains for the wave propagation.

Another linear equation system that will be used in this work are Maxwell's equations that govern electrodynamic waves. Electrodynamics is an important application area and in some settings like for example electrodialysis its influence needs to be coupled with fluid dynamic simulations.

2.1 Navier-Stokes Equations

The first conservation law is the conservation of mass and with density denoted as ρ and velocity as $\vec{v}$ it reads:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0. \tag{2.1}$$

The second considered quantity is the momentum $\rho\vec{v}$ and its conservation leads to:

$$\frac{\partial (\rho \vec{v})}{\partial t} + \nabla \cdot (\rho \vec{v} \otimes \vec{v}) + \nabla p = \nabla \cdot \boldsymbol{S}. \tag{2.2}$$

External forces are omitted in the notations here for brevity. The stress tensor is denoted by $\boldsymbol{S}$ and the pressure by p. A dyadic product is indicated by $\otimes$.

Conservation of energy finally results in

$$\frac{\partial e}{\partial t} + \nabla \cdot ((e+p)\vec{v}) = \nabla \cdot (\boldsymbol{S}\vec{v} + \kappa \nabla T), \tag{2.3}$$

where the energy density e summarizes the kinetic and inner energy of the fluid. Thermal conductivity of the fluid is denoted by κ and the temperature by T.

To close the system a description of the fluid material has to be made. The assumption of an ideal gas yields the following relation of p and e:

$$p = (\gamma - 1)\left(e - \frac{\rho\vec{v}\cdot\vec{v}}{2}\right).$$

When assuming a Newtonian fluid, the stress tensor $\boldsymbol{S}$ can be expressed in terms of the velocity gradient by

$$\boldsymbol{S} = \mu_v(\nabla \otimes \vec{v} + (\nabla \otimes \vec{v})^T) + (\lambda \nabla \cdot \vec{v})\boldsymbol{I},$$

where the material parameters dynamic viscosity μ_v and bulk viscosity λ are used to describe the viscosity of the fluid and $\boldsymbol{I}$ is the identity matrix. Often, the bulk viscosity is neglected in numerical models. In this work it is chosen to be $\lambda = \frac{2\mu_v}{3}$, which is a typical assumption in literature [46].

This set of equations are referred to as the Navier-Stokes equations (*NSE*) in the remainder of this work. They can also be written in vector form by collecting the conservative variables into one vector

$$\vec{U} = \begin{pmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ e \end{pmatrix}$$

and defining the flux functions $\vec{F_x}(\vec{U})$, $\vec{F_y}(\vec{U})$ and $\vec{F_z}(\vec{U})$ as follows:

$$\vec{F_x}(\vec{U}) = \begin{pmatrix} \rho v_x \\ \rho v_x^2 + p \\ \rho v_x v_y \\ \rho v_x v_z \\ v_x(e+p) \end{pmatrix}, \vec{F_y}(\vec{U}) = \begin{pmatrix} \rho v_y \\ \rho v_y v_x \\ \rho v_y^2 + p \\ \rho v_y v_z \\ v_y(e+p) \end{pmatrix}, \vec{F_z}(\vec{U}) = \begin{pmatrix} \rho v_z \\ \rho v_z v_x \\ \rho v_z v_y \\ \rho v_z^2 + p \\ v_z(e+p) \end{pmatrix}.$$

The viscous right hand side in its vectorial form can be written with the following vectors $\vec{W}_x$, $\vec{W}_y$ and $\vec{W}_z$:

$$\vec{W}_x(\vec{U}, \nabla \otimes \vec{v}, \frac{\partial T}{\partial x}) = \begin{pmatrix} 0 \\ (2\mu_v + \lambda)\frac{\partial v_x}{\partial x} + \lambda\left(\frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right) \\ \mu_v\left(\frac{\partial v_y}{\partial x} + \frac{\partial v_x}{\partial y}\right) \\ \mu_v\left(\frac{\partial v_z}{\partial x} + \frac{\partial v_x}{\partial z}\right) \\ \Phi_x + \kappa\frac{\partial T}{\partial x} \end{pmatrix}$$

$$\vec{W}_y(\vec{U}, \nabla \otimes \vec{v}, \frac{\partial T}{\partial y}) = \begin{pmatrix} 0 \\ \mu_v\left(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}\right) \\ (2\mu_v + \lambda)\frac{\partial v_y}{\partial y} + \lambda\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_z}{\partial z}\right) \\ \mu_v\left(\frac{\partial v_z}{\partial y} + \frac{\partial v_y}{\partial z}\right) \\ \Phi_y + \kappa\frac{\partial T}{\partial y} \end{pmatrix}$$

$$\vec{W}_z(\vec{U}, \nabla \otimes \vec{v}, \frac{\partial T}{\partial z}) = \begin{pmatrix} 0 \\ \mu_v\left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}\right) \\ \mu_v\left(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}\right) \\ (2\mu_v + \lambda)\frac{\partial v_z}{\partial z} + \lambda\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right) \\ \Phi_z + \kappa\frac{\partial T}{\partial z} \end{pmatrix}$$

The vector $\vec{\Phi}$ is used as an abbreviation for the terms originating from the stress tensor $\boldsymbol{S}$ in the energy conservation law and fully read

$$\begin{aligned} \Phi_x = (2\mu_v + \lambda)\frac{\partial v_x}{\partial x} + \lambda\left(\frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right) \\ + \mu_v\left[v_y\left(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}\right) + v_z\left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}\right)\right] \end{aligned}$$

$$\begin{aligned} \Phi_y = (2\mu_v + \lambda)\frac{\partial v_y}{\partial y} + \lambda\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_z}{\partial z}\right) \\ + \mu_v\left[v_x\left(\frac{\partial v_y}{\partial x} + \frac{\partial v_x}{\partial y}\right) + v_z\left(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}\right)\right] \end{aligned}$$

$$\Phi_z = (2\mu_v + \lambda)\frac{\partial v_z}{\partial z} + \lambda\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right) + \mu_v\left[v_x\left(\frac{\partial v_z}{\partial x} + \frac{\partial v_x}{\partial z}\right) + v_y\left(\frac{\partial v_z}{\partial y} + \frac{\partial v_y}{\partial z}\right)\right].$$

With these vectors the Navier-Stokes equations for Newtonian fluids and ideal gas can be written as

$$\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}_x(\vec{U})}{\partial x} + \frac{\partial \vec{F}_y(\vec{U})}{\partial y} + \frac{\partial \vec{F}_z(\vec{U})}{\partial z} = \frac{\partial \vec{W}_x(\vec{U})}{\partial x} + \frac{\partial \vec{W}_y(\vec{U})}{\partial y} + \frac{\partial \vec{W}_z(\vec{U})}{\partial z}. \tag{2.4}$$

Note, that the dependence of the viscous fluxes on the derivatives have been omitted for brevity. Due to the dissipative terms in the Navier-Stokes equations, they contain hyperbolic and parabolic parts that require different numerical treatments and make the computation complicated for these equations.

2.2 Euler Equations

A simplification of the *NSE* can be obtained by neglecting the dissipative terms on the right hand side of Equations (2.2) and (2.3). The compressible inviscid equations obtained thereby are denoted as Euler equations (*EE*) in this work. They can be obtained in vectorial form from Equation (2.4) by omitting the right hand side, which results in

$$\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}_x(\vec{U})}{\partial x} + \frac{\partial \vec{F}_y(\vec{U})}{\partial y} + \frac{\partial \vec{F}_z(\vec{U})}{\partial z} = 0. \tag{2.5}$$

They are applicable where dissipation can be neglected, e.g. in regions far away from walls and without turbulence. The Euler equations are easier to treat numerically, as they are purely hyperbolic. Solving them instead of the full compressible *NSE* reduces the computational effort considerably. However, their nonlinearity still imposes a relatively high computational cost that is unnecessary for pure wave propagations.

2.3 Linearized Euler Equations

If there are only small changes in the state variables and the nonlinearity in the *EE* can be neglected, the linearized Euler equations (*LEE*) can be used in the numerical model. This is for example the case for sound wave propagation away from the sound generating flow [3].

For the linearization of the Euler equations, their non-conservative formulation

$$\frac{\partial \vec{P}}{\partial t} + \boldsymbol{N_x}(\vec{P})\frac{\partial \vec{P}}{\partial x} + \boldsymbol{N_y}(\vec{P})\frac{\partial \vec{P}}{\partial y} + \boldsymbol{N_z}(\vec{P})\frac{\partial \vec{P}}{\partial z} = 0 \tag{2.6}$$

can be used, and the matrices $\boldsymbol{N_x}$, $\boldsymbol{N_y}$ and $\boldsymbol{N_z}$, can be directly evaluated with a fixed mean state. The primitive variables $\vec{P}$ are given by:

$$\vec{P} = \begin{pmatrix} \rho \\ v_x \\ v_y \\ v_z \\ p \end{pmatrix}$$

and the matrices are defined accordingly as:

$$\boldsymbol{N_x}(\vec{P}) = \begin{pmatrix} v_x & \rho & 0 & 0 & 0 \\ 0 & v_x & 0 & 0 & \frac{1}{\rho} \\ 0 & 0 & v_x & 0 & 0 \\ 0 & 0 & 0 & v_x & 0 \\ 0 & \gamma p & 0 & 0 & v_x \end{pmatrix},$$

$$\boldsymbol{N_y}(\vec{P}) = \begin{pmatrix} v_y & 0 & \rho & 0 & 0 \\ 0 & v_y & 0 & 0 & 0 \\ 0 & 0 & v_y & 0 & \frac{1}{\rho} \\ 0 & 0 & 0 & v_y & 0 \\ 0 & 0 & \gamma p & 0 & v_y \end{pmatrix},$$

$$\boldsymbol{N_z}(\vec{P}) = \begin{pmatrix} v_z & 0 & 0 & \rho & 0 \\ 0 & v_z & 0 & 0 & 0 \\ 0 & 0 & v_z & 0 & 0 \\ 0 & 0 & 0 & v_z & \frac{1}{\rho} \\ 0 & 0 & 0 & \gamma p & v_z \end{pmatrix}.$$

In the linearization a fixed state $\vec{P}_0$ is chosen and only perturbations $\vec{P}_\delta$ of this state are considered as variables in the equation system. Neglecting quadratic terms of the perturbations, we obtain the linear system with constant matrices $\boldsymbol{N_x}$, $\boldsymbol{N_y}$ and $\boldsymbol{N_z}$ as

$$\frac{\partial \vec{P}_\delta}{\partial t} + \boldsymbol{N_x}(\vec{P}_0)\frac{\partial(\vec{P}_\delta)}{\partial x} + \boldsymbol{N_y}(\vec{P}_0)\frac{\partial(\vec{P}_\delta)}{\partial y} + \boldsymbol{N_z}(\vec{P}_0)\frac{\partial(\vec{P}_\delta)}{\partial z} = 0. \tag{2.7}$$

Note that for the linearized Euler equations the notation in (2.7) is also the conservative form, due to the constant matrices.

2.4 Maxwell's Equations

Electrodynamics is governed by Maxwell's equations and with an isotropic, linear material they read:

$$\nabla \cdot (\epsilon \vec{E}) = \rho^e \tag{2.8}$$

$$\nabla \cdot \vec{B} = 0 \tag{2.9}$$

$$\frac{\partial \vec{B}}{\partial t} + \nabla \times \vec{E} = 0 \tag{2.10}$$

$$\frac{\partial \epsilon \vec{E}}{\partial t} - \nabla \times (\vec{B}/\mu) = -\vec{j} \tag{2.11}$$

Gauss's law (2.8) states a direct relation between the divergence of the electrical field $\vec{E}$ and the electrical charge density ρ^e. Similarly, the magnetic field $\vec{B}$ has to be divergence free as there are no magnetic charges, which is expressed in (2.9). The two fields evolve in time according to Faraday's (2.10) and Ampère's law (2.11). In these equations, the environment is described by permittivity ϵ and permeability μ of present materials. Note that there is no time-dependency in (2.9) and, if there is no variation in ρ^e over time also not in (2.8). They express a purely spatial relation and should be satisfied for all times if they are satisfied at one point in time. However, this is numerically not necessarily true and it might be necessary to apply some convergence correction in the numerical simulation [57].

2.5 Review and Relevance of the Considered Equations

The presented equations (*NSE*, *EE* and *LEE*) implement different models for fluid motion. They show that even if only fluid motion is considered, vastly different equations can be applied. Their solution requires adapted numerical schemes, resulting in a heterogeneous simulation domain with very different computational properties. A coupling framework as presented here, allows the combination of the different models and thereby the reduction of computational costs where possible.

The Navier-Stokes equations involve parabolic parts, which are notoriously expensive to handle in explicit time integration schemes, due to their strong stability constraints on the time step. With high order approximations, this limitation gets even more severe. Together with their nonlinearity, this results in expensive computations for domains solving

the *NSE*. Therefore, it is desirable to switch the equations to solve for as soon as possible to the *EE* and minimize the area in which the *NSE* are deployed. While the *EE* are still nonlinear, they are purely hyperbolic and can be treated more easily by according numerical schemes. Linearization finally, offers the largest savings in computational effort, but is sufficient to model the propagation of acoustic waves. Thus, the *LEE* are suitable to cover the large domains of acoustic wave propagation.

In addition to fluid dynamics, also electrodynamics are considered. A coupling of electrodynamics and fluid flows is necessary in applications like electrodialysis. Maxwell's equations covering electrodynamics are linear equations for linear, homogeneous, isotrop materials. However, we will use them in a setting with discontinuities in the material distribution to illustrate the high-order handling of such geometries.

All considered equations are conservation laws and the numerical schemes presented in the next chapter are especially designed for this kind of equations.

3 Deployed Numerical Methods and Their Parallelization

After introducing the fundamental equations in the previous chapter, we now briefly discuss the numerical methods that will be used throughout this work to solve them. All previously described equation systems are time dependent, and a direct simulation requires their time resolved computation. To solve them, Rothe's method [38] can be used to treat time and space discretizations seperately from each other. We will first briefly discuss the deployed time integration methods and then move on to spatial discretization schemes in this chapter.

3.1 Time Integration

By Rothe's method, the partial differential equation can be split into an ordinary differential equation in time and a partial differential equation in space. Therefore, an ordinary differential equation integrator is required to solve the time evolution. A vast variety of numerical ordinary differential equation solvers are available today [5] and the most appropriate one can be chosen for each purpose. One important classification that can be made for numerical ordinary differential equation solvers is the division into implicit and explicit schemes. Implicit schemes require the solution of a linear equation system in each iteration but are usually unconditionally stable. Explicit schemes on the other hand are only conditionally stable and generally require a restriction on the step size but do not require the solution of a linear equation system [9]. However, in transient simulations, a high temporal resolution is anyway required and this limitation is not as severe, as it might appear at first. In explicit time integration each time step solely depends on the previous one. This is also highly attractive for the coupling scheme, where data from neighboring domains needs only to be retrieved for older time steps. Therefore, we will use explicit time integration methods in the numerical frameworks here.

A popular explicit time integration method is offered by the family of explicit Runge-Kutta methods [45], which make use of multiple stages within each time step to achieve a high-order time integration [86]. The classical

fourth-order Runge-Kutta method is used in *APES*. Each computation of the individual stages require the evaluation of the spatial discretization scheme but otherwise no additional computational costs are attached to this method. Thus, it provides an efficient and straight-forward method for the time integration.

The coupling method, however, requires the state values at arbitrary pionts in time, not just at discrete stages as offered by the classical Runge-Kutta method. This could be overcome by using a so-called continuous Runge-Kutta method, however they still suffer from another drawback of the Runge-Kutta family of methods. For high-order schemes in time, the Runge-Kutta methods are exhibiting an increasingly high cost beyond the fifth order [61]. The phenomenon, commonly referred to as Butcher barrier, requires the method to deploy an more than proportional growing number of stages to achieve a given error convergence order. Due to this effect, Runge-Kutta methods become overly expensive for very high orders. Thus, instead of a continuous Runge-Kutta method, another approach is chosen in the coupling application *KOP* to obtain a high-order time integration. This approach, the *ADER* (Arbitrary high-order using DERivatives) method [74] makes use of a Taylor expansion to represent the time-evolution of the state. The Taylor expansion adds the need for the computation of temporal derivatives to the numerical scheme but overcomes the order barrier in a one-step method. Temporal derivatives can be obtained from spatial derivatives via the Cauchy-Kowalevsky procedure [44]. Spatial derivatives on the other hand become available by a high-order spatial discretization. Therefore, the *ADER* strategy provides a method to arrive at a high-order numerical method in space and time.

3.2 Cartesian Structured Meshes

Cartesian structured meshes are the simplest meshes to descritize a given volume. They offer also the most efficient access to neighboring elements in stencil based methods like those discussed in this chapter. However, they are very restrictive and do not allow an adaptation to geometrical constraints of the computational domain. An option to allow for maintaining the structured nature in solvers but simulating more complex geometries is offered by body-fitted meshes [78]. The generation of body-fitted meshes is much more complicated in three dimensions than in two. They also have the additional cost of transformations between reference coordinates and physical coordinates attached to them. In an aero-acoustic simulation however, we are facing large volumes of space without any obstacles.

For these domains cartesian structured meshes are well suited and their advantages can be fully exploited. The *KOP* approach enables the combination of structured and unstructured domains in a single simulation and a dedicated, parallel solver is available in this context. All numerical schemes presented in the following sections are also suitable for unstructured meshes but specialized implementations for structured meshes allow for a reduction in the computational effort. Thus, one of the goals in the coupling strategy is the maximization of the area that is computed with the structured grid. The structured solver is an important building block in the coupled simulation but its parallelization is straight-forward. In contrast to the processing of unstructured meshes, there are no scalability limits imposed by this part of the application. This is due to the globally known topology of the mesh.

3.3 Method of Finite Volumes

To treat the equations numerical in space, there are various numerical methods available. With compressible flows, shocks might appear in the solution, resulting in local discontinuities. Specifically to this end, Finite Volume Methods (*FVM*) have been developed. They exploit the fact that conservation laws are solved and discretizes the domain into small volumes that interact with each other. Due to the nature of conservation laws, the integral mean of conserved quantities in each volume changes only as much over time, as there are quantities flowing over the surface of the volume. This can be seen by considering the integral form of conservation equations like (2.4). By integrating over a finite time interval from t^n to t^{n+1} and a finite volume Ω_i, we obtain the weak formulation

$$\int_{t^n}^{t^{n+1}} \int_{\Omega_i} \frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{F}_x(\vec{U})}{\partial x} + \frac{\partial \vec{F}_y(\vec{U})}{\partial y} + \frac{\partial \vec{F}_z(\vec{U})}{\partial z} \mathrm{d}V\mathrm{d}t = \int_{t^n}^{t^{n+1}} \int_{\Omega_i} \frac{\partial \vec{D}_x(\vec{U})}{\partial x} + \frac{\partial \vec{D}_y(\vec{U})}{\partial y} + \frac{\partial \vec{D}_z(\vec{U})}{\partial z} \mathrm{d}V\mathrm{d}t. \quad (3.1)$$

Let us shorten the notation of (3.1) and use $\nabla\cdot$ to express the scalar product with the gradient. Assume also some generic flux $\vec{f}$, covering all conservation laws, and combining $\vec{F}$ and $\vec{D}$ in the case of the *NSE*. With these abbreviations the following general weak form for a conservation law

can be written as:

$$\int_{t^n}^{t^{n+1}} \int_{\Omega_i} \frac{\partial \vec{U}}{\partial t} + \nabla \cdot \vec{f}(\vec{U}) \mathrm{d}V \mathrm{d}t = 0. \tag{3.2}$$

We now introduce the integral mean value in each cell i

$$\overline{\vec{U}_i} = \frac{1}{|\Omega_i|} \int_{\Omega_i} \vec{U} \mathrm{d}V. \tag{3.3}$$

The idea of *FVM* is now to describe the evolution of the overall solution by the evolution of these integral mean values $\vec{U}_i$. By employing the divergence theorem and plugging the definition of the integral mean (3.3) into (3.2), we obtain the semi-discrete evolution equation

$$\overline{\vec{U}_i}^{n+1} = \overline{\vec{U}_i}^{n} - \frac{1}{|\Omega_i|} \int_{t^n}^{t^{n+1}} \int_{\partial\Omega_i} \vec{f}(\vec{U}) \vec{n} \mathrm{d}S \mathrm{d}t. \tag{3.4}$$

Here, $\partial\Omega_i$ denotes the surface of the volume Ω_i, and $\vec{n}$ the normal vector on this surface. Equation 3.4 is an exact evolution for the integral mean $\overline{\vec{U}_i}$ but it requires the exact state $\vec{U}$ at the surface $\partial\Omega_i$. Only the time evolution for the integral means are available. Thus, the exact distribution $\vec{U}(\vec{x}, t)$ is not known in the numerical solution. Instead, we need to introduce an approximation for the flux on the surface that only depends on the integral mean values. As this is the main approximation in the *FVM* the choice of a numerical flux is important for the overall quality of the solution.

With an appropriate numerical flux $\vec{g}$ that only depends on the state left and right of the face, we then get the approximated time evolution for the integral mean values

$$\overline{\vec{U}_i}^{n+1} = \overline{\vec{U}_i}^{n} - \frac{1}{|\Omega_i|} \int_{t^n}^{t^{n+1}} \int_{\partial\Omega_i} \vec{g}(\overline{\vec{U}_i}, \overline{\vec{U}_j}) \mathrm{d}S \mathrm{d}t. \tag{3.5}$$

Where $\overline{\vec{U}_j}$ is the integral mean value in the cell adjacent to the surface of cell i. The time integration in (3.5) can now be solved by a time integration method from the previous section.

The integral mean offers only a rough estimation of the state within the cells and only a first order error convergence is obtained with this approximation. A first order *FVM* scheme in space and time is obtained by using a simple forward Euler time discretization with constant fluxes on the interfaces for (3.5). Though, such a first order *FVM* solver is straight

forward to implement and highly stable, it introduces a strong numerical damping and an extremely large number of cells time-steps is required to obtain sufficiently small errors for time resolved wave phenomena. This is especially a severe problem in multi-scale settings, where the time period that needs to be covered for the most resolved domain is likely to be so long, that the damping effects of the numerical scheme will dominate the solution.

More accurate approximations can be obtained by a so-called reconstruction. For a reconstruction, multiple cells are considered and an interpolating polynomial is computed that recovers the integral mean in each cell. All cells considered in this reconstruction compose the stencil of the scheme. The higher the degree of the polynomial, and the order of the numerical scheme, the more cells are required for the stencil. With the reconstruction, the value at the cell surface can be more accurately approximated. The reconstructed state representation also enables the computation of derivatives for the *ADER* time integration and therefore a scheme with the same order in space and time. As the direct neighbors are anyway required to compute the fluxes, it is quite natural to deploy at least those for the reconstruction of states within cells. Further cells will increase the communication effort, as an increased neighborhood is required. Another point to consider is the tendency of higher order representations to oscillate near discontinuities, which might destroy the solution. To alleviate this shock-capturing schemes can be deployed. The method of choice in this work is the usage of Weighted Essentially Non-Oscillatory (WENO) reconstruction proposed by Osher et al. [60]. For this reconstruction, multiple stencils need to be evaluated and in smooth regions, their result can be combined to form a larger high order stencil approximation, but in the proximity of discontinuities, those stencils containing the oscillations are discarded.

The stencil cells also lead to a simple idea for the implementation of boundary conditions. At the border of the computational domain boundary conditions can be prescribed by filling virtual cells outside the domain with values such, that exactly on the boundaries, the boundary conditions hold true, when those cells are used as regular neighboring cells during the usual computation. Such cells are not considered for time step updates in the computation and are referred to as ghost cells. The ghost cells are filled according to the boundary condition and the state in the adjacent cell before each time step. By deploying this concept, no cell inside the domain needs to be treated specially.

A similar concept can be deployed for the parallelization, where the overall mesh is partitioned into as many parts as there are processes. To

allow the usual computation, a set of halo cells surronds those partitions to hold the data from the remote cells. These halo cells need to be communicated in each time step and are not considered in the computation.

3.4 Discontinuous Galerkin Finite Element Method

The Discontinuous Galerkin Finite Element Method (DG) extends the concept of the FVM method by replacing the integral mean value (3.3) by function series to represent the state within the element with greater detail. Thus, the state in a given element i is approximated by

$$\vec{U}_i(\vec{x}) = \sum_{j=0}^{m-1} \vec{a}_j \phi_j(\vec{x}). \tag{3.6}$$

The functions ϕ_j might be chosen freely but a common choice is a polynomial basis. With this approach, the single degree of freedom in each element is replaced by m degrees of freedom $\vec{a}_j$, resulting in a high-order approximation of the state for large m. Similarly to FVM we now consider the integral over the volume of the element and apply the Galerkin approach with the multiplication by a test function $\psi_k(\vec{x})$, which yields

$$\int_{\Omega_i} \frac{\partial \vec{U}_i}{\partial t} \cdot \psi_k + \nabla \cdot \vec{f}(\vec{U}_i) \cdot \psi_k \mathrm{d}V = 0. \tag{3.7}$$

Integration by parts then results in

$$\int_{\Omega_i} \frac{\partial \vec{U}_i}{\partial t} \cdot \psi_k \mathrm{d}V + \oint_{\partial\Omega_i} + \left(\vec{f}(\vec{U}_i) \cdot \vec{n}\right) \psi_k \mathrm{d}S - \int_{\Omega_i} \vec{f}(\vec{U}_i) \cdot \nabla \psi_k \mathrm{d}V = 0. \tag{3.8}$$

As in the FVM the state on the interface $\partial\Omega_i$ is allowed to jump, and the exact state on the interface is not known. Thus, a numerical approximation for this flux has to be taken, and we replace the exact flux by a numerical flux $\vec{g}$. We need m test functions ψ_k to obtain a fully determined system of equations in this approach. Usually, the test functions are chosen from the same function space as the ansatz functions ϕ. Though, the choice of function spaces is mathematically not relevant for the method to work, it numerically has a huge impact. We will use polynomial functions here for both, the representation of the solution $\vec{U}_i$ and the test functions ψ_k.

3.5 The $P_N P_M$ Scheme

The $P_N P_M$ scheme with WENO reconstruction was introduced by Dumbser et al. for compressible flows [13]. It offers a generalized discretization, combining the element local representation of the *DG* with the reconstruction from *FVM*. This scheme uses a polynomial representation inside elements with a polynomial degree of N, just like the *DG* outlined above. Note that we used m to denote the number of degrees of freedom in (3.6). In one dimension the polynomial degree is given by $N = m - 1$, for multiple dimensions the number of degrees of freedom depend on how the polynomial space is constructed but the same logic applies and we have a certain given maximal polynomial degree in our state representation. However, in $P_N P_M$ we now extend the representation by a reconstruction mechanism like WENO. Thus, the scheme yields an increased accuracy with a representation of the state by a polynomial of degree $M \geq N$. The reconstructed polynomial basis Φ is orthogonal and chosen to coincide with the *DG* basis ϕ. This results in the higher order basis functions Φ_i for $i > N$ all being orthogonal to all ϕ. The $P_N P_M$ scheme unifies the *FVM* and the *DG* in the sense that either one is just a special case of the $P_N P_M$. Choosing $N = 0$ results in the classical *FVM*, where high-order approximations are purely achieved by reconstruction. While choosing $N = M$ yields a *DG* solver, where no reconstruction is applied at all. A main advantage of the $P_N P_M$ scheme is its flexibility to apply reconstructions in addition to the *DG* representation and thereby gain the ability to deal with shocks efficiently. However, a drawback of the reconstruction is the need to communicate larger stencils and we, therefore, generally want to keep the reconstruction at a minimum.

4 Scalable Unstructured Solver

For general applications of the flow solver it is important to allow the discretization of arbitrarily shaped geometries like a nozzle as indicated in Figure 4.1. This is generally achieved by unstructured meshes, where the complete geometrical information is explicitly available for each element. Obviously such a mesh description imposes a severe overhead and introduces additional implementation complexity when compared to structured meshes, where the topology is implicitly known. This additional complexity decreases the achievable by introduced overheads like indirect memory accesses and additional book-keeping. A benefit of the coupled approach is the possibility to minimize the space that needs to be covered by unstructured meshes. However, the spatial resolution usually has to be quite high in the proximity of obstacles to resolve small scale phenomena close to the walls. Thus there are typically many elements in the unstructured domain, even with a restricted volume. Due to the high resolution, the induced time steps to fulfill the stability criterion are usually small compared to other domains in the simulation. Therefore the domain might easily have to do an order of magnitude more iterations in time than its neighboring domains. Given these factors, the unstructured part contributes a large share to the overall computational costs of a typical aero-acoustic coupled simulation. This part should thus be the one, which has to be distributed across the largest share of computing resources. For this reason, its scalability is a major concern in the overall design of the coupled parallel solver.

This chapter discusses this crucial component of the overall simulation. Its main focus is the scalable implementation of the deployed numerical scheme on unstructured meshes. As modern supercomputing systems are all built up by more and more distributed resources, the scalability, especially with respect to memory consumption is outlined. Most parallelization concepts described here are generally applicable to discretizations based on unstructured meshes and not limited to the specific numerical scheme.

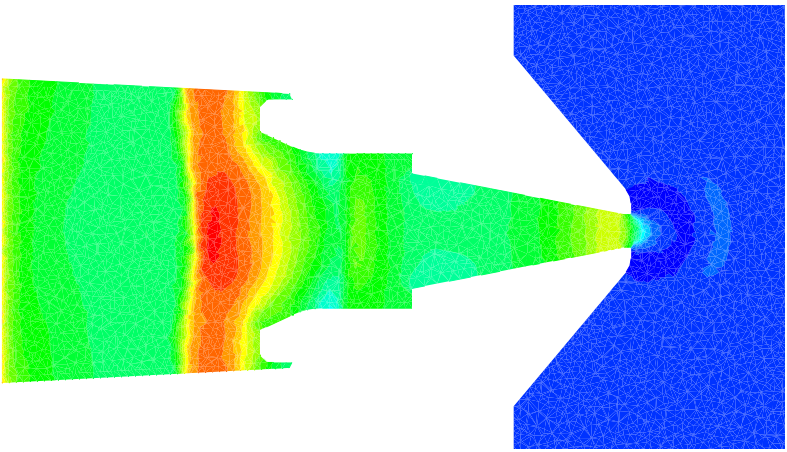


Figure 4.1: Illustration of a complex geometry, represented by an unstructured mesh. Shown is the cut through a supersonic nozzle, with an instanteneous density field. The mesh is indicated by the brighter lines.

4.1 Requirements by the Numerical Scheme

The unstructured solver utilizes the $P_N P_M$ method. As described in Section 3.5, this method is a generalization of the Finite Volume method and inherits many of its properties. There are little requirements by the scheme on the mesh itself and it is well suited for the deployment on unstructured meshes. However, one major task needs to be solved. Namely the stencil for each element needs to be generated. Though the stencil can be greatly reduced by the internal polynomial representation, a proper treatment of shocks with the WENO reconstruction requires a sufficiently large stencil. This implies the need for neighbors of neighbors, which can not be easily found in the distributed unstructured mesh.

In the following we discuss, how the stencil identification can be implemented on massively parallel computing systems, where the mesh is distributed across many partitions. We will see that a special mesh format is required for a fully distributed handling of the mesh by the solver and we develop a parallel method to find arbitrary stencils in unstructured, partitioned meshes.

4.2 Distributed Mesh Handling

While the global data required to describe structured meshes is minimal and is easily stored for each process, this is not true for unstructured meshes. Unstructured meshes have to be described explicitly in their geometrical and topological properties. The required memory to represent the mesh is therefore not negligible, and the storage of the complete mesh data on each and every process is not feasible for large-scale simulations. Prior to the new concept introduced here, the non-trivial tasks of neighbor definition and partitioning were achieved by the duplication of the complete mesh data on each process, limiting the scalability drastically. Figure 4.9 illustrates the limiting memory consumption by the global mesh information in each process. Instead, the mesh has to be distributed across all processes, but the mesh operations, which might require arbitrary data throughout the domains then have to be done on the distributed mesh. The format described in the following sections addresses this problem and provides a definition of the mesh, which allows for the efficient lookup of mesh data on remote partitions.

The need for a completely distributed computation to avoid memory bottlenecks dictates a scalable mesh handling right from reading it from disk onwards. To achieve this scalable reading, we introduce a mesh format that allows simple, distributed reading of the mesh in parallel simulations with an arbitrary number of processes. This mesh format uses a space-filling curve ordering [2] for its elements, which provides a serialization for the elements while preserving locality to the largest part. That is, elements close by in the serialized one-dimensional list are also close to each other in the three-dimensional space. Figure 4.2 illustrates the Z-curve ordering in two dimensions. Please observe in this figure how points that are close to each other in the two dimensional mesh, generally are also close to each other in the one-dimensional ordering. The locality of the higher dimensional space is thus kept to some extent. Other space-filling curves offer an even better locality. The Hilbert space-filling curve for example avoids the jumps that can be seen in the Z-curve ordering [53]. Due to these properties of the space-filling curve, a reasonable partitioning is achieved by cutting the list of elements into chunks of equal size, allowing each process to read its part without additional communication or data. An analysis on the quality of partitions based on space-filling curves is given in [93]. A comparison to graph partitioning methods that allow for an optimization with respect to the communication surfaces is provided in [70].

Figure 4.3 illustrates an unstructured mesh on top of a spatial ordering

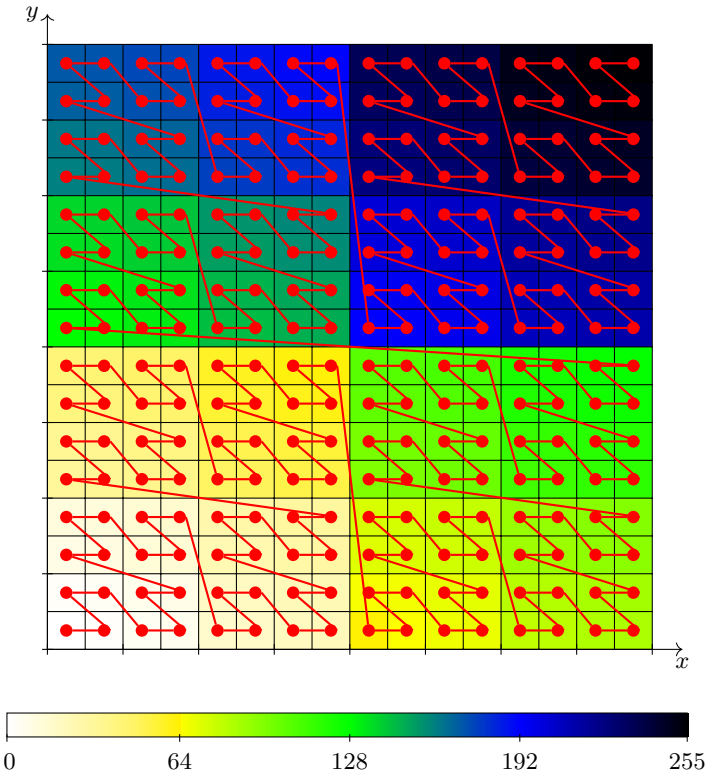


Figure 4.2: Illustration of Z-ordering space-filling curve. The ordering along the (red) curve is given by the coloring. A finite iteration of the space-filling curve with an ordering from 0 to 255 is shown.

by a space-filling curve. To assign an ordering value for each element, its barycenter can be mapped to a space-filling curve that covers at least the complete mesh. All elements then are ordered accordingly and the resulting mesh can then easily be partitioned by cutting the linear element list into equally sized contiguous chunks. The resulting ordering of elements in the unstructured mesh is shown in Figure 4.4. Here the color from the barycenter is taken for the complete element and the position of each element in the overall mesh is indicated by the numbering from 1 to 23.

For simplicity, this format is abbreviated as *GEUM* for *G*eneral *E*lementwise *U*nstructured *M*esh in the following text [40]. After the description of the mesh format and its parallel processing, a distributed algorithm is presented for the neighborhood search. This neighborhood search was the major obstacle for the fully distributed simulation of unstructured meshes up to this new implementation. With these two strategies, large-scale simulations on unstructured meshes are enabled, and the unstructured solver can utilize highly distributed computing systems in parallel executions.

4.2.1 *GEUM* format description

The main goal of the *GEUM* format is to efficiently process the mesh data stored on disk in a parallel simulation run. To achieve this, the format uses a plain, easily accessible structure. A mesh is constructed by a set of vertices in space, a set of elements, describing the connections between the different vertices and a set of boundary conditions. While the boundary conditions are not strictly necessary to describe the mesh itself, they are usually required to run any meaningful simulation on it. An element is described by the vertices it is connected to and a set of sides, describing the surface of the element with an ordered subset from the vertices of the element. General header data is stored in an ASCII-formatted file, mainly describing the layout of the actual mesh data, stored in binary format. This header file starts with a summary block, where some overall specifications are declared:

1. total number of elements
2. total number of vertices
3. number of different element types

As the format is supposed to be as general as possible and capable to describe arbitrary polyhedral elements, each element type has also to be described in the header. For each type of element the following specifications are made:

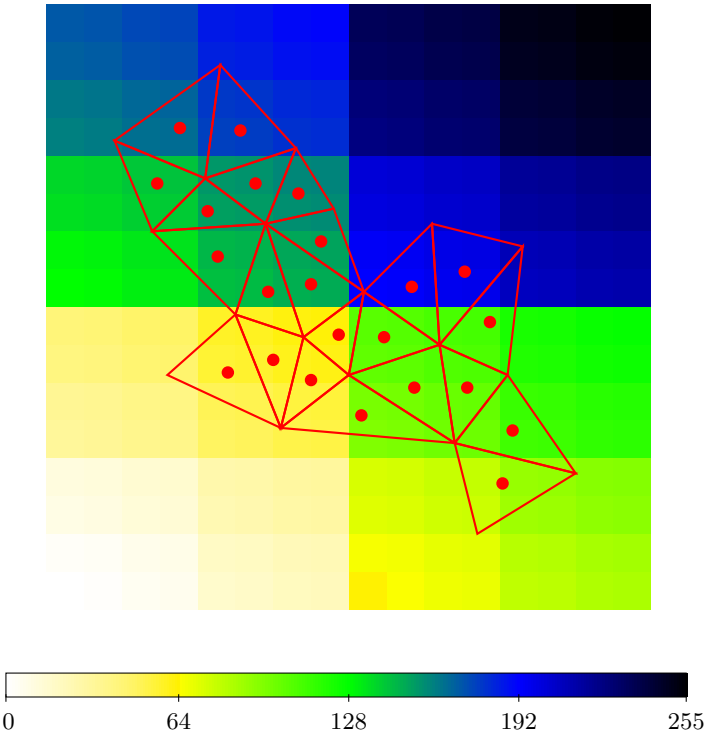


Figure 4.3: An unstructured triangle mesh shimmed with the space-filling curve from Figure 4.2. Each element is assigned its ordering value according to its barycenter location. The unstructured mesh and the barycenters of elements are drawn in red, while the coloring in the background indicates the ordering by the Z-ordering.

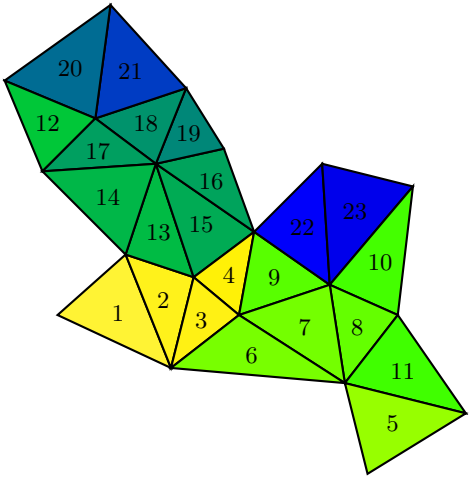


Figure 4.4: Resulting ordering of elements in the unstructured mesh of Figure 4.3 after applying the space-filling curve sorting. The colors indicate the space-filling curve value for each element, as obtained by the barycenter. The numbers indicate the ranking of the elements.

1. number of elements of this type
2. number of vertices, the element is connected to, for tetrahedrons this would for example be 4.
3. number of sides describing the surface of the element, for hexahedrons this would be for example 6.
4. A list of offsets, with as many entries, as there are sides for the element. These describe the start of each side in the following list of element vertices.
5. A list of element vertices, with a length given by the last entry in the list of offsets above.
6. number of binary files
7. for each binary file:
   - name of the binary file
   - offset, if more than one file is used.

The element sides are described in compressed sparse row (CSR) format, where an offset list provides the starting point for each side in the serialized array of element vertices.

Consider for example a tetrahedron as shown in Figure 4.5. A tetrahedron has four vertices and four triangle sides, each side connected to three vertices. In the example the vertices are numbered 1 to 4. The vertices within each side definition are ordered such, that the normals all point outwards when following the right hand rule. We describe the tetrahedron now by storing the vertex indices for each side. For the four triangles of the tetrahedron in the example these are:

- Side A: 1, 2, 3
- Side B: 1, 4, 2
- Side C: 1, 3, 4
- Side D: 2, 4, 3

A single one-dimensional array, as illustrated in Figure 4.6 stores the vertices for all sides. Now we need a list of offsets to distinguish individual sides. That is, we need a list with the length given by the number of sides of the element to point to the index of the last vertex for that side. In the

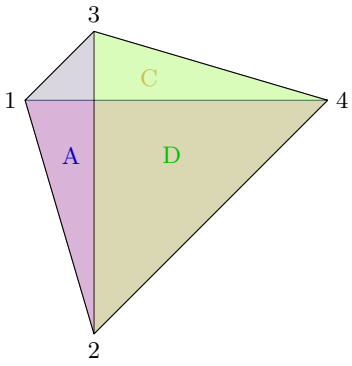


Figure 4.5: Example tetrahedron with the vertices 1 to 4. Sides are indicated by colors and labeled with letters at their barycenters. Only sides A and D are in the foreground. Labeling for side B is left out, to avoid confusion.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	1	4	2	1	3	4	2	4	3

Figure 4.6: Linear array of vertices for all sides of the tetrahedron in Figure 4.5. The numbers above the array elements indicate their respective indices.

example of the tetrahedron, this list for the four sides results in the offset list $\omega =$ | 3 | 6 | 9 | 12 | where each entry refers to the index in Figure 4.6. The offset for the first side is always 0 and does not have to be stored, while the last entry immediately provides the length of the array for the side vertices. Vertex positions for a given side σ are now found at the indices $\omega(\sigma - 1) + 1$ to $\omega(\sigma)$ with the not stored $\omega(0) = 0$. For example, the third side (C) in Figure 4.5 has, according to the offset list, the indices from 7 to 9 in the list of vertices in Figure 4.6.

With this layout of the header it is simple to parse the information in the application and create data structures during runtime as they are required. After reading the overall header, an array for all element types can be allocated. Then we can proceed to read the data for each element type. First, the number of sides for the current element type, which allows us to allocate the list of offsets is read. Then, after filling this list of offsets,

we know how long the vertex list for all sides has to be, and we can allocate it accordingly. Finally, we can fill this list and read the vertex indices for all sides into that list. The actual data for each element type is stored in a file of its own, resulting in files, which are completely uniform within themselves. Additionally, the data might be distributed across several files to ease parallel processing. With this strategy it is for example possible to have one data output file for each available IO node. To allow the reading of the data of such data spread across multiple files, the offset of each block of data in the overall data set is stored along with the filenames in the header file. In most cases a single data file for each element type is sufficient, though. In this case, the offset can be omitted, as it is known to be 0. Finally the header is completed with information on the vertex data. Again multiple files might be used to store the physical coordinates of the vertices. The first data about the vertices is the number of files used to store them. This number is followed by a list of filenames, and if there is more than one, a list of offsets indicating the position of each of them in the global set of vertices.

In addition to this header file, there is an additional ASCII file describing the boundary conditions. Boundary conditions are simply identified by a describing label and their order. The boundary description file therefore contains first the number of boundary conditions, followed by a list of labels. A connection of the boundary conditions to the appropiate surfaces in the mesh is directly stored in the elementwise data structure as described below.

The binary data file for a given element type contains the global vertex positions to which each element is connected, followed by the neighbor identification on each side. A description of the relation between vertices and sides for the element type is given in the header above. This results in a block of signed 8 byte integers with a length of the number of vertices and the number of sides in this element type. That means, a tetrahedral element is represented by $(4 + 4) \cdot 8 = 64$ bytes. As all elements of the same type possess the same number of vertices and sides, the position of data to be read for a given element can be computed independently of all other elements. Furthermore, the uniform shape of the data stored in the files, allows for straight forward conversions from big-endian to little-endian ensuring the portability of mesh data across different computing architectures. The global numbering of elements and vertices starts with 1. The complete non-positive range of the signed integer can therefore be used to indicate not existing elements at domain boundaries. With this representation it is possible to address 10^{18} elements and as many vertices and different boundary conditions.

If there are multiple element types in the mesh, an additional file is used to achieve an overall sorting of all elements. This mapping file contains for each global element the element type and the position of the element within the elements of this type. These are two signed integers with 8 bytes for each element. For uniform meshes with just one type of elements, this file is not required. The data describing the mesh is completed by the file containing the vertices in the mesh. These vertex coordinates provide the full geometrical description of the mesh, while all the other data describe the topology of the mesh. Only in this file there are double precision numbers to be found, however each number occupies again 8 bytes, allowing the same endianness conversion as for the rest of the binary data. Each vertex is described by a consecutive block of three values, thus a given vertex is easily found in the file.

The described format purely relies on files in a given directory for largest portability and highest utilization of the efficiency provided by the MPI implementation. However, at the cost of an additional layer in the IO, the described format can also be put into a container like HDF5 [77].

All mesh data in *GEUM* is available in a form that enables completely distributed reading by independent processes. In other available mesh formats this is usually not possible, due to the lack of explicit neighbor information. The presented mesh format shifts the burden of the neighborhood search from the solver to the pre-processing step, removing one obstacle to scalability from the solver. Note however, that only direct adjacency information is stored, stencil neighborhoods with non-direct neighbors still need to be constructed at runtime by the solver. This construction will be discussed in Section 4.3.

4.2.2 Parallel processing of *GEUM* data

As outlined in the format, all the connectivity information is stored in the file on a per element basis. Thus, for a given element all the neighboring elements are known, after reading its record. In combination with a space-filling curve partitioning this allows for an efficient parallel handling of meshes [28]. The space-filling curve provides a meaningful sorting criteria for all elements, allowing a regular distribution of elements along the curve into locally computable partitions. Figure 4.4 gives an example of the ordering of an unstructured mesh with the help of a space-filling curve. Since each partition can compute not only the splitting positions for its own set of elements as well as those for all other partitions. Because each partition receives a contiguous chunk of elements, the hosting partition for all global element indices are always known without any further communication.

If the basic partitioning by the space-filling curve is deemed insufficient for the actual computation, it is still possible to redistribute the elements with a better approach after the initial distributed reading from disk. However, with this proposed concept it is possible to read the mesh completely in parallel, where each task only needs to store the data of its own local part, resulting in a fully scalable mechanism to load unstructured meshes.

The address of an element in parallel computations is built by a tuple of integers given by the rank of the task processing it, and the local position in the list of elements within that partition. With the simple rule for the partitioning along the space-filling curve, these adresses can be found without communication, as the global position of an element in the mesh can be directly translated. What remains to be defined is the method to describe the splitting positions for all partitions along the space-filling curve. For a balanced work on the mesh, each process should get the same number of elements. Thus, we can compute the local number of elements m for each partition by with an overall number of processes p for a global number of elements N by

$$m = \left\lfloor \frac{N}{p} \right\rfloor . \tag{4.1}$$

If there is a remainder r in this integer division, this remainder can be equally distributed to the first r partitions, resulting in one additional element for all those partitions. This is only a slight complication, and as N and p are known by all processes anyway, there is no communication needed to build the address of each global element.

After the actual adresses of all local elements in the parallel run-time configuration are known, the adjacent neighboring partitions are also known. At this point each process is aware of the remote elements it will need for the computation. The communication setup itself can then be set up without any further communication by just collecting all elements that need to be obtained from a given process into a single list. With such a list of elements for each neighboring partition, it is known with whom how much data needs to be exchanged. As the adjacency is symmetric, the exchange is also symmetric and each partner will receive as many elements as it receives. Thus, even so all processes just hold their own mesh partition and just minimal information about the global setup (total number of elements and total number of processes), each process can determine its communication partner completely local with the information provided by *GEUM*. To encapsulate the solver with the actual numerical scheme from the parallel implementation, we make use of additional layers of elements

that locally on each partition represent the remote elements. These elements are not computed by the local solver, but filled by communication and used in the stencils. This is a similar approach as the ghost cells, used for the implementation of boundary conditions, and we refer to them in the parallel context as halo elements. The information about these halo elements can now be directly exchanged with peer-to-peer communication. As will be described later on, this initial information on the immediate neighborhood will be used as a basis to find extended neighbor connections for higher order reconstructions. It is remarkable that the pre-processed mesh information by *GEUM* allows for such a scalable treatment of completely unstructured meshes with arbitrary polyhedral elements. Nevertheless, the non-scaling neighborhood identification now merely is shifted from the solver to a pre-processing tool, which itself can pose a bottleneck now. This might not be a severe restriction in many setups, but will pose a problem at some point, when the overall problem size that supercomputing systems could solve, are getting too large. At this inevitable point the mesh pre-processing becomes the limiting factor and larger systems can not be exploited anymore for more detailed simulations. We will address this bottleneck in Chapter 7 with a scalable mesh representation. However, this requires us to implement our own mesh generator.

A final remark on the IO of *GEUM*: Due to the very regular layout in the data files of the mesh it is not only possible to read the data with MPI-IO functions but also by direct access in Fortran. This is the main motivation to use different files for the various element types, which otherwise would also require variable record lengths within the files. Due to the well defined independent blocks of elements in the file, there is no overlap in the parallel reading of element data. Read conflicts might arise but only when the same vertex is read simultaneously by multiple processes. Such conflicts can only appear for vertices shared by multiple partitions and the reading via Fortrans direct IO usually works even for massively parallel computations.

4.3 Distributed WENO Stencil Search

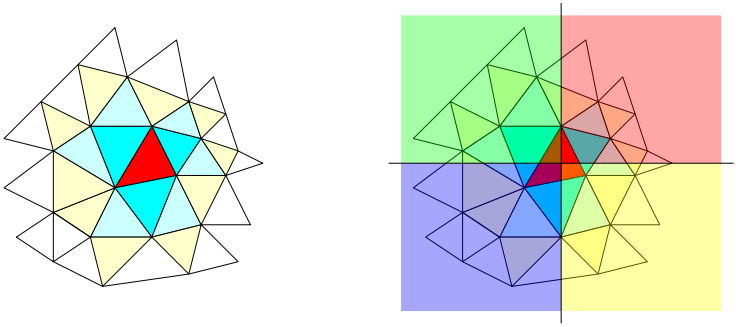
As already mentioned, WENO schemes of order 3 and higher, require reconstruction data not only from direct neighbors that have a common face with the current element but also neighbors of neighbors. In the $P_N P_M$ scheme, the number of required neighbors might be arbitrary large for a given reconstructed order M. While direct neighbors are immediately available from the *GEUM* format, these extended stencils are not stored

explicitly anywhere. Thus, they need to be constructed at runtime. This is expensive in serial informations, where the complete mesh information is available but finding the arbitrarily large neighborhood in the distributed mesh poses a non-trivial problem, as the neighboring elements might be arbitrarily distributed across various processes. As no mesh information beyond directly adjacent elements at the interfaces is available on each process, a mechanism is required to request additional data from remote mesh parts during the search for required neighbor elements. In MPI this can be achieved by one-sided communication or with the help of a communication procedure enabled by the newly introduced non-blocking barrier in the MPI-3 standard [20].

4.3.1 Sequential WENO stencil construction

Before moving on to the parallel treatment, we briefly discuss the serial stencil construction here. A more detailed description of the stencil construction can be found in [12]. For the WENO reconstruction, there are 9 stencils considered to cover the different spatial directions. The illustrations stick to the two-dimensional case to avoid confusion but the algorithm works in three dimensions in the same way. First, there is a stencil for the central reconstruction that uses layered rings of elements around the element to reconstruct. This stencil is depicted for two dimensions in Figure 4.7a, where the different colors indicate the individual layers around the element to reconstruct in red. Then there are eight additional stencils, one for each sector of the three-dimensional coordinate system with its origin in the barycenter of the element to reconstruct. In two dimensions, there are four sectors, which are shown in Figure 4.7b. For each of the sectors a non-central stencil is searched.

The algorithm to find all elements in the stencils follows an iterative search starting from the central element to reconstruct. Each iteration relates to one layer in Figure 4.7a. In each iteration all direct neighbors of the current layer are considered and added to the next layer, if they are not already part of an older layer. Additionally, each new element in the stencil is categorized to belong to one of the sectors, depending on the location of its barycenter. Thus, the first layer contains the direct neighbors of the central element, the second layer contains the neighbors of the neighbors and so on. All elements are added to the central stencil until a sufficient number of elements for the reconstruction has been added to it. Each element also is added to the respective sectorial stencil until a sufficient number of elements has been added to that stencil to allow a reconstruction with the desired polynomial degree. The iterations in this



(a) Layers of searched elements for the stencil determination

(b) Quadrants of stencils to search

Figure 4.7: Illustration of searched elements to identify stencils in 2D.

process are repeated until either all stencils are filled with the required number of elements, or a user defined maximum number of considered elements is reached. If the maximum is reached, some stencils will not contain the required number of elements and these stencils are deactivated for the computation. This case of deactivated stencils is only expected in the proximity of boundaries, where an insufficent number of neighbors might be available in the direction towards the boundary.

Figure 4.8 illustrates the resulting stencils for the example in Figure 4.7. In this two-dimensional example there are four sectorial stencils, which are indicated by the different colors (according to the sector colors in Figure 4.7b). Here, we were looking for a reconstruction by four elements, and each stencil contains the central element plus three sectorial elements. The central stencil in this example consists just of the central element and its three direct neighbors.

4.3.2 Parallel stencil search strategy

As indicated in Section 4.2, the parallel search for stencil elements in the higher order WENO schemes was previously done by providing the complete mesh data to each process. While this enabled the parallel search for stencil elements by all processes on their respective partitions, this requires the duplication of the complete mesh data on each process. The data duplication provides the simplest approach by not adapting

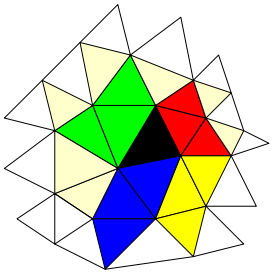


Figure 4.8: Elements of the non-central stencils.

the search strategy at all and instead using the serial algorithm on each process. However, the required memory to describe unstructured meshes is not negligble as each element needs to be explicitly specified with its sides, vertices for each side, neighbors of each side and coordinates of the vertices. With some additional book-keeping data the implementation uses around 593 bytes per element. Therefore, this previous strategy severely restricts the size of computable meshes on systems with locally limited available memory, i.e. all distributed memory systems.

An estimation of the required data to fully describe a tetrahedral mesh can be done as follows. For each element at least four vertices and four neighbors need to be stored. The neighbors are represented either by a global identification number with 8 bytes or a tuple of a local identification number on a given partition and the partition number with four Bytes each. For the representation of boundary conditions an integer needs to be stored for each side and for each side an indicator for the orientation of the neighboring side has to be provided. Finally the barycenter of each element has to be stored in three 8 Byte reals. Thus, in total a minimum of 104 Bytes is required to represent each element in the unstructured mesh. In the solver implementation some more data is stored, to avoid recomputations. The vertex coordinates also use some non-negligble memory, though it can not so easily be expressed on a per element basis, as the relation between number of elements and number of vertices depends on the mesh itself. The estimation of 104 Bytes, therefore, represents the bare minimum of required data and only allows the approximation of theoretical maximal problem sizes that could be computed with a given amount of memory per process. Commonly available memory per core on modern supercomputing systems

is around 2 GB, limiting the computable problem size to a theoretical maximum of 20 million elements in total. This maximum also neglects all other memory consumption besides the mesh, like the actual solution that we are interested in. While this theoretical limit already imposes a severe restriction for any simulation on the unstructured mesh, the actual implementation showed a much larger overhead per element of around 593 Bytes per element to represent a tetrahedral mesh. This tremenduos overhead is partly due to variables for a mesh description that allows various element types and allocated communication buffers. Due to this large, non-scalable memory consumption the totally solvable problem size is limited by this old parallelization strategy to a maximum of roughly 3 million elements on a system with 2 GB of main memory per core.

Figure 4.9 illustrates the memory consumption for a weak scaling, where the problem size grows with the number of processes (constant problem size per process). The measured memory per process is categorized into:

- *local*, the memory that is required by each process locally for distributed data, like the solution in the local partition.
- *misc.*, memory overheads that are considered minor and can not be directly attributed to local distributed data.
- *MPI*, memory that is consumed by the MPI library itself.
- *global*, the memory that is consumed by the global mesh information.

The figure clearly shows for a single process that the required memory for the mesh representation is negligble in comparison to the rest of the data in the local partition. With a growing number of processes and the accordingly growing total problem size in this weak scaling, the required memory on each process for the mesh quickly dominates the memory consumption. In this scenario, a problem that would only consume around 200 MB per process, fails already due to the memory consumed by the mesh with only 100 processes.

Since the memory per core is rather to decrease than to increase in the future [16], the change to the new distributed mesh handling as described in the following is essential to enable large scale simulations on massively parallel systems. The pre-processing of mesh data with *GEUM* enables the completely distributed loading of the mesh, without explicit global mesh information on each process. However, after loading the mesh it is still necessary to construct stencils for the numerical scheme. This stencil construction now also has to be done in parallel and on the distributed mesh.

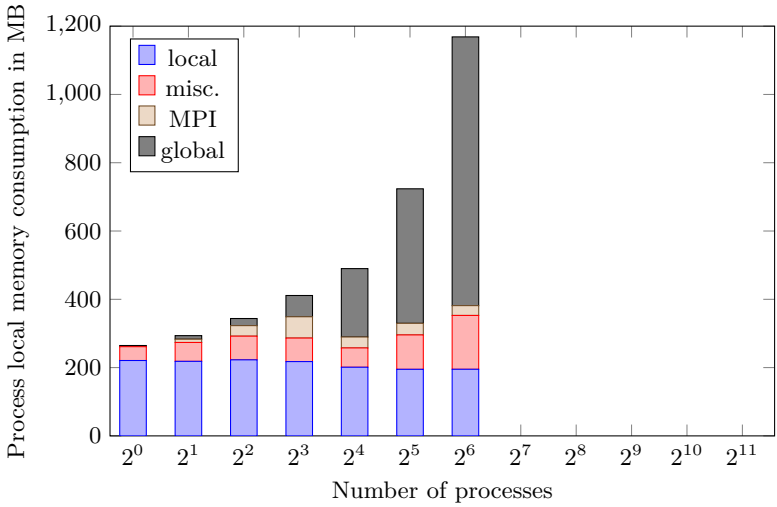


Figure 4.9: Memory consumption with global mesh information on each process.

While the previous approach with locally available global mesh data allowed for an easy deployment of the sequential algorithm in each partition, the new distributed approach adds the complexity to obtain remote data during the stencil search. The algorithm of the stencil search itself still is the same as described above but in the distributed setup we are now facing the problem that there is no local information on non-direct neighbors. Thus, when the stencil search encounters a partition boundary it needs to obtain information from another process. In MPI this is generally achieved by actively sending a message with this information. However, all processes are performing this search simultaneously, and the process holding the information in question does not know in advance, which other processes will ask it for which data. A possibility to deal with this is the usage of one-sided communication, which was introduced in MPI-2 [19]. This feature allows the access of any process to a defined window in the memory of another process, without the accessed process actively participating in the communication. However, it requires the exposure of all the mesh data in a raw chunk of memory, which is not always easily achievable with involved data structures as needed for unstructured meshes. Furthermore the communication concept, where one of the partners is completely passive, does not fit nicely into the usual MPI design. Under the hood the MPI implementation has to take care of these communications on the passive side. Thus, the activity on this end is still present, but just hidden from the application, which might result in bad performance. The MPI standard [19] elaborates on the need for an asynchronous daemon in the MPI layer on page 358. As can be seen there, the one-sided approach is rather involved and has potential portability issues attached.

Due to these implications of the one-sided communication in MPI, it seems to be attractive to use point-to-point communications instead, which fits better into the design of MPI. This can be achieved by implementing requesting and serving code: Each process requests remote elements during the course of its search for WENO stencils, while at the same time it listens for such requests from other processes. However, this strategy leads to a new problem, as no process can determine if the global task of finding all stencil elements is finished and it can stop listening and serving its mesh information. This termination of the stencil search task also has to be done, when using one-sided communications, but in this case a traditional barrier can be used to find the global completion of the task. With an approach based on point-to-point communications and some serving code on each process, this is no longer possible, as the serving has to continue, even after the local search has been finished. Thus this part in the code would have to be executed even after entering the barrier. In the recent

MPI-3 standard, non-blocking collectives have been introduced. For earlier MPI implementations the libNBC library offers the same functionality [34]. This part of the new MPI-3 standard offers a solution for the termination problem in the concept without one-sided communications, as it also introduces a non-blocking barrier. With the help of this non-blocking barrier it is now possible to signal the local termination of the stencil search, while continuing the serving of mesh data requested by remote processes. The global termination of the search task then can be tested, as the non-blocking barrier will be marked as finished, after all processes have called it and thus all local searches have been terminated.

4.3.3 Parallel distributed WENO stencil construction

With the communication strategy relying on actively passed messages, the implementation of the search for stencil elements gets a little bit more complicated. The cost attached to the avoidance of some automatic treatment by one-sided communication features under the hood is that these communications have to be taken care of in the application itself. A serving part needs to be introduced as well as a requesting part, those parts have to be intermixed with the sequential search algorithm itself.

As can be seen in the state diagram of the required program in Figure 4.10, the central state is the search for stencil elements. In addition to this a waiting state is introduced by the need to wait for requested remote data. Finally incoming requests from remote processes need to be satisfied. This is achieved by the three serving states, which all perform the same actions within, but appear in different contexts. The parallel search for all required stencil elements is done as follows: Each process iterates over its local elements and looks up all its direct neighbors to add them to the list of stencil elements. If the stencils are not all completely filled by this yet, a new layer of elements is considered, by looking up all direct neighbors of the elements in the list constructed so far. Whenever a new direct neighbor is encountered on a remote partition, a request is sent to the according process, and the process enters the waiting state to retrieve the data. To avoid deadlocks, this wait is not realized by a busy wait, but instead incoming requests are served until the data from the remote partition is finally received. After the requested data has been stored locally, the search can continue as usual. In order to prevent overly long waiting times on requesting processes the serving state is also entered everytime an element has completed all its required layers, and all stencils are known. When the local search for stencil elements is finished and all stencils are filled, the non-blocking barrier is initiated, and a serving

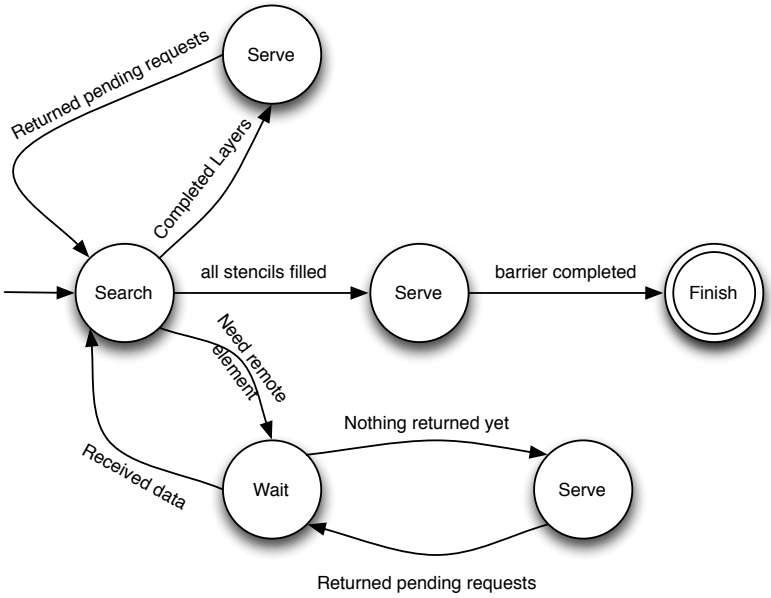


Figure 4.10: State diagram for the distributed WENO stencil search.

only mode is entered. At this point only incoming requests are served and periodically the completion of the non-blocking barrier is tested. As each process reaches this state only after all its local stencils have been fully defined, the non-blocking barrier can only complete when all stencils globally have been found. Thus the global search for stencil elements is completed when the non-blocking barrier is completed, and the serving state can be left by all processes.

Received elements from remote partitions are not only stored in the stencil information for each element, but in a global list. This avoids elements being requested twice by the same process and thus minimizes the communication effort. The list is organized in a dynamic data structure, which provides fast lookups for elements already in the list. Each element is uniquely identified by its global position in the mesh, represented by an eight byte integer. This provides a key for the array to search, and each element is put only once into the array. An efficient implementation providing these features is the Judy array library [75], which is accessed via Fortrans ISO-C-Binding module. These dynamic sparse arrays are actually not only required for the maintenance of received remote elements, but also for the construction of searched element layers. In the construction of these layers, each element should be present only once, however multiple elements will have the same neighbor, thus any element to be added has to be checked for existence in the list of already found stencil elements.

With *GEUM* and the distributed stencil search, a completely scalable method for unstructured mesh handling is now available. As can be seen in Figure 4.11, the memory consumption now remains fairly constant in the same weak scaling scenario as in Figure 4.9. The distributed stencil search together with the newly introduced mesh format enable thereby the usage of large meshes on several thousand processes. An important prerequisite to solve large problems.

4.4 Tracking Changes for Parallel Debugging

When developing parallel applications it is important to ensure the correctness of the parallel computations. Usually the parallel execution should produce exactly the same results, as the serial execution, if no order dependent computations, like summations across multiple processes, have to be done. However, debugging parallel applications is often non-trivial, especially when large amounts of data are involved. Some problems also occur only when many processes are used in the computation, as rare corner cases might only become apparent in this situation. While several tools

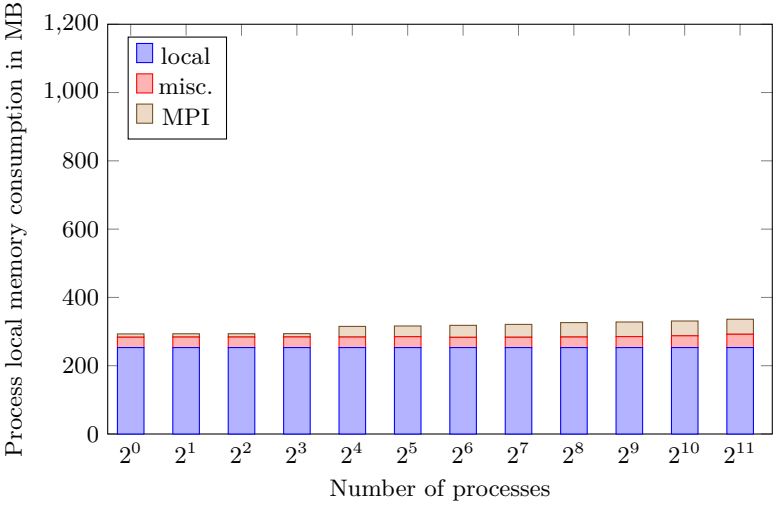


Figure 4.11: Memory consumption with distributed mesh information.

exist that allow for parallel debugging and program analysis, they are often limited to a usage with some few MPI processes. Debuggers like Totalview and DDT enable the introspection of data during the runtime on each process, but they do not provide an easy overview to globally distributed arrays. Yet, it often is useful to compare the content of such arrays with varying process counts, whenever it is expected to be independent of the parallelism in the computation.

With the global, partition independent, ordering of elements prescribed by the previously presented *GEUM* format, it is possible to compare data bitwise between various partitions. This includes the comparison of a serial run on a single partition and runs with multiple partitions. Thus, the chosen data layout provides the means to assess the consistency of simulation results in parallel executions. Checking for this consistency is of great help to identify potential problems. It also proofs to be a useful tool to validate restart functionality with varying process counts, as the results from the simulation should not be affected by restarting.

For overall assessments it is sufficient to compare the binary *GEUM* formatted output written to disk. A bitwise identity can be easily checked in a separate step after the runs. Usually, a checksum algorithm is used

to compare files with each other. For example the widespread cyclic redundancy check (CRC) with a polynomial length of 33 bits (CRC-32) [27] can be used for this task. Its implementation is for example available in the compression library zlib [23].

Luckily, the checksum computation in zlib is capable of working with partial sums. This allows its usage in parallel and is suitable for a concise check of distributed data at runtime. The parallel fast check for changes in distributed data sets allows for a better debugging of large scale simulations by inserting calls to the check into the application, as it enables the detection of code sections causing erroneous data even in runs on large numbers of cores. Besides this manual debugging work, the parallel computation of checksums also allows automatic checks for correctness, for example after communication. The CRC-32 implementation of the zlib library is used to compute partial checksums on each process, completely independent of each other and in parallel. Then its functionality to combine partial checksums (`crc32_combine`) is exploited in an MPI reduction with a user defined operation.

With this approach a scalable method is available to provide a concise analysis of distributed data. In a hexadecimal representation of the checksum, 8 digits are used to describe the state of the data. This, along with some context information can be printed to a log during the execution of the application. The developer is thereby able to track, if a code block changes the given data on any process or if a block of code behaves differently for varying process counts. This mechanism provides a first hint for a more detailed analysis, and thus, provides a great help in narrowing down problems in massively parallel runs.

5 Single Core Optimization Strategies

Though this work is mostly concerned with distributed parallel processing, the main goal is to lower the overall time to solution. For this goal, it is also important to utilize the available hardware as good as possible. Such single core optimizations are detailed and explained in this chapter. A few important features of modern processors are briefly discussed and some strategies in the implementation to utilize them are shown. For a general overview on this topic refer to the introduction by Hager and Wellein [26]. The main focus here is put on vectorization, as this is a mechanism of increasing importance for single core performance. Vector lengths on the x86 architecture, for example, has steadily increased with various revisions of SSE instructions, AVX instructions and then AVX2 instructions. Instead of investigating these short vector instructions that are only slowly growing as the architectures evolve. We will take a look in the future and utilize a full vector system as offered by the NEC SX systems. Vectorization strategies enabling the utilization of these systems are generally also beneficial on commodity scalar architectures.

5.1 Vectorization

Simple performance improvement of computations where each single instruction is performed sequentially for each single data (SISD) by reduced cycle times has come to an end [66]. However, Moore's law [54] of exponentially growing number of transistors in CPUs still applies. Those transistors are now used to build more cores and functional units, that can work in parallel within a single CPU [49]. Computing algorithms therefore have to exploit this parallelism in order to experience any speed-up on future hardware. In a rough categorization, following Flynn's taxonomy [18], the parallelization can be sorted into a single instruction stream acting on multiple data streams (SIMD), multiple instruction streams acting on a single data stream (MISD) and multiple instruction streams acting on multiple data streams (MIMD). All modern processors provide several levels of different parallelism out of those categories. As instructions are demanding in terms of electric power consumption, SIMD parallelism is an effective method to save energy. This chapter concentrates on this

SIMD parallelism and uses for it the common term vectorization. It is a fundamental concept in GPU programming, but is also found in general CPUs in form of SSE instructions in the x86 architecture or the Double Hummer of the BlueGene/P Power processor for example. Specific details on how these instructions can be efficiently used, can be found in [21]. Not surprisingly, the utilization of vector units is most important on dedicated vector systems like the NEC SX systems, with its latest incarnation being the NEC SX-ACE. The key requirement for vectorized processing of multiple data streams is their independence and this section is meant to briefly highlight its importance.

Another important property of modern hardware is the access to memory. Though this resource is also growing exponentially, the growth rate in bandwidth and latency of the memory access is lower than that of the instruction execution. This problem is commonly referred to as memory gap [31]. A development to counter this issue and leverage a higher data rate to the instruction streams is the usage of transistors on the CPU as cache, which allows fast access to frequently used data. With the growth of the memory gap, the cache infrastructure got deeper, larger and more complex. The result of this development is a memory hierarchy in all modern computing systems. To feed the vector instruction stream properly an implementation has to take into account this memory access hierarchy and the data layout has to be designed accordingly. Sustaining high performance on a given system, the software has to be aware of the access times and sizes of memory on the various levels in the hierarchy of the hardware. On some systems like the IBM Cell processors the transfer from one memory region to another has to be explicitly programmed by the application. The design of programs for this kind of architecture thus, has to take the memory hierarchy explicitly into account. Caches to the contrary try to provide a view on the complete accessible main memory and implement a lot of logic in the hardware to achieve this transparancy to the user. Though, this provides for the applications a convenient single level access to the main memory, it obscures the means to keep often used data close to the processing units. Applications are not forced to take care of the memory hierarchy in this case. However, to fully exploit the computing power and achieve a high sustained performance, applications have to be written with these mechanisms in mind. Thus, the application needs to be aware of the locality of its data and aim for many operations on close by data. We will discuss this in a little more detail in Section 5.3.

From these two guiding principals, data independence and data locality, some basic concepts for single core optimizations can be derived. They are important for all modern processors, and increasingly so with growing

vector lengths in the processing units, as for example exposed by the new AVX instructions for the x86 architecture, the integration of GPUs into CPUs or Intels Xeon Phi.

5.2 Importance of Visibility of Data Independence

A necessity for any parallel execution on multiple data streams is their independence. This independence needs to be visible to the system. Otherwise, it can not take advantage of it. As vectorization is usually performed by the compiler, the independence needs to be clearly recognizable for the compiler in the code. If the independence can not be sufficiently expressed in the programming language, compiler directives might be used to help the compiler. To show some basic concepts for the vectorization and the impact on the sustained performance, a simple but widely used kernel, a dense matrix-matrix multiplication is used here. This is an important kernel in linear algebra and heavily optimized implementations of them usually exist in vendor-provided libraries on HPC systems. A reference for the upper bound of achievable performance can be stated with the help of these implementations. To highlight the necessity of the detectable independence of operations on the multiple data, a Fortran and a C implementation of this matrix-matrix product are used and compiled with their respective compilers.

In Listing 5.1 a naive implementation of a matrix-matrix product with dynamic data is shown. Even in this simple case, vectorization is prohibited, as the arrays are all defined as pointers and may overlap according to the C semantics. The NEC SX-9 has a theoretical peak performance of 102.4 GFLOPs, the C code from Listing 5.1 shows a performance of only 0.088 GFLOPs for 1024×1024 matrices. As the compiler can not deploy vectorization, in this case, the code can not take advantage of the vector processing units and, therefore, runs only with 0.086 percent of the theoretical peak performance.

The very same algorithm, but in Fortran as shown in Listing 5.2, is running with roughly 16 GFLOPs without any tunings. Please note, that the index ordering is reversed in the Fortran implementation in comparison to the C implementation, to achieve the same physical memory ordering. Though it might be, that the multidimensional arrays in C result in distributed memory sections, the impact of this factor was found to be relatively low on this system in comparison to the vectorization effects. Still to reduce the differences between the two implementations, the C-code uses a single allocated block with a sufficient length to store the complete matrix

Listing 5.1: Simple Matrix product in C

```
void mxm( int lda, int m, int l, int n,
          double a[][lda],
          double b[][n],
          double c[][n] )
{
   int    i, j, k;
   for( i = 0 ; i < m; i++ ) {
      for( k = 0; k < n; k++ ) {
          c[i][k] = 0.0;
      }
   }
   for( i = 0 ; i < m; i++ ) {
      for( j = 0; j < l; j++ ) {
         for( k = 0; k < n; k++ ) {
            c[i][k] = c[i][k] + a[i][j]*b[j][k];
         }
      }
   }
}
```

and just interprets it as a two dimensional array in the subroutine. In Fortran the pointer attribute is not as flexible as in C and those restrictions allow the compiler to assume data independence here. However, it is more natural in Fortran to avoid the usage of pointer attributes and instead use the *allocatable* attribute where possible. In subroutines these arguments can even be completely hidden from the compiler, giving it the greatest flexibility for optimization. This is indicated in the commented declaration of the array arguments, which is the typical form for array arguments in Fortran 77. To which degree the compiler can exploit this information is of course depending on the compiler itself. The NEC compiler for the SX system is already providing a fairly vectorized code even with the pointer attribute attached to the arrays. As can be seen, on this system with the NEC compilers the recognition of data independence results in a performance difference of three orders of magnitude. The C implementation could of course also be enhanced by directives to explicitly inform the compiler of data independencies. However, this comparison also nicely highlights the convenience offered by the Fortran semantics for vectorized computations. In fact, this very naive approach can be heavily enhanced and a tuned version for this algorithm from the BLAS library provided by the vendor, gains over 80 GFLOPs on the NEC SX9.

It should be emphasized, that the main difference here is indeed the visibility of data independence to the compiler. Only if the compiler can detect, that the three used arrays are independent, it can vectorize the operations. If existing data dependencies are not apparent to the compiler, it is necessary to either change the code or provide hints to the compiler in the form of some directives. Otherwise the compiler will not be able to automatically generate vectorized code. Though this impact is seen most drastically on a vector system like the NEC SX, this effect can also be observed on more common architectures with much shorter vector registers.

5.3 Exploiting the Memory Hierarchy

Usually, computations are executed in a pipelined fashion, which results in some starting and closing time intervals of each execution block, where the functional units are not fully used. From this point of view it would be beneficial to make the innermost loops as long as possible in order to minimize the effect of those dead times. However, it is often necessary to transfer data from one execution block to the next. In this case it is necessary to keep this data as close to the processing units as possible. As the available memory space gets smaller the closer it is to the functional

Listing 5.2: Simple Matrix product in Fortran

```
module mxm_module
contains
subroutine mxm(lda, m, l, n, a, b, c)
  implicit none

  integer :: lda, m, l, n
!!  real(kind=8) :: a(lda,m), b(n,l), c(n,m)
  real(kind=8),pointer :: a(:,:), b(:,:), c(:,:)

  integer :: i,j,k

  c = 0.0_8

  do i=1,m
    do j=1,l
      do k=1,n
        c(k,i) = c(k,i) + a(j,i)*b(k,j)
      end do
    end do
  end do

end subroutine mxm
end module mxm_module
```

units, it is necessary to build suitable chunks of data, which fit into those smaller storages.

Data exchange with the main memory is bound by limited bandwidth and high latencies, in comparison to the floating point operations speed. Due to this growing memory gap [16], it is a key strategy to avoid memory accesses for optimized codes. By keeping a chunk of data nearby for several execution blocks increases the number of operations per byte loaded from main memory. Exploiting the memory hierarchy provided by the hardware yields dramatic benefits for some algorithms and is more important than very long vectors on most architectures. Thus, locality of data should be exploited wherever possible. Yet, the available storage close to the processors should be filled as much as possible, that is, with as long vectors as fit into the limited storage. Each chunk of data should follow the rule of visibility of independent data. Also the longer each chunk itself is, the more efficient it can be executed. Therefore any implementation has to be a compromise between maximizing chunk sizes but minimizing the necessary data transfers to and from the main memory.

The NEC SX series provides very fast storages, directly addressable by the programmer, in the form of vector data registers. Their size is obviously limited by the vector length and there is only a limited number of registers available. As these data registers are almost equivalent to the actual arithmetic registers, it is beneficial to keep vectors in them, which are frequently accessed. If the vector data registers can be used, the block size for innermost loops is naturally given by the vector length of the hardware (256 words in case of the NEC SX9).

Newly introduced with the NEC SX9 is an addressable data buffer (ADB), which acts as a kind of cache and sits between the main memory and the processing units. Its size of 256 KB is very limited, yet already large when compared to the available vector data registers, as it translates to 128 vectors against 32 vector data registers. It also provides greater flexibility for the arrays, that may be put on it. However, it is also much slower than the vector data registers. Explicit compiler directives need to be provided for each data that should be kept on the ADB. Any other data bypasses it.

Other systems usually provide fully associative caches, where it is generally sufficient to keep the data chunks in reasonable size. The hardware automatically keeps fitting data in its caches. This approach yields less control, but is potentially easier to use. In any case the effects for the programmer are similar, leading to the same strategies to take advantage of the available memory closer to the processors.

5.4 Vectorization of the Cauchy-Kowalevsky Procedure

The Cauchy-Kowalevsky (CK) procedure [44] is a core component in both, the numerical scheme of the solvers and the coupling mechanism. It is used by by the *ADER* scheme to achieve a high-order time integration and by the coupling to arrive at an extrapolation for intermediate time step values that are not provided by a neighbor with a coarser time resolution. In this section we will highlight, how this crucial part of the code can be improved from an original version that works reasonably fine on a scalar system to one that also provides high performance on a vector system.

The implementation of the CK procedure utilizes the approach by Dyson, described in [14] for two Euler equations in two dimensions. It relies on repeatedly applying the generalization of Leibniz's rule [59] in multiple dimensions. With two space dimensions and one in time Leibniz's rule yields

$$\frac{\partial^{a+b+c}(f(x,y,t)g(x,y,t)}{\partial x^a y^b t^c} = \sum_{k=0}^{c}\sum_{j=0}^{b}\sum_{i=0}^{a} \left[\binom{a}{i}\binom{b}{j}\binom{c}{k}\frac{\partial^{(a-i)+(b-j)+(c-k)}f(x,y,t)}{\partial x^{a-i}\partial y^{b-j}\partial t^{c-k}}\frac{\partial^{i+j+k}g(x,y,t)}{\partial x^i \partial y^j \partial t^k}\right]. \quad (5.1)$$

Listing 5.3 shows the original code implementation of this method. There are several nested loops to be seen. The outermost loop with the loop counter variable `iGP` iterates through all Gaussian integration points. For each of these integration points, the CK procedure has to be done, resulting in the six nested loops. Those loops reflect the underlying structure, found in (5.1). All derivatives for $a+b+c < m$ need to be computed for a given representation order m in time. For each of the derivatives, the three nested summations have to be computed. Refer to [14] for more details on the algorithm for the Euler equations. In Listing 5.3 only a first block of the innermost three loops is shown. This block computes an intermediate result for the algorithm to find the derivatives for the Euler equations and is followed by several more of these blocks. All within the outer three loops for a single derivative. Each following block utilizes the results from previous summation blocks. To illustrate the vectorization strategies it is sufficient to look at this first code block here. There are two large arrays `V` that holds intermediate results and avoids recomputations and `W` that contains the actual derivatives. The tuple `(a,b,c)` identifies a derivative and `i,j,k` are summation indices. There are many Gaussian integration

Listing 5.3: Original CK implementation excert

```
REAL :: V( 1:6, 0:Ord-1,      &
  &             0:Ord-1,      &
  &             0:Ord-1, nGPs ) ! temp.
REAL :: W( 1:4, 0:Ord-1,      &
  &             0:Ord-1,      &
  &             0:Ord-1, nGPs ) ! deriv.
REAL :: d, V11, V21
! Ord around 10; nGPs around 10^6

do iGP=1,nGPs

  do c=0,Ord-2
    do b=0,Ord-1-c
      do a=0,Ord-1-c-b

        do k=0,c
          do j=0,b
            do i=0,a-1
              V11 = V11                          &
                & + d * W(1, a-i, b-j, c-k, iGP) &
                &     * V(1, i, j, k, iGP) ! scalar
              V21 = V21                          &
                & + d * W(1, a-i, b-j, c-k, iGP) &
                &     * V(2, i, j, k, iGP) ! scalar
            end do
          end do
        end do
       ! Several blocks of this kind,
       ! each using the results of previous blocks!
```

points (`nGPs`), for which this operation has to be performed. Typically, in the order of millions. The other dimensioning variable `Ord`, refers to the order of the scheme and has mostly an order of magnitude around 10.

With this initial version of the code a poor performance of only 436 MFLOPs on the NEC SX-8 can be observed because the deeply nested loops in the end just perform a scalar operation. Due to this low performance, this operation consumed around 60 % of the overall computing time on the vector system. However, the computation for the various Gaussian integration points are all independent from each other, and we can change the order of executions. By performing each instruction to all Gaussian integration point and only then moving on to the next instruction, each operation can be vectorized and we achieve a higher performance.

This change is illustrated in Listing 5.4. Note that we not only changed the ordering of the loops but also the indices of the large arrays `V` and `W` to allow for an efficient memory access. With these changes, the implementation achieves 2.6 GFLOPs on the NEC SX-8 machine. However, we now increased the memory consumption by requiring the intermediate variables `V11` and `V21` to be arrays of size `nGPs`. In the actual algorithm there are some more of those intermediate variables that get inflated to arrays by the vectorization. Thus, this implication is more severe than it might appear at first. Furthermore, the change affects the execution on scalar systems badly because of the increased memory bandwidth demands and reduced data locality.

To overcome these drawbacks, let us introduce a strategy called strip-mining or loop-sectioning [87]. Strip-mining splits the long loop over all `nGPs` Gaussian integration points into shorter loops and the only executes one "strip" at a time. We thereby get one additional loop over all strips. Listing 5.5 shows the resulting code. The newly introduced parameter `vrl` describes the length of our strips. It is chosen to be 256, which is the vector register length of the NEC SX-8. Using this strip-length enables us also to use the vector data registers on this architecture for the temporal variables `V11` and `V21`. Those intermediate arrays now only need to have the strip length of `vrl`. To put them into vector data registers, we use the compiler directive `!cdir VREG`. The vector data registers provide us with fast access to the stored data and eases the memory bandwidth requirements.

Now have a look at Listing 5.5 to see the newly introduced outer loop with the loop counter variable `strip`. This is the loop over all our shortenend loops, which are chunks of Gaussian integration points. It is the outermost loop here, so we perform the complete CK procedure for each strip of integration points, before continuing with the next strip. The strip length `vrl` might not divide the overall number of integration points

Listing 5.4: Vectorized CK implementation

```
REAL :: V(nGPs, 1:6, 0:Ord-1, &
  &                  0:Ord-1, &
  &                  0:Ord-1  ) ! temp.
REAL :: W(nGPs, 1:4, 0:Ord-1, &
  &                  0:Ord-1, &
  &                  0:Ord-1) ! deriv.
REAL :: d
REAL :: V11(nGPs), V21(nGPS)
! Ord around 10; nGPs around 10^6

do c=0,Ord-2
  do b=0,Ord-1-c
    do a=0,Ord-1-c-b

      do k=0,c
        do j=0,b
          do i=0,a-1
            V11 = V11                          &
              & + d * W(:, 1, a-i, b-j, c-k) &
              &     * V(:, 1, i, j, k) ! vector
            V21 = V21                          &
              & + d * W(:, 1, a-i, b-j, c-k) &
              &     * V(:, 2, i, j, k) ! vector
          end do
        end do
      end do
     ! Several blocks of this kind,
     ! each using the results of previous blocks!
```

by an integer number, and we take care of that by computing an actual loop length `est` $\leq$ `vrl` to ensure the last strip has the correct length. We then replace the innermost loop, that is the array assignment in Listing 5.4, by the shortened loop with length `est`. Moreover, we realize that the temporary array `V` was overly large in the previous implementation and can also be reduced to the length of the strip instead of the total `nGPs`. That is, the temporary array is required for the overall CK-procedure, but only for each integration point and we can reuse the memory of the temporary variable in all strips. We now have two different indices for the large arrays, once the global one across all integration points in `W`, and once the index within the strip, used in `V`. The global index in `W` is obtained by adding the offste `strip` to the strip local index `iGP`.

With these changes, a performance of 3 GFLOPs is achieved, which is only a slight improvement over the previous variant with 2.6 GFLOPs. However, we now have the possibility to adjust the inner loop length to the machine instead of requiring a large bunch of memory in dependency of the problem size. This makes this variant now equally well suited for vector and scalar architectures, providing us with a kind of portable performance. The strip-mining length parameter even provides us now with an option to tweak according to the cache size of the scalar processors.

The strip-mined version of the code in Listing 5.5 provides a much better execution performance on the NEC SX-8, then the original version in 5.3 but in comparison to the actually available theoretical peak performance on the vector system, it still is not quite satisfactory. A single processor of the SX-8 has a theoretical peak performance of 16 GFLOPs, so the achieved performance so far is less then 20 % of that. In the next and last Listing 5.6 one more step is shown, which boosts the performance beyond 30 % of the peak performance.

In Listing 5.6 a change is made, that might appear minor but yields a rather large increase in the achieved performance. With the strip-mining step, we also reduced the size of the temporal array `V` to the strip-mining length. This reduction enabled us, not only to reduce the required memory but also to improve the locality of the required data. Can we achieve the same for the working array itself? At first it might not appear like this should be possible, after all it contains the resulting data we want to compute. Yet, on second thought, we are utilizing too much data here. We actually are only interested in the time series in the end. All those mixed derivatives are only needed as intermediate values and not used anymore after the time series has been obtained. Thus, we can restrict the input data `W` to just the spatial derivatives and the output to just the derivatives in time. This newly introduced output array is called `TimeDer`

Listing 5.5: Strip-mined CK implementation

```
INTEGER, parameter :: vrl = 256
REAL :: V( vrl, 1:6, 0:Ord-1, &
  &                  0:Ord-1, &
  &                  0:Ord-1  ) ! temp.
REAL :: W(nGPs, 1:4, 0:Ord-1, &
  &                  0:Ord-1, &
  &                  0:Ord-1  ) ! deriv.
REAL :: d
!cdir VREG(V11, V21)
REAL :: V11(vrl), V21(vrl)
! Ord around 10; nGPs around 10^6

do strip=0,nGPs-1,vrl
  est=min(nGPs-strip,vrl)
  do c=0,Ord-2
    do b=0,Ord-1-c
      do a=0,Ord-1-c-b

        do k=0,c
          do j=0,b
            do i=0,a-1
              do iGP=1,est
                V11 = V11                        &
                  & + d * W(iGP+strip,           &
                  &         1, a-i, b-j, c-k) &
                  &     * V(iGP, 1, i, j, k)
                V21 = V21                        &
                  & + d * W(iGP+strip,           &
                  &         1, a-i, b-j, c-k) &
                  &     * V(iGP, 2, i, j, k)
              end do
            end do
          end do
        end do
       ! Several blocks of this kind,
       ! each using the results of previous blocks!
```

in Listing 5.6. We then make use of a local working array `LW` that only needs to have the strip length but provides space for all mixed space-time derivatives. The spatial derivatives now need to be copied into this local working array. However, this copy can be done outside the six times nested loop construct. Thus, we spent an one-time memory copying but gain then a stride-one memory access in our repeatedly executed vector operations in the inner most loop. Actually, we need to do a two copies, once in the beginning and once at the end to copy the data into the `TimeDer` result array. Nevertheless, the gain by the better memory access within the deeply nested loops is dramatic and we can achieve a performance of 5.2 GFLOPs with the code version from Listing 5.6.

It is remarkable that this change also enables the vectorization on scalar processors and by utilization of SSE instruction the performance is also nearly doubled on scalar systems in the last code version in comparison to the original one. For the vector system NEC SX-8, we achieved a speed-up of more than a factor of ten from 436 MFLOPs to 5.2 GFLOPs. In other words, without this serial optimization, the parallel execution of the application would waste 90 % of the computational resources in each process. This highlights the importance of single core optimizations along with the scalability.

5.5 Machine Comparison With *APES*

This chapter, so far, should have highlighted the importance of single core performance for large simulations. Let us now in conclusion have a look at the performance, as observed on different systems with distinct properties. Three different processor architectures will be considered here, the NEC SX-ACE vector processor, Intels x86 architecture and IBMs BlueGene Q Power processors. For Intels x86 processors we also consider two different interconnect systems. Once an Infinband based cluster and once a Cray Aries torus network.

The machines for this scalability analysis are the three German national supercomputing systems and a small installation of a SX-ACE system at the HLRS in Stuttgart:

- *Hornet*: Cray XC40 system with Intel Haswell E5-2680v3 2,5 GHz and 12 cores per processor. This system provides a fast torus interconnect via Cray's Aries network. It is offered by HLRS in Stuttgart.

- *Juqueen*: IBM BlueGene Q with IBM PowerPC 1.6 GHz and 4 cores per processor (16 per node). The BlueGene offers a fast 5D torus

Listing 5.6: CK implementation with reduced memory

```
INTEGER, parameter :: vrl = 256
REAL :: V(vrl, 6, Ord, Ord, Ord) ! temp.
REAL :: W(nGPs, 4, Ord, Ord) ! spat. deriv.
REAL :: TimeDer(nGPs, 4, Ord) ! time deriv.
REAL :: LW(vrl, 4, Ord, Ord, Ord) ! local work
REAL :: d
!cdir VREG(V11, V21)
REAL :: V11(vrl), V21(vrl)
! Ord around 10; nGPs around 10^6

do strip=0,nGPs-1,vrl
  est=min(nGPs-strip,vrl)
  do iGP=1,est
    LW(iGP,:,:,:,1) = W(iGP+strip,:,:,:)
  end do
  do c=0,Ord-2
    do b=0,Ord-1-c
      do a=0,Ord-1-c-b

        do k=0,c
          do j=0,b
            do i=0,a-1
              do iGP=1,est
                V11 = V11                              &
                  & + d * LW(iGP, 1, a-i, b-j, c-k) &
                  &     * V(iGP, 1, i, j, k)
                V21 = V21                              &
                  & + d * LW(iGP, 1, a-i, b-j, c-k) &
                  &     * V(iGP, 2, i, j, k)
              end do
            end do
          end do
        end do
       ! Several blocks of this kind,
       ! each using the results of previous blocks!
```

interconnect with special support for collectives. It is provided by the Forschungszentrum Jülich.

- *SuperMUC*: Lenovo NeXtScale nx360M5 WCT system with Haswell E5-2697v3 2.6 GHz and 14 cores per processor. The nodes in this system are connected by a non-blocking tree with Infiniband FDR14. It is operated by LRZ in Garching.

- *Kabuki*: NEC SX-ACE testing installation with 64 nodes in total. The vector processors have 4 cores and the nodes are connected by an IXS crossbar.

For the scaling analysis we will use the Lattice Boltzmann (LBM) solver *Musubi* [30] from the *APES* suite. The LBM kernel is relatively small and can be highly optimized. Thus, it as an attractive code for benchmarking of machines. Note however that it is strongly memory bandwidth dependent. As simulation setup we use a simple cubical domain with periodic boundary conditions in all directions. For the initial condition a small Gaussian pulse is used.

Figure 5.1 shows a strong scaling analysis for *Musubi* on the four introduced systems. The x-axis represents the theoretical peak performance of the machine fraction used for the simulation run. The y-axis represents the obtained performance by the LBM solver in terms of floating point operations per second. We can observe that the three scalar systems all exhibit a very similar behaviour with nearly ideal scaling over a large fraction of the machine. Even a short period of superlinear speed-up can be observed in the Intel processor systems. These are due to caching effects, where the problem per process has been reduced so far that it completely fits into the cache. The NEC SX-ACE system *Kabuki* stands out in this analysis. It is only a small system with a total of 64 nodes and is, therefore, limited in the theoretical peak performance but as far as it scales it outperforms all other systems significantly. Even so, the scaling is worse than on the large Petascale systems. The advantage in the high sustained serial performance is sufficiently high to compensate the relatively bad scaling. The bad scaling behavior arises in this case from the decreasing problem size per process and a thereby decreased vectorization.

The comparison in Figure 5.1 highlights that a simple concentration on scalability does not suffice and we always need to keep the serial basis in mind. It also shows that number of floating point iterations are just one measure to assess a computing system and we need to also consider other factors, like memory access.

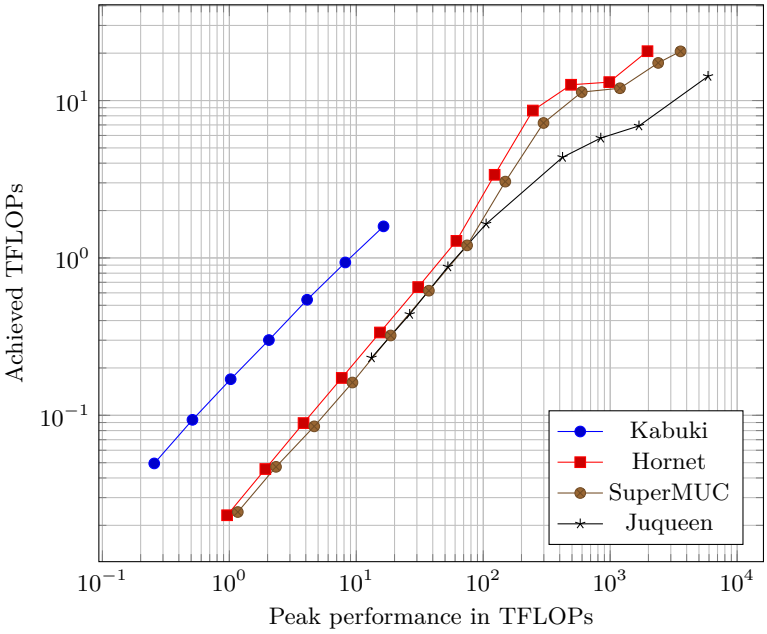


Figure 5.1: Sustained performance scaling on german supercomputing systems. *Kabuki* is a small NEC SX-ACE system. *Hornet* is a Cray XC 40 Petascale system. *SuperMUC* is a Lenovo NeXtScale Petascale system, and *Juqueen* is a BlueGene Q Petascale system.

6 Scalable Distributed Coupling Method

This chapter describes the coupling method in general and the specific features that enable a parallel distributed computation. The serial version of the coupling developed in [82] does not face some of the issues discussed here. Section 1.2.1 described the fundamental idea of the coupling to be based on point values that are exchanged between domains. A prerequisite to enable such a coupling is the identification of points in a given set of domains. In serial all mesh information is available, and points are identified by simply checking all mesh elements for their containment. Checking the complete volume might be legitimate in serial, as the exact element containing the point needs to be determined anyway. However, in parallel, such a search is not an option anymore. In the distributed computation we try to keep the information on remote partitions as small as possible, as explained in Chapter 4. Thus, remote elements are not available. Instead, we only check point containment against surfaces of the partitions. With the partitioning of unstructured domains, these surfaces take the form of arbitrary polyhedra. Thus, while the serial strategy was only concerned with the containment in simple geometries like tetrahedra or hexahedra, we now need to deal with the containment of points in arbitrary polyhedra.

The search for the points itself is done in parallel and distributed, avoiding large peaks in memory consumption. After the strategy for the initial point identification has been discussed, we will look into the data exchange at those points during the simulation. The space-time expansion of the numerical scheme is exploited to allow updates on the coupling information without a prescribed ordering that would impose a serialization. Synchronization problems that arise from the two different tasks coupling and actual solving on each process are explained and eliminated.

With parallelization and multiple discretization domains, we have two different kinds of splittings of the overall computational domain. First there are the individual domains, and these are then split into multiple partitions. Let us introduce the term *section* for these parts. Each section is identified by the process that is computing it, and the domain it resides in. The coupling now happens between individual sections without any intermediate layers. It is possible that multiple domains are computed

by the same process but in most setups there is each process dedicated to a single partition of a single domain. However, the concept of sections allows for the most general distribution of the heterogeneous domain setup on parallel computing systems.

6.1 Point Localization in Arbitrary Polyhedrons

The coupling method relies on ghost elements at the coupling interfaces. As explained in Section 1.2, the status in these ghost elements is constructed by values at discrete points. The discrete points are chosen at the integration nodes of numerical integration schemes and provide and abstraction from the underlying meshes. Values at points can easily be obtained in the deployed numerical schemes, as there are either reconstructed polynomials or state representation in form of polynomials within each element available. A remaining problem is the localization of these points. The partitioning of unstructured domains for parallel computations results in arbitrary polyhedral volumes for individual domains. Figure 6.1 illustrates this in three dimensions with transparent partitions of different color. For the coupling, we need to identify points within such polyhedra. This section discusses options for this task and explains the algorithm chosen for the implementation.

The problem that needs to be solved here is the point containment decision for arbitrary polyhedra. There are two classes of algorithms that are typically used to solve this problem. One is based on the *Jordan curve theorem* and relies on the computation of intersections. The other is based on the *Gauss-Bonnet theorem* [90] and requires the computation of oriented solid angles. Both approaches are applicable in three dimensions and for arbitrary polyhedrons.

In the following, we will investigate both approaches. First the popular method based on the *Jordan curve theorem* is investigated, then we move on to its alternative based on the *Gauss-Bonnet theorem*. For the coupling we will use the latter one and the reasoning for this choice will also be discussed.

6.1.1 Approach based on the Jordan curve theorem

The Jordan curve theorem states that a simple closed curve in the plane divides the plane into an interior and an exterior region, and a point in either part can only be connected to the other by a line, that intersects the curve. This has been extended to higher dimensions by the Jordan-Brouwer separation theorem. Due to the fact, that the interior region is limited, the

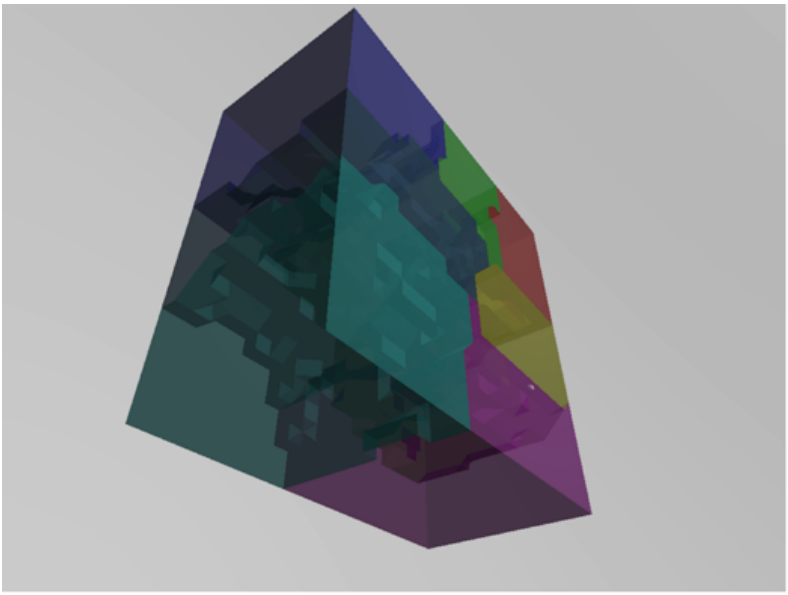
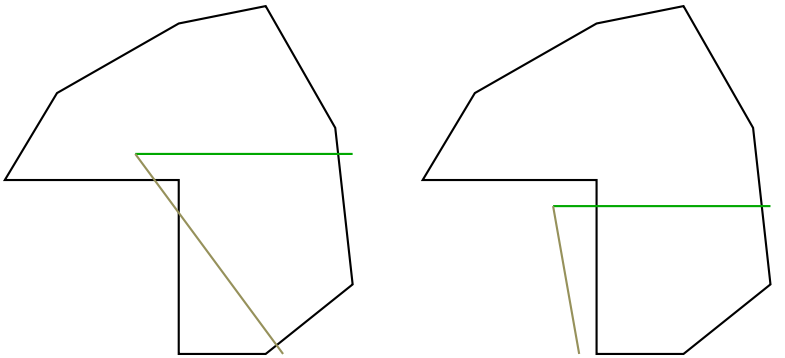


Figure 6.1: An example for the partitioning of a three dimensional mesh with arbitrary surfaces.



(a) Point inside the polygon. Here we observe an odd number of intersections (one for green, three for yellow).

(b) Point outside the polygon. For this point outside the polygon we observe either no intersection (yellow) or an even number of intersections (green).

Figure 6.2: 2D-Illustration of the Jordan algorithm to decide point containment. Two rays are cast from a single point into two different directions.

containment of a given point in the interior can be tested by connecting the point with a point infinitely far away that has to be in the exterior part. In other words, we cast a ray from the point in an arbitrary direction. This ray from an internal point has to intersect the surface. Thus, the check to be computed in this algorithm is the ray surface intersection.

As outlined above, the ray has to intersect the surface at least once if the point is inside the polyhedron. When there is no intersection at all, the point has to be in the exterior part. For multiple intersections it can be seen, that always two intersections would cancel each other out, as one of them has to be an entering into the interior, while the other one has to be an exit from it. With this pairwise canceling of intersections, the point containment can be decided upon counting the intersections. For an even number of intersections, the point has to be in the exterior, while for an odd number of intersections, the point has to be in the interior. Because of this behavior, this method is also referred to as parity check. Figure 6.2a shows the situation for points inside the polygon. The situation for points outside the polygon is shown in Figure 6.2b.

The attractive feature of this method is the fact, that the ray surface

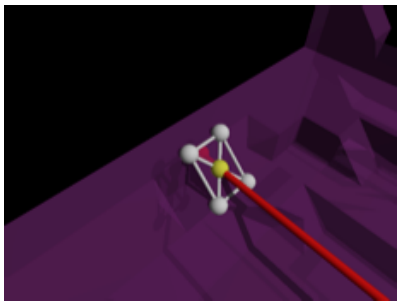


Figure 6.3: Illustration of a corner case for the Jordan algorithm, where the ray intersects a vertex of a partition interface.

intersection can be computed efficiently. All meshes considered here will have surfaces, which can be represented by a union of plane triangles. Intersection tests are, therefore, only necessary for ray-triangle intersections and further the intersections themselves do not have to be actually computed, instead tests for their existence are sufficient. The computational effort can be reduced when using a specific ray, like the positive x axis. However, a drawback arising in this approach is the occurence of degenerate cases in the ray surface intersection. For example the ray might be coinciding in a line section, or the intersection might be exactly on a vertex. The question that arises here is how many intersections have to be accounted in such degenerate cases? Whenever the ray runs through a vertex, there are potentially two line segments of the polygon that might count this as an intersection. While this is solvable in two dimensions, the degenerated cases get much more troublesome in three dimensions, this is for example nicely explained by Kalay in [39]. Figure 6.3 illustrates the intersection of the ray with a vertex in the surface of a three-dimensional mesh partition. There are four triangles indicated by the white lines with a common vertex in the yellow sphere. The ray is shown in red and runs through this common vertex. To actually decide the intersection type in such a case, all adjacent surface triangles have to be taken into account as illustrated in Figure 6.4 by the green lines and spheres. In this example the triangles are coplanar. However, in general this might not be the case and an enclosing surface may not be found.

A common solution to overcome such degenerate cases is a slight rotation of the ray. However, this looses the advantage of exploiting special ray

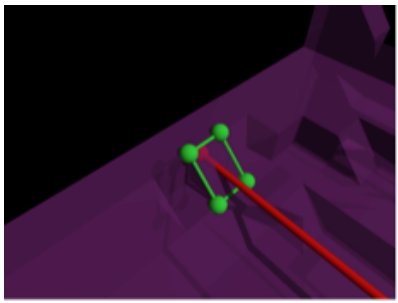
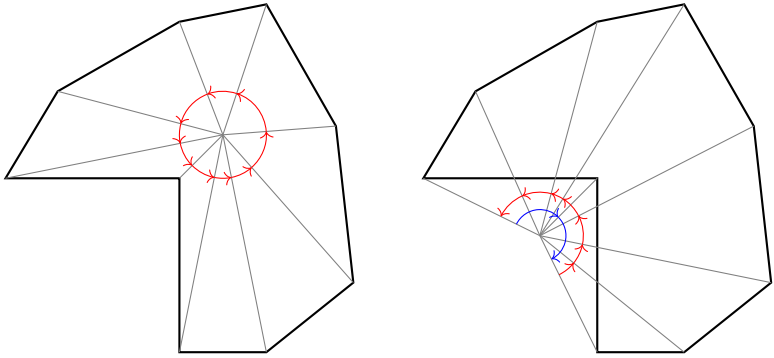


Figure 6.4: Required faces to be checked to correctly identify the corner case in the Jordan algorithm.

directions like the positive x axis. Another approach is offered by the simulation of simplicity [15], where vertices are slightly perturbated to always obtain a non-degenerated case. In any case the parity check requires some additional effort in implementation and computation to decide the containment of the point. For some cases there might not even be a satisfactory answer from the algorithm. This gives reason to investigate an alternative, that is based on the Gauss-Bonnet theorem.

6.1.2 Approach based on the Gauss-Bonnet theorem

The Gauss-Bonnet theorem states that the total curvature of a well-behaved closed surface, as those of the partition boundaries in the considered meshes, will always be 4π three dimensions, regardless of the shape of the surface. Correspondingly, in two dimensions the total curvature of closed surfaces is 2π. The sign, under which a given surface part will be seen from a certain point of view depends on the orientation of the surface normal. This can be exploited to decide the containment of a point within an arbitrary polyhedron as described by Lane et al. [47] With proper orientation of all surface elements (all normals pointing outwards), any point inside a polyhedron will see a total solid angle of 4π, while those on the outside will see a total solid angle of zero. Figures 6.5a and 6.5b illustrate this idea in two dimensions for some arbitrary polygon. Points inside a polygon add the angles with each polygon side up to a total of 2π, as illustrated with an example in Figure 6.5a. Figure 6.5b on the other hand shows, how the angles cancel each other out for points outside the polygon.



(a) Point inside the polygon. The summed angle of all surfaces is 2π for points inside the polygon.

(b) Point outside the polygon. The summed angle of all surfaces is 0 for points outside the polygon. Angles counted positive are shown in red, and those counted negative in blue.

Figure 6.5: 2D-Illustration of the Gauss-Bonnet algorithm to decide point containment in polygons.

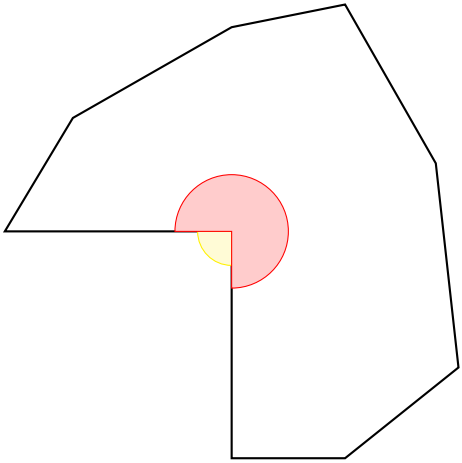


Figure 6.6: Illustration of the situation for a point on a vertex of the polygon. The resulting angle from the algorithm can be interpreted as the fraction of the point that belongs to the polygon (red). A potential neighboring domain is indicated in yellow.

However, also this approach suffers from degenerate cases. These arise, when the point is exactly on the surface of the polyhedron. In this case, the integrated solid angle seen from that point might be any value between 0 and 4π. Such a case with the point on one of the vertices of a polygon in two dimensions is shown in Figure Figure 6.6. The red angle indicates the summed total angle as seen from the vertex of the polygon. Unfortunately, such corner cases can not be excluded in the scenario of the coupling interfaces. Interpolation points might very well happen to coincide with partition boundaries of arbitrary mesh configurations.

The corner cases highlight a common issue in both approaches: There is always a final binary decision that needs to be taken. A point either belongs to a partition or not. However, if a point is indeed exactly on a partition boundary, there is no real preference for which partition it should be accounted to. There just needs to be a consistent decision, and the point needs to be accounted for exactly one of the adjacent partititons.

The method based on the *Gauss-Bonnet theorem* allows an elegant solution to this problem. Instead of trying to take a binary decision, a point can simply be accounted for the domain for which it has the largest

total solid angle. This covers the common case as well as the degenerated ones and even offers some geometrical meaning. Such a geometrical interpretation can be most easily understood when looking at a vertex that is shared by several partitions. Figure 6.6 offers an illustration of this scenario in two dimensions. Imagine a neighboring partition to the depicted polygon on the lower left. Both polygons are adjacent to the vertex, for which the angles are shown. In the shown polygon, the angle sums to the red angle, while for the outside polygon the angle shown in yellow is obtained. As can be seen, the integrated solid angle of each partititon describes the opening angle under which the point is reached from within the partition. This angle can also be understood as the share of the vertex for this partition. Thus, attributing such a point to the partition with the greatest solid angle is the most natural approach to deal with this degenerated case.

An implementation issue, that arises by this algorithm free of binary decisions is that all points will be accounted as being part of one domain, even if they are actually in none of them. Though this case should never happen, it might occur due to wrongly defined mesh configurations by the user. It therefore is necessary to inform the user about points which can not be put into any domain. A solution to this issue is the generation of some warning or error if the maximal solid angle for a point is close to zero. For a maximal solid angle close to numerical accuracy it is likely that no domain actually contains the point. Millions of partitions would have to be adjacent to the point to legitimately reach such a small maximal fraction and we safely can discard this scenario.

Other deficits in this approach are the need for oriented surfaces and the need for trigonometric functions to compute the solid angle of each surface part. Both are only minor, as the orientation of the mesh surface can always be ensured and efficient methods to compute the solid angle are known especially for plane triangles [84]. The benefits on the other hand make this strategy very attractive to the use case in this work. Due to the avoided binary decision it is extremely robust and every point will be accounted for exactly one partition that has to take the responsibility for it. A decision on the maximum of the solid angle can be done with a parallel reduction operation and the workload for each point is exactly the same.

6.1.3 Point containment summary

We investigated two options to decide the point containment in arbitrary polyhedrons. The ray casting method based on the *Jordan curve theorem*

has the advantage that it works independently of the surface orientations. Theoretically this method can be computed fast with specialized ray-triangle intersections. However, due to corner cases in many binary intersection decisions, it usually is necessary to perform several ray castings, which diminishes the computational efficiency. The greatest disadvantage for the coupling scenario is the bad treatment of points on interfaces, which requires careful treatment.

For the second approach, based on the *Gauss-Bonnet theorem*, the surfaces need to have properly defined normals. This is not a severe restriction in our scenario, as the surfaces are obtained from mesh definitions, and we always can ensure a correct definition of the surface elements. With this restriction it becomes possible to avoid binary decisions and corner cases altogether. All points, also those on interfaces between partitions, can be attributed to one partition by a simple maximum decision.

6.2 Distributed Coupling Scheme

The described method to localize any given point in an arbitrary polyhedron can now be used to find the actual coupling relation between the various domains. This coupling is to be done in a distributed parallel algorithm to allow its deployment on large scale supercomputing systems. As the coupling scheme is purely based on discrete points, it works equally fine on 2D and 3D domains and with arbitrary domain layouts.

Arbitrary domain layouts are important to avoid too limited applicability of the coupling method for engineering simulations, that might have a complex geometrical setup. However, this generality of the geometrical configuration turns the identification of neighbor relations into a quite expensive operation, as each coupling point needs to be tested against all partitions. This can only be sped up by deploying some spatial sorting that would allow a faster identification of spatial regions to test for containment. In this section the strategies to deal with the full problem of any point in any domain and the simulation steering based upon this are described.

6.2.1 Coupling interface identification

Two strategies to deal with the coupling interface identification in parallel and distributed are suggested and their scalability potential is discussed in this section. As already described, the coupling is done on the basis of the partitioned domains. These are referred to as sections and are uniquely identified by the domain number and the rank of the process

that is computing it. Coupling points that will be used to exchange values between the domains therefore have to be located in these sections.

For a total number of N coupling points and s sections, the number of tests that have to be performed are therefore $T = N \cdot s$, where N is fixed for a given problem, while $s(p)$ will grow with the number of processes p. In the parallel identification of the coupling neighborhood, these T tests are distributed on p processors, resulting in an overall ideally achievable running time of T/p. However, this neglects that each containment test itself gets cheaper with shrinking number of surface elements of each section.

The first considered parallelization strategy is the exchange of surface descriptions between all sections. Each section then tests its local coupling points against all other section surfaces and decides upon this with which sections a communication will be needed. With this strategy all surfaces of all sections need to be stored in each section, which is not desirable for large numbers of sections, that is large scale simulations with many small sections. However, it avoids the distribution of all points to all sections and identifies all local coupling points in parallel. A further limitation of this approach is the uneven distribution of coupling interfaces across all processes, resulting in load imbalances.

In the second parallelization strategy the coupling points are broadcasted to all processes, and each section just needs to test these points against its local surface. While in this approach the identification of point containments is serialized, the work load is better balanced as domain descriptions are kept local. This avoids the increased memory consumption for large numbers of sections, while at the same time the memory requirement due to the coupling points can be kept constant by using a maximum package size for the exchange of points. As the containment testing of points is essentially serialized with respect to the point set, such a limiting into packages does not affect the time complexity.

While in the first approach the coupling neighbors are identified by the requesting domain and afterwards an all to all operation informs the source domains about the requests, the second approach uses allreduce operations to identify the winning domain that will provide data at given points. This has also the benefit that a vectorized testing of all points against the local surface description can be performed to speed up the containment identification.

The second algorithm to test point containment in parallel and with distributed sections in summary has the following steps:

- `MPI_Allgather` is used to gather the number of coupling points on

each process (sum of all coupling points in all domain partitions on that process). Obviously this requires memory in the order of numbers of processes on each process. However, the amount of required memory is just a single 4 Byte integer.

- After all processes are aware of the number of coupling points that will be requested by each process, the processes broadcast their actual point coordinates one after another. The number of points broadcasted in a single message by any given process is limited by an user defined upper limit on the package size.

- Upon each broadcast, each process computes the maximal share for each of its domain partitions at each of the requested points, using the Gauss-Bonnet based method.

- The global maximal share for each point is then found with the help of an `MPI_Allreduce` with the `MPI_MAXLOC` operation. The allreduce operation is used to notify all processes of the actual coupling partners, such that subsequent communications can be made on a peer to peer basis.

- After all points have been localized, the interacting processes exchange their domain information and set up the actual communication between sections.

This approach scales nicely due to the allreduce operation that has only a running time of $\mathcal{O}(\log p)$. No memory bottleneck is introduced as the usage of point packages allows for a limitation of memory consumption in the global exchange. However, it remains the problem, that all coupling points have to be tested essentially in serial by all processes. Though, the testing typically gets cheaper with smaller domain partitions per process, this is an intrinsic obstacle to a highly distributed simulation. Even in the static setup considered here, it might inhibit large simulations by overly long initialization times. In more dynamic settings with dynamic load balancing or mesh adaptation, where the exchange partners have to be recomputed, the costs for this non-scaling approach would quickly get prohibitive high. It can only be overcome by a restriction in the domain topology, such that the localization of points can be done without explicit testing on all processes. This will be highlighted in more detail in the development of the octree based mesh framework *TreElM*.

6.2.2 General coupling properties

To the solvers in each domain, the coupling interfaces completely look like ordinary boundary conditions. Boundary conditions are prescribed by ghost elements outside the computational domain. In the case of coupling interfaces, these ghost elements are provided by the coupling framework. These ghost elements therefore build a sort of adapter between adjacent domains. This scheme was introduced by Utzmann et al. in [83]. With the construction of these cells by point values it is possible to couple arbitrary domains together in three dimensions. Due to the concept of a coupling via boundary conditions, the solvers in each domain are mostly independent of the other domains. Therefore, it is possible to use the best adapted implementation and equation system in each of the domains. The previous section described, how the coupling points required to build the ghost elements can be located in parallel. Now the actual parallel coupling between the various domains in the simulation is described.

For the coupling four orthogonal classifications of domain properties can be made:

- *Equations*. The domains might solve for nonlinear Navier-Stokes, nonlinear inviscid Euler or even linearized Euler equations.
- *Schemes*. Discretization can be based on Finite Volume, Discontinuous Galerkin or Finite Differences.
- *Time*. Explicit schemes adopt the time step proportional to the spatial step size. Coarse meshes therefore can go for larger time steps than fine meshes. A sub-cycling has to be implemented.
- *Meshes*. Coarse and fine as well as structured-unstructured meshes are used.

The most important domain property that affects the parallel execution, is the mesh representation. Parallelization of each domain is achieved by partitioning the mesh in the domains. This results in the arbitrary polyhedral sections described above. At the same time it partitions the coupling interfaces in the same way as the domains are subdivided. This aspect will be highlighted with greater detail in Section 6.2.3.

The other classes are now briefly presented to provide a full overview. A detailed description for the coupling in the serial scheme is given by Utzmann in [82]. Coupling of different equation systems can be achieved by defining proper interface conditions. Such interface conditions need to translate one state into another as required by the adjacent domains.

The schemes of interest in in this work are the Discontinuous Galerkin (DG), the Finite Volume (FV) and the Finite Differences (FD) scheme. None of them causes problems, as they all can make use of ghost cells to prescribe boundary conditions. As the state in each cell can be obtained from discrete point values for all of them, it is no problem to construct conforming ghost cells. The ghost cells are constructed as for boundary conditions completely independent of the adjacent coupling domain. Then the set of discrete points are employed to obtain state values from the other domain. Again the other domain does not need to care about the actual cells of the requesting domain, instead it just needs to determine a state at all requested points and return them. By the usage of points in between, we obtain a coupling scheme that abstracts from the meshes on either side but still allows for high-order representations as an arbitrary number of points can be used to construct the state in the ghost cells.

Different time-steps in adjacent domains are obtained by a sub-cycling scheme, where larger time-steps are defined as multiples of the smallest time-step and interaction only happens at common time levels. In between the common time levels, the ghost cells for the domain with a smaller time step are updated by the Cauchy-Kowalevsky (CK) procedure [44]. The CK procedure allows a conversion of spatial derivatives into temporal derivatives in the partial differential equation. Thus, it enables the construction of a Taylor expansion to extrapolate the current state in time.

6.2.3 Spatial coupling

The spatial coupling is based on interpolation points in the ghost cells at the interface between two adjacent domains. For Finite Differences, the interpolation points are the actual ghost point coordinates. In Finite Volume and Discontinuous Galerkin schemes a volume integration rule is required for the cell to construct the intra-cell representation of the state. This is achieved by Gauss quadrature, and its integration points are used as the interpolation coordinates for these schemes. Restricting the method to discrete points simplifies the coupling, as it turns it essentially independent of the actual domain configuration and spatial dimension of the domains. That is an important feature to allow simulations of complex settings and apply the method in a wide range of applications. In Figure 6.7, an example of a ghost cell with some interpolation points is shown. The ghost cell is overlapping two domains and the discrete points used for the construction of the cell can be uniquely assigned to the appropriate one.

The state in the providing domains needs to be evaluated by that

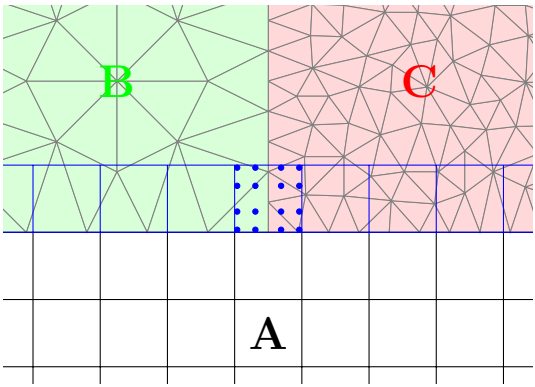


Figure 6.7: A coupling ghost cell of structured domain A, overlapping both, domain B (green) and C (red). Ghost cells are shown in blue, for the cell overlapping both neighbors, the Chebyshev-Gauss integration points for fourth order are indicated by blue dots.

domain for all these points. Domains providing data at the interpolation points are called source domains, whereas the ones with the ghost cells are referred to as target domains. The evaluation is done with the state representation used by the numerical scheme in that domain. However, if this evaluation is deemed too expensive and not necessary for the required accuracy in the target domain, the evaluation might be limited to fewer degrees of freedom. A nice effect of this coupling approach is the fact that the coupling works independently of the numerical discretization in the domains. Domains that are solved with *DG* can simply evaluate the polynomials at required point coordinates, while *FVM* schemes will employ their reconstruction to obtain the polynomial interpolation for all coordinates. In any case, the coupling mechanism does not need to care about how the values are obtained but can rely on the fact, that values are provided with the accuracy available in the discretization of the source domain.

On the other side, the number of interpolation points to construct the ghost cell is given by the order of the scheme in the target domain. Now, with a polynomial degree d^S in the representation of the source domain, and a number of coupling points d^T, dictated by the scheme order in the target domain, the computational effort for the coupling is

given by $C = d^T \cdot d^S$. With high-order schemes on both sides, this quickly gets expensive. Therefore, there is an option to limit d^S deliberatly. A limitation of the interpolation order is especially desirable for a finer source domain coupling with a coarser target domain. In this case the accuracy of the coupling method is not affected too much by a reduced interpolation order, as the higher spatial resolution itself already provides accurate data to the coarser domain. This can be illustrated with an error approximation on the basis of a Richardson extrapolation [64], as given in (6.1).

The idea in the Richardson expansion is to approximate the error E by a monomial series of the step size h:

$$E = c_0 h^0 + c_1 h^1 + c_2 h^2 + c_3 h^3 + \dots \text{ for } h < 1. \tag{6.1}$$

If the interpolation order is assumed to be the same as the one of the numerical scheme, only higher order terms remain in the error estimation. Neglecting all but the then dominant term offers a simple error approximation of the interpolation. Requiring equality of errors on both sides results in (6.2).

$$\mathcal{O}(c_A h_A^{p_A}) = \mathcal{O}(c_B h_B^{p_B}) \text{ with constants } c_A, c_B \tag{6.2}$$

Let h_B be the step size of the finer resolved mesh and p_B its order of interpolation accuracy. After rearranging (6.2) and solving for p_B, the relation in (6.3) is found to approximate the necessary interpolation order in the higher resolved domain.

$$p_B = \mathcal{O}(\frac{\ln(h_A)}{\ln(h_B)} \cdot p_A - k) \text{ with } k = frac\ln\left(\frac{c_B}{c_A}\right)\ln(h_B) \quad \text{and } h_B < h_A < 1 \tag{6.3}$$

As the ln function is strictly increasing, and < 0 for the given spatial intervals, the absolute value of $\ln(h_B)$ is always larger than the one of $\ln(h_A)$ under the given assumptions. If the coefficients in the Richardson expansion were equal, the constant k diminishes. Neglecting k we then obtain a lower required polynomial order p_B on the finer resolved mesh to maintain the same accuracy as required by the coarser mesh. Thus, when coupling a highly resolved source domain with a coarser resolved target domain, it is appropriate to limit the polynomial terms for the computation of state values at the integration points of the target domain.

In structured Finite Volume or Finite Difference source domains we use Lagrangian polynomials in each dimension to find the state at the requested interpolation points. In unstructured source domains with DG the numerical solution within the cells themselves are directly used. For

unstructured source domains using the $P_N P_M$ scheme [11] as described in Section 3.5, the reconstruction is performed and the reconstructed polynomials are evaluated.

A complication arises where the stencil for the polynomial around the integration point itself includes ghost cells again. In such a configuration a circular data dependency appears for the coupling interface. To resolve this problem exactly, it would be necessary to solve an equation system across all influenced coupling points. However, such an iterative solution for the coupling interface would further increase the already high computational costs for the interfaces. Another option instead of solving the cirular dependency, is given by using an approximation for the ghost cells needed in the interpolation. The data dependency is broken by using an extrapolation of the state in ghost cells with the Cauchy-Kowalevsky (CK) procedure [44]. As this extrapolation needs to be done for intermediate time-steps anyway, this fits nicely into the rest of the framework. The CK procedure converts spatial derivatives into temporal derivatives, which allows the extrapolation by a Taylor series. It therefore, allows us to maintain the high-order spatial accuracy also in the temporal extrapolation and maintains the accuracy across the coupling interfaces.

With the state values at the interpolation points, the state in the ghost cells of the target domain have to be constructed. As the interpolation points are chosen such, that they are conforming to the numerical scheme of the target domain, this results in simply applying the element local treatment as in regular fluid elements of that domain. Usually, this treatment is a numerical integration of the state values.

In the described coupling mechanism a natural separation appears between the interpolation step and the construction of the ghost cells. The interpolation has to be performed in the source domain and no information other than the coordinates of the integration points are required from the neighbors. Whereas the construction of states in ghost cells just needs the state values in its integration points, and no further information about the source domain is needed. This separation ensures the generality of the approach to couple arbitrary domains, and provides a natural path for parallelization which will be elaborated in the following section.

6.2.4 Parallelization

The parallel identification of the coupling points needed to build the ghost elements at the interfaces has already been described. In the following, the parallel strategy for the coupling itself is explained. Indeed the parallelization is very straight forward, as there is an inherent separation in the

coupling method. Yet, the parallelization on the level of physical domains is not sufficient. Each domain usually is very demanding in itself and needs to be parallelized also internally. Such a parallelization is achieved by partitioning the domain into smaller parts. Then each part can be computed independent from the other ones with communication on the interfaces.

As described in the introduction of this chapter, we will use the term *section*, to address the interacting parts in the parallel, heterogeneous simulation. The need for this new term arises from the two different subdivisions in the overall simulation domain we are facing in the parallel execution. In the multi-scale setup we have various physical domains with different discretizations and equations to solve that need to be coupled. In a parallel execution, we now subdivide each physical domain again into smaller partitions. This subdivision results in two indices for each individual part. First, there is the index of the physical domain and secondly there is the rank of the process, which will process this part within the physical domain. Such a processing unit is referred to as a section in this work to distinguish it from (physical) domains and (parallel) partitions. With this concept it is possible to distribute the computational domain arbitrarily on the available processes. Especially, a process might participate in the computation of multiple domains. Thus, a single process might compute several physical domains but each domain might also be computed by multiple processes. With the unique identification of each section by rank and domain index, any distribution is just equally handled by communication between the various sections. This yields the greatest potential for scaling, as no central processing is required. All communication happens between equal peers in a local neighborhood, so the total number of processes used in the simulation is not limited by any many-to-one communication patterns, that would be imposed by dedicated processes taking care of the coupling itself.

However, there are two levels present in the parallel scheme, as there is a difference in the communication within each physical domain and the communication between adjacent domains. To separate these levels in the communication layout, MPI communicators are used to group the processes within each physical domain together. The individual communicators allow the solvers in each physical domain to act within this communicator just as it would act if run without any coupling. Thus, a homogeneous structure is maintained within each solver, and all heterogeneity is left to the coupling itself. The achieved modularity allows easier integration of existing solvers and helps to separate the different algorithms. Despite this division into two levels all the communications for the coupling are nevertheless done

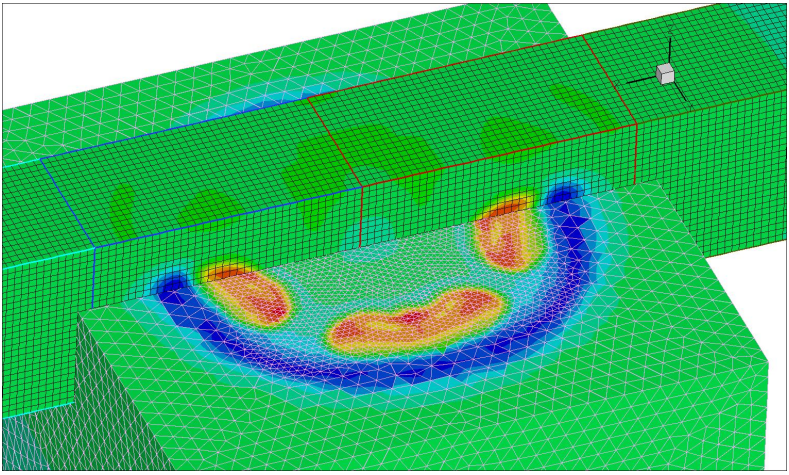


Figure 6.8: Illustration of a decomposed and partitioned three-dimensional coupling setup with unstructured and structured partitions. Only one row of structured partitions are shown to allow the view on the unstructured domain beneath. The color indicates the instanteneous pressure field of the coupled flow simulation.

between individual sections. Processes just happen to rely on different communicators for the coupling via ghost cells (`MPI_COMM_WORLD`) and the exchange within the physical domain (local communicator). This layout is sketched in Figure 6.8, where 2 different domains are shown, one structured and one unstructured. The image is a cut through a 3D computation, and the sections are shown along with the containing meshes.

Figure 6.9: Separation of the overall computing time into two different tasks. These blocks of tasks repeat every iteration.

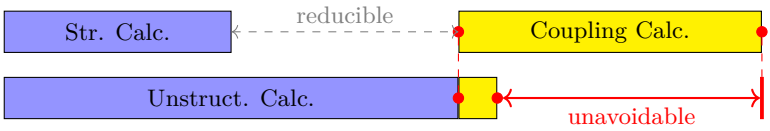


Figure 6.10: Illustration of a setup with strongly imbalanced code blocks and two points of synchronization. There appear two idle times. Different ones in each execution thread. The one (in grey labeled reducible) between domain calculation and coupling can be minimized by using more processes for the unstructured part (lower row). However, the other (shown in red as unavoidable) can not not be diminished by the same means, it would even get worse with more processes for the unstructured part.

6.2.5 Balancing and synchronisation

The coupling mechanism adds another task to the solver. Each section with a coupling interface not only needs to compute the elements of the computational domain, but also has to perform the evaluations for the ghost-cells. As outlined above, this task might be expensive and far from negligble. It is also a task that might consume dramatically different amounts of time in the two coupling partners. The two time blocks of different tasks are depicted in Figure 6.9. In the parallel simulation there obviously needs to be a synchronization, as at some point in time the data at the coupling points needs to be exchanged. However, as it happens, the serial and initial parallelization performed the computation of the time step width after the domain calculations but before the coupling point computations. Thus, there where two synchronization points active in the setup, separating both parts from each other. This causes large waiting times if not both blocks in themselve are balanced, as illustrated in Figure 6.10. In that figure the simulation is split into two processes, one with an unstructured part and the other with a structured part. Both are coupled with each other. Imagine the structured solver is much faster than the unstructured one, but calculating the coupling data is much slower. With the two synchronization points, one after the domain calculation and one after the coupling calculation, we are stuck with the situation in Figure 6.10.

On each side there is large idling time gap, due to the massive imbalances in each code part. The structured solver has to wait until the unstructured

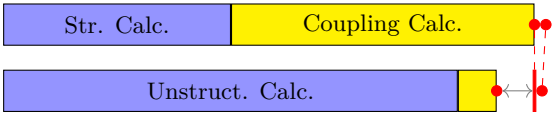


Figure 6.11: Moved middle synchronization point after the second synchronization, to combine both code execution blocks and allow for a balancing of the overall computing time. Though there are two synchronizations required here, they are now immediately following each other, resulting essentially in a single point of synchronization.

solver has computed its domain, only then it can start to compute the coupling data. The unstructured solver on the other hand is quickly done with the computation of the coupling and has to wait on the structured solver to complete this step, before advancing to the next iteration. Only one of these gaps might be reduced by using more processes for the part that consumes more time. The other gap will inevitably remain or even grow. In Figure 6.10 this is indicated by the reducible gap between structured calculation and the coupling calculation on that side. By putting more processes on the unstructured side of the execution, the unstructured calculation will shrink and this reducible gap diminishes. However, the gap in the coupling imbalance on the unstructured solver will remain, and even grow.

Luckily, the synchronization between both code parts is merely the agreement on the time-step size and subcycling, which is easily moved. The only reason for it to be at this point in the code execution is because this happens to be the point, where it has been in the serial code version. Moving the synchronization for the timestep after the coupling point exchange yields the timing blocks shown in Figure 6.11. With this change it now is possible to achieve a balancing between the two solvers, as there is only a single point of synchronization per iteration remaining. The single task block allows us to distribute processes according to the required computing time in both sides.

The lesson to learn from this example is the importance of few synchronizations and the gathering of all synchronizations at a single point if possible. If it is not possible to reduce the points of synchronizations to a single point during execution, we will likely face imbalances that can not be cured by any tuning of process distributions.

6.2.6 Summary coupling

In summary the distributed parallel coupling scheme has the following design. The coupling mechanism is purely based on the exchange of data at discrete points in space. All required interpolation points are localized on a per section basis and communication structures for the data exchange at these points are creaated accordingly. As described in Section 6.2.1, two strategies for this parallel localization of the coupling points are available. After this initial setup only peer to peer exchanges are required between adjacent sections. This exchange is done in the coupling framework and causes some overhead on both sides. The source domain needs to perform the interpolation to all requested points and the target domain has to construct its ghost cells from the received data. These additional computing times need to be considered when partitioning the domains. Especially, it is necessary to avoid synchronization points between the solver computation and the coupling computations. Though, they might sum up to the same time on either side, they can vary drastically in their parts and intermediate synchronization between them would lead to large idle times. Synchronization between the domains are necessary to determine, if the time-steps and the sub cycles need to be adjusted.

6.3 Coupling Across Different Machines with PACX-MPI

The domains used in simulations might vastly differ in terms of the numerical requirements. Therefore, it might be beneficial to also use different hardware for the individual domains within the coupled simulation.

6.3.1 Structure of the application

While the individual solvers in the KOP framework can be run as stand alone and independent of each other, the coupling framework itself is built on the basis of an MPI environment enclosing all parts. The coupling part acts as the main program, which calls the solvers and controls the complete setup. There may be any number of different physical domains coupled together, each with its appropriate solver. Those are steered by the coupling framework by imposing the subcycling and communication on common iteration cycles. As outlined above, the coupling is realized by taking care of the boundary conditions at the interfaces between domains. The taken approach of discrete points for the coupling between different physical domains enables arbitrary discretizations, numerical schemes and even equations in adjacent domains and retains the high order of

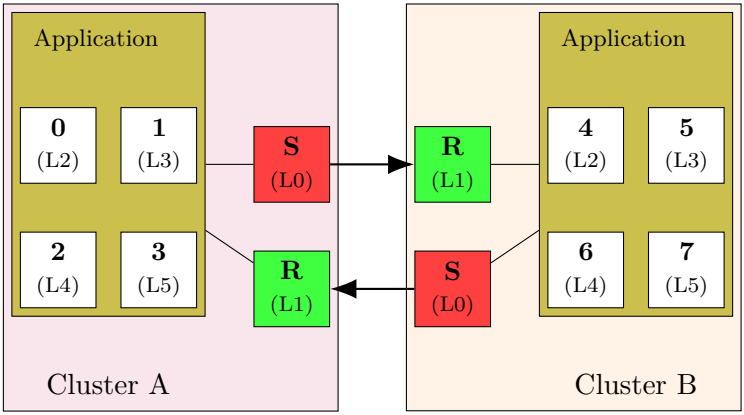


Figure 6.12: Layout of the PACX-MPI communication across distinct clusters. The MPI ranks as seen by the application are shown in large bold letters from 0 to 7. Below them the local process numbers are noted in brackets.

involved schemes across domain boundaries. Typically, there is more data to exchange more often within a physical domain than between different domains, resulting in a relatively weak interaction between domains. Nevertheless, all components are tied together into a single program in order to fully eploit the possibilities offered by an MPI environment.

6.3.2 PACX-MPI

PACX-MPI [10] is an MPI layer developed at HLRS which enables an MPI parallel application to be spread across several clusters [22], hence its name PACX as in *Pa*rallel *C*omputer e*x*tension. For the application this distribution is hidden by PACX-MPI, which acts as an intermediate layer between the application and the actual MPI libraries on each involved cluster.

6.3.3 Communication layout

PACX-MPI takes care of the communication between different clusters by using one sending and one receiving daemon on each of them. Communication requests between processes residing on different machines is intercepted

by PACX-MPI and tunneled through the daemons to the corresponding machine. Communication between processes within the same cluster is handed over to the local MPI implementation. With this approach the optimal usage of the networking infrastructure on each side is available to the application and PACX-MPI is free to use any different appropriate network protocol to establish the inter-cluster link. This layout of the process distribution within PACX-MPI with the communication daemons and the mapping of local MPI ranks to global MPI ranks is shown in Figure 6.12. In the figure two clusters with six processes each are depicted. On each cluster two processes need to be reserved for PACX-MPI, they will take care of the communication between clusters. One acts as a sender (S) and one as a receiver (R), these are colored in red and green respectively. The remaining eight processes across both clusters are numbered according to the MPI rank visible to the application. In addition, each process is labeled with the cluster-local process number in brackets.

6.3.4 Heterogeneous environment

The PACX-MPI library may be used to link smaller, similar clusters together to gain more computational power for a single simulation, than on each of them may be available. But it is also capable to couple different kinds of machines together. For example, it transparently takes care of necessary data conversions from little to big endian. In the following sections, we will cover how this combination of different architectures can reduce overall computation times in the heterogeneous simulation by exploiting the best fitting system for each numerical scheme. As PACX-MPI is providing the usual MPI interface to the application transparently, it provides a homogeneous setting on top of the heterogeneous hardware. This yields the advantage to have the possibility to distribute any MPI application across a heterogeneous environment, however it should be kept in mind that, in order to use the hardware efficiently, the application should be aware of the weak links between different clusters nevertheless.

6.3.5 Starting an application using PACX-MPI

There are two possibilities to start an application using PACX-MPI. One option is to define the complete setup by hand in configuration files on each site. This approach is a very static and strict one, but the complete setup can be defined in every detail, which might be necessary in some configurations. A more dynamic approach is offered by the separate startup server provided by PACX-MPI. The startup server has to be started before

the application instances. It then listens on a configurable TCP port for incoming connections. When the application processes are started, they send their setup informations to the startup server that collects them and sends all informations back to each cluster after the last instance is connected. With this the startup phase is complete and the startup server terminates. This approach is much more flexible, as the application just has to have the information where the server is listening and only needs to be informed about the number of intended application instances, which is a command line argument. At the same time, the PACX-MPI clients only need to be aware of the common interchange point given by the startup server.

A problem in the startup of the application that is not addressed by PACX-MPI is the need for a simultaneous execution on systems, where the execution is determined by a scheduler. As the involved clusters might each run their own, independent scheduler, the time slots assigned to the application might not overlap each other. This need for a co-scheduling in such an environment is an open issue and currently only overcome by manually adjusting the execution slots on all involved sites. For integrated heterogeneous systems managed by a single scheduler this problem does not exist and a simultaneous execution block can be found. However, such machine configurations are not very common.

Regardless of which of the two startup methods is used, the PACX-MPI clients read their information from a configuration file, that either describes the startup server to use, or provides detailed information on the other instances of the application. This configuration file has to be named `.hostname`. For a setup with a startup server this file has to contain just a single line:

```
Server <startupservername.startupserverdomain> <Rank>
```

Where the rank denotes the ordering of the client. The counting of clients starts with 0 and also defines the ordering of MPI ranks across all machines. Each client will hold a contiguous block of MPI ranks according to the number of processes used on that machine. The address `<startupservername.startupserverdomain>` describes the location of the startup server to connect to. Finally, the first term `Server` is a keyword indicating the usage of a startup server.

For a manual configuration without a startup server the configuration on all machines has to contain the definition of all participating PACX-MPI clients. This is achieved by the following line for each machine:

```
<hostname> <number of processes> (startup-command)
```

Here `<hostname>` denotes the hostname of the client and `<number of processes>` the number of processes on that machine. Finally, there

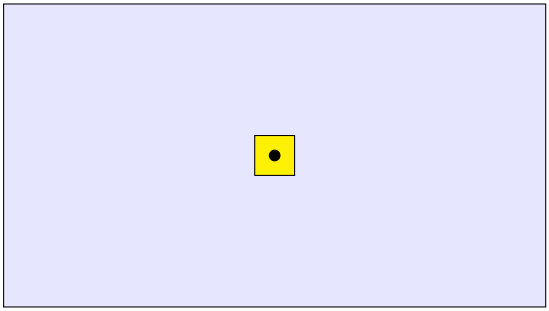


Figure 6.13: Domain setup for the scattering at a sphere. In blue the surrounding structured mesh is indicated. The black dot represents the sphere, and the yellow domain represents the unstructured mesh embedding the spherical geometry.

might be an optional `(startup-command)` given, that is to be used on the machine to start the MPI application. The ordering of MPI processes across the distributed system is given by the ordering of host definition lines.

6.3.6 Heterogeneous computations

To investigate the possibilities of a heterogeneous simulation on a heterogeneous supercomputing environment with the help of PACX-MPI, the NEC SX-8 vector machine at HLRS is used in combination with an IA-64 cluster. A 3D test case where a sinusoidal wave is scattered at a perfectly reflecting sphere of one meter radius is used to analyze the behaviour of the KOP framework in this setting. The exact solution for this problem is known and was proposed as an aero-acoustic benchmark by Morris for the *Second CAA Workshop on Benchmark Problems* [55]. The space around the sphere is discretized with an unstructured mesh and the Discontinous Galerkin scheme is used to solve the equations in this domain. This domain is embedded in a structured mesh where a Finite Differences scheme is deployed.

The unstructured mesh consists of 9874 elements, while the much larger space, covered by the structured mesh, is composed by around 42 million elements. In total, a volume of $102.2 \times 57 \times 57$ meters is covered in the simulation embedding the sphere with one meter radius. This configuration of domains is shown in Figure 6.13.

Table 6.1: Running times for the possible setups

	Itanium II	NEC-SX8	Both coupled
Elapsed time	27925 s	10966 s	3207 s
Unstructured	2994 s	7746 s	3019 s
Structured	23887 s	2871 s	2869 s
Coupling	1012 s	321 s	554 s
Waiting	0 s	0 s	164 s

The complete simulation was done on each architecture for comparison and then each domain was run on the appropriate machine in a coupled run. Due to the nature of the coupling a simultaneous execution of both domains is required. Ideally this would be ensured by the scheduler, however this feature is not available in the infrastructure of this setup. The simultaneous execution on both parts, therefore, has to be ensured manually. This can be done by starting the computation on the cluster interactively and submitting a job for the part on the vector machine to the scheduler. For the initialization of the PACX-Setup, a startup server running on the cluster is used.

The running times on each architecture and in the coupled run are shown in Table 6.1. Note, that the coupling of both systems involves a combination of resources and the elapsed time is lower than the sum of all parts because of this parallelization. By looking at the total elapsed time on each architecture, we see an advantage of almost a factor of three (10,966 s instead of 27,925 s) if the computation is done on the vector architecture instead of the scalar cluster. Thus, we could conclude that the vector machine is better suited for this simulation setup, that utilizes a large structured domain around the smaller unstructured domain embedding the sphere. But the table also highlights that the unstructured part of the simulation is consuming most of the time on the vector machine. We also observe that it is more than two times slower than the execution of the unstructured part on the scalar architecture, due to the bad vectorization of the highly irregular execution patterns in the unstructured part. Obviously, executing this part of the simulation on the vector system is not optimal, and the total running time could be improved by putting this part on the scalar system. The KOP framework with PACX-MPI allows us now to do this and the result is shown in the third column of Table 6.1. We see that for both the unstructured and the structured part, executed

on the respective best fitting architecture, the fast execution is almost achieved. Only small overheads, due to balancing are added in the solver parts, and each one roughly takes 3,000 s. The coupling is dominated by the unstructured domain. However, as the scalar system now does not have to compute the coupling parts of the structured part anymore. Thus, in comparison to the run on the scalar machine alone, the overall coupling time is also cut down. Due to the imbalances between both machines in terms of exection times for this setup, there is an additional waiting time that has to be spent in the coupling. This imbalance can be minimized by an appropriate balancing between the two machines. Unfortunately, such a balancing has to be done manually in current cluster environments, as an adaptation to the observed running times on either machine would require an increase or decrease of the node counts. Such a dynamic change of resources is generally not supported by the scheduling system. Nevertheless, the manual setup for a static coupling as considered here is still feasible and as can be seen, the benefits can be large. By offloading the unsuitable part of the simulation from the vector system NEC SX-8, we were able to cut the time to solution to one third of the execution on the vector system alone. It is remarkable, that this speed-up is larger than the addition of another vector processor, which would only be capable of cutting the overall running time into half.

The heterogeneous execution of the heterogeneous simulation enabled us to perform a simulation with 42 million elements in the structured domain faster than on either homogeneous system. This mesh size is large for the scalar system, and the coupled simulation achieved a reduction of the overall computing time by almost a factor of ten. The large reduction in computing time is due to the high efficiency of the structured mesh on the vector system. Direct addressing and highly regular execution patterns in this solver result in a full exploitation of the vector instructions. In the unstructured solver on the other hand this vectorization is not so easily achieved, and its execution is better left to the scalar system. However, in the aero-acoustic setup like the considered one with a large far-field in comparison to the flow field around obstacles, the computing resources nicely fit the numerical requirements.

6.4 Concluding Remarks on the Coupling Mechanism

In this chapter the implementation of the scalable distributed coupling mechanism and its advantages and deployment were highlighted. The coupling is designed to work with arbitrary domain configurations. While

this allows a large flexibility in the setup of domains, we also encountered a major scaling bottleneck in this approach. This scalability limit is imposed by the fact that nothing can be assumed about remote partitions and everything has to be communicated. Thus, an all-to-all communication pattern is inevitable. We discussed, how this can be solved on a distributed system and that the simulation can still scale to thousands of processes but the computation of coupling point locations across all processes remains expensive and even increases in costs with growing process numbers. In the end, we are always stuck with the need for such an all-to-all communication pattern if no further restrictions are imposed. From a broader point of view this scalability wall highlights the general observation that a flexibility in the execution introduces costs that will limit the computational efficiency. It might be argued, that scalability is not everything and a few thousand processes might suffice forever if the individual processors increase their capability. Unfortunately, the hardware development rather seems to trend towards the opposite direction and we will deal with smaller local resources in larger distributed systems. In the future we might need to deal with hundreds of thousands or even millions of interconnected relatively small and slow processor cores, similar to IBM's BlueGene system. The sketched coupling system enables already large multi-scale simulations on various systems and also the exploitation of distributed computing resources without major bottlenecks but it might not be well prepared for future computing developments.

Therefore, we move on in the following chapters and introduce a restriction on the meshes that will allow us to overcome not only the scalability limitation by the coupling but also the inherent problem in the mesh generation for unstructured meshes as discussed in Section 4.2. However, this major change necessitates a general redesign of all parts, involved in the simulation. A dedicated mesh format that enables not only its scalable handling in distributed solvers but also its scalable generation, requires a new mesh generation tool as well as a fundamentally adapted solver infrastructure. The basis of this infrastructure is implemented in a suite of tools that we refer to as *APES*.

7 Distributed Octree Mesh Infrastructure

The approach in this work so far has been to drive the distributed parallelization of the general coupling scheme for arbitrary geometrical setups as far as possible. However, it lead to the conclusion, that the approach is ultimately limited in its scaling, due to the necessary identification of coupling interfaces in the unknown partitioned space. While this might be tolerable in static setups, it will be a limiting factor in dynamic simulations with varying meshes. At this point, a data structure that supports fast spatial searches becomes necessary. One suitable data structure for this task is offered by the octree. The octree is given by the recursive binary subdivision of a given domain in each spatial dimension. It therefore allows for spatial lookups in logarithmic time complexity.

Though, the introduction of such a search structure for the coupling interface eliminates the scalability bottleneck in the connectivity identification of the coupling, we still face the limitation of unstructured meshes. In Chapter 4 the connectivity search for general unstructured meshes was moved from the solver into a pre-processing step. This results in a scalable solver, but it merely shifts the bottleneck. It helps to compute tremendeously larger problems, but still is ultimately limited in its scalability by the pre-processing.

The necessity for a fast spatial search to support the distributed coupling, raised the idea to turn this need into a primary foundation and use the data structure as underlying mesh in all coupled solvers as well. This eliminates the distinction between structured and unstructured domains, as there is just a single kind of domain, representable in the octree. Thereby the preprocessing step for unstructured meshes is replaced by the need for the generation of octree meshes. Due to the known topology of the tree, this generation can be achieved in parallel with a scalable distributed memory approach. However, this fundamental change in the concept, requires a redesign of the involved solvers and their implementation on top of the octree mesh concept. The implementation thus is done in a completely new framework. In the following the basic concept and design of this new framework, called *APES* (Adaptable Poly-Engineering Simulator) [65], will be outlined.

7.1 General Relevance of the Approach for Complex Geometries

Mesh-based algorithms build one of the most important discretization schemes for the numerical modeling of field problems. They are also well suited for parallel computations, as the meshes can be partitioned and the smaller problems solved concurrently. Their basic idea is the discretization of the computational domain by covering it fully with smaller, well defined elements. For complex geometries, it is in general necessary to use an unstructured mesh to discretize the computational domain. In unstructured meshes each element has typically a different size and form. Therefore, all information of element geometry and neighbor relations have to be described explicitly. This fully explicit description however, poses a severe hurdle for parallel processing, as potentially any part of the mesh might be required in the neighborhood of any process. Due to the limited memory, providing the complete mesh information locally to each partition is not an option, as the ability to process larger meshes is often the motivation to switch from one to many computing units in the first place. Although this limitation can be moved out of the solver itself, as shown in Chapter 4 with a preprocessing as offered by *GEUM* [40], the scalability bottleneck just gets shifted and ultimately limits the overall problem size. Tu et al. [81] pointed out the importance to avoid such bottlenecks throughout the complete chain of steps in a simulation.

Thus, not only the main computational loop, but the entire simulation process, has to be considered, including mesh generation and post-processing. Let us now look at a strategy to parallel mesh handling by stepping back from the most general unstructured mesh form to a little more restricted topology. As pointed out above, the octree offers such a topology. It builds a frame for global adressing but allows for a sparse mesh representation and greater flexibility than a structured, Cartesian mesh. In this sense, a mesh based on an octree is a middle ground between the rigid topology of structured meshes and the complete freedom of general unstructured meshes.

7.2 Octree Meshes in the Solvers

The octree is obtained by recursively bisecting a cubical domain in each dimension, such that it is subdivided into 8 smaller (child) cubes. This is then recursively applied to each (child) cube, resulting in an ever finer loalization. Choosing a cube as root element ensures a mostly isotropic

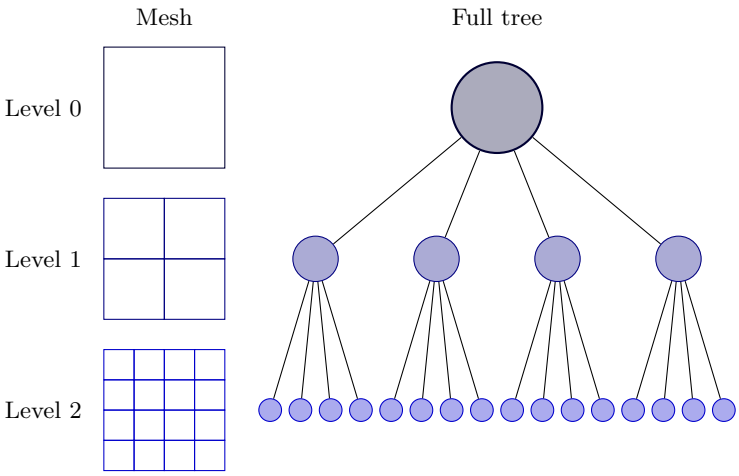


Figure 7.1: Illustration of the spatial bisection with the help of a quadtree. On the left, the mesh is shown for 2 refinements of the universe square at level 0. Right to it the full tree shows the relation of the mesh elements to each other as obtained by the repeated bisection.

behavior of the discretization. Note however, not the full cube of this root element will usually be used in the simulation. Instead we use a sparse mesh and explicitly store all elements that are to be included. The mesh, therefore, is treated like an unstructured mesh with explicit elements and their dependencies. As it is hard to follow 3D illustrations, we will instead mostly plot two-dimensional quadtrees, where the same concept applies. At the root of the quadtree there is a square, that is subsequently divided into four squares by a bisection of the root in each dimension. Figure 7.1 illustrates this concept with the actual mesh elements on the left, and the tree showing the topological relation on the right. Thus, this is exactly the same as for the octree, only that there are four instead of eight children.

Because this will appear commonly in the description, let us introduce the term level L for the refinement steps. We start at the universe cube or square and refer to it as level 0. Each bisection step increases the level by one, such that in 3D we will have 8^L nodes on level L in the full tree. Similarly for the 2D quadtrees we obtain 4^L. Note, how the localization rapidly narrows down within few iterations. This also enables the fast narrowing down of the neighborhood for any element in the mesh.

The usage of octree meshes for flow simulations has long been proposed, for example by Flaherty et al. [17]. However, the emphasis in this work is its special suitability for highly parallel distributed simulations. By using octree meshes the main bottleneck of unstructured meshes can be avoided, while the flexibility of local refinement and resolution of arbitrary complex geometries is maintained. Octree meshes offer, therefore, a compromise between easily distributed structured meshes and highly flexible unstructured meshes. The known topology in the octree keeps required global information low, and the connectivity between elements can mostly be computed locally. Though, it would be possible to deform the actual mesh elements and still benefit from the octree topology information, only cubic elements will be considered here. Cubic elements yield the advantage, that they usually can be computed efficiently in numerical schemes, or are even required, as for example in the Lattice Boltzmann Method. In the high order Discontinuous Galerkin solver, designed on top of this kind of mesh in *APES* this is exploited with an efficient dimension by dimension computation of integrals. Chapter 8 will present a possibility to describe high order geometry information within the cubical elements, reducing the need for deformed elements. The cubical elements also have the advantage, that they avoid issues with bad mesh conditions due to unfortunately formed elements. For these reasons, only cubical elements are pursued further in the following.

Figure 7.2 shows an example for a mesh, that is obtained by the described

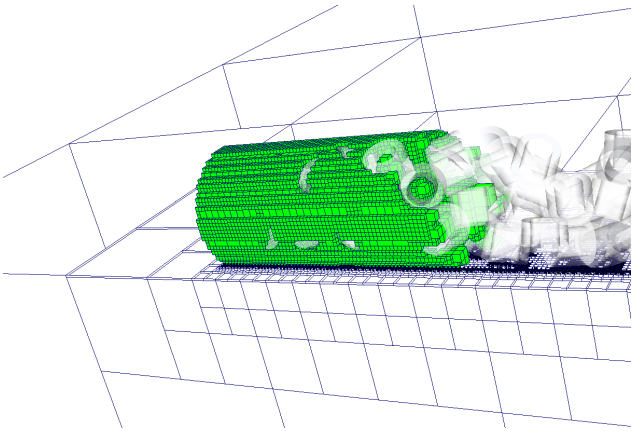


Figure 7.2: Illustration of a mesh obtained by the described octree discretization.

octree discretization. It shows the filling of a channel with many small cylindric obstacles, indicated by transparent objects. The complete space covered by the octree is outlined by the blue grid lines and shows the refinement towards the channel with obstacles. Inside the channel, there is the actual computational domain, shown by the green cubical elements. Only these elements are stored in the mesh and processed by the solver. The other elements indicated by the wireframe are only there to illustrate the refinement in the octree towards the geometry.

The basic concept in the *APES* framework is to exploit the known hierarchical topology, provided by the octree. Another concept that is supported by the octree and beneficial for the parallel operation is the strictly elemental view on the mesh data. As each elemental will be uniquely associated with a certain partition during the computation, such a view enables the parallel treatment right from reading or creating the mesh onward. The elemental view on the mesh data is therefore maintained in the data layout and the operations on the data.

7.3 Introduction of the Common *TreElM* Library

The octree handling is implemented as a common library, used by all tools in the processing chain of the simulation. Describing its main features,

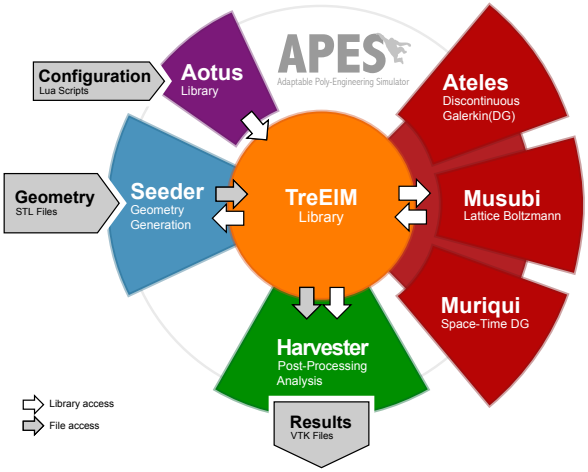


Figure 7.3: Schematic organization of the *APES* framework.

the tree-based elemental mesh is abbreviated as *TreElM*. It is used it as the central common component in the *APES* framework. The *TreElM* library is available online as open source [42]. This library enables a fully distributed tool-chain and opens the possibility of a highly scalable coupled simulation.

The tools in the framework range from the mesh generator *Seeder* [29], which is capable to produce meshes in the *TreElM* format, over the Discontinuous Galerkin solver *Ateles* for compressible and the Lattice Boltzmann solver *Musubi* for incompressible flows to the post-processing tool *Harvester*. Their interaction is sketched in Figure 7.3. The mesh generator not only produces elements in the octree, but also offers dedicated boundary descriptions for the solvers. In Chapter 8 we will have a look at the generation of high order geometry descriptions within the cubical elements of the mesh for *Ateles*. Along the same lines *Harvester* provides a dedicated post-processing that allows the visualization of high order simulation results by sampling coarse elements and showing fine details in the final visualization files.

Though, an integrated chain of tools is pursued, there is still the need to output large amounts of data from the applications point of view. Frequent

output of results is for example necessary in transient problems, but it is also required for restarts of long running simulations. Since neither input nor output (I/O) can be avoided to the largest part, even with an integrated workflow, the data structures are designed to be suitable for efficient parallel reading and writing. Another advantage of writing intermediate steps to the disk is the possibility to deploy each step in the chain of tools on the most suitable machine. This is especially useful for the visualization of the final output, that possibly depends on many libraries and benefits from special graphics hardware. Finally, reusing already generated meshes in multiple simulations minimizes the initialization effort. The format of the serialized sparse octree achieves this goal by using an element-wise organization of the mesh data. Before we turn to the implementation of this idea for distributed, parallel executions in Section 7.4, let us first introduce the fundamental concepts of the mesh representation. For illustration purposes a 2D quadtree will be used, but note that the concepts directly apply in the very same way in 3D for octrees.

7.3.1 Spatial ordering by space-filling curves

Figure 7.1 shows three refinement iterations for a quadtree, what is missing is the relation of the nodes in the tree (on the right) to the actual mesh elements (on the left). Namely, we need to find a rule to order the 4 children obtained by a bisection step. Depending on which rule is chosen, a specific ordering of the elements in the mesh arises. The curve obtained by traversing the elements in this order is a space-filling curve, providing a one-to-one mapping between multidimensional element coordinates and a onedimensional element rank. There are many space-filling curves [68] available to choose from, but the most relevant ones for octrees or quadtrees, are the *Hilbert curve* and the *Z* or *Morton curve*. The *Hilbert curve* has the nice property that a high locality can be preserved, as at most one dimension increases by one in the multidimensional coordinates, when increasing the rank by one according to the curve. A problem for this curve however, is raised by the need for a rotation of the element ordering depending on the refinement step or level in the tree. A less locality preserving but more efficient ordering is offered by the *Z curve* [56], where the spatial orientation of the curve does not depend on the level, and a transformation from multidimensional coordinates to a onedimensional ranking can simply be achieved by interleaving bits. Refer to Figure 4.2 for an illustration of this space-filling curve. The binary representation, as required for bit interleaving, is a direct consequence of the bisection mechanism that generates the tree. The bit interleaving technique for the

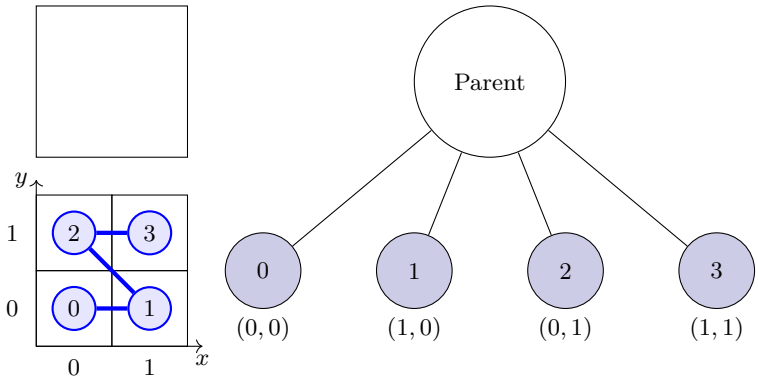


Figure 7.4: Relation of a parent to its four children in a 2D quadtree. The mesh elements are shown on the left, and the tree nodes on the right. The spatial arrangement of the four chilren is defined by the *Z curve* indicated by the blue line. Note, how the (x, y) coordinate pairs can be interpreted as binary representation of the rank along the *Z curve* by concatenating them like yx.

Z curve will be explained later in this section.

Let us have a look at this relation between a parent node and its four children in a quadtree in Figure 7.4, where the spatial arrangement of elements is defined by the *Z curve*. The rank of each element along the space filling curve starts with 0 and can be easily constructed from the integer coordinate pair (x, y), where the elements in each dimension are also counted from 0 onwards, as indicated by the axes in Figure 7.4. Note, that this mechanism works for arbitrary dimensions, and especially in 3D we obtain for the octree relations shown in Table 7.1.

Now that we have this rule in place, it is applied in each bisection step, such that on each level of the full tree we obtain a mapping between mesh element coordinates and the onedimensional rank along the space filling curve. For this, let us have a look at level 2 of the quadtree in Figure 7.1. Level 1 is just one refinement and is arranged as shown in Figure 7.4, now the same operation is performed for each of the elements from level 1 to obtain the 16 elements on level 2, and we obtain a mesh as depicted in Figure 7.5. The image provides three axes for each direction.

- A black axis to indicate the straight forward numbering of elements

Table 7.1: Construction of the space filling curve rank in a 3D octree.

(x, y, z)	zyx_2	rank
(0,0,0)	000_2	0
(1,0,0)	001_2	1
(0,1,0)	010_2	2
(1,1,0)	011_2	3
(0,0,1)	100_2	4
(1,0,1)	101_2	5
(0,1,1)	110_2	6
(1,1,1)	111_2	7

in each dimension along with the binary representation of this integer coordinate value

- A blue axis labeled *L1*, indicating the coordinates in the first refinement of the universe square in level 1.

- A red axis labeled *L2*, indicating the child coordinates within the the elements of the first refinement, following the rule set out in Figure 7.4.

Note, how the coordinate value can be constructed by concatenating the bisection steps as bits, starting with the first refinement for the most significant bit. Thus, the coordinate $x(L)$ on a given level $L > 0$ can be interpreted as the traversal of the bisection tree and with the child coordinate ξ_l in each iteration, we obtain

$$x(L) = \sum_{l=1}^{L} \xi_l \cdot 2^{L-l} \tag{7.1}$$

Where l iterates over the levels from the root node to the leaf on level L. For example, we arrive at coordinate 2 by starting with $\xi_1 = 1$ and moving to $\xi_2 = 0$, resulting in $10_2 = 2$. Now we can find the *Z curve* rank in a multidimensional mesh by interleaving the bits of the individual coordinates, i. e. grouping the bits of same significance from all coordinates together and building a single value by concatenating these bit sequences. In d dimensions we get d coordinate components x_i, which each can be

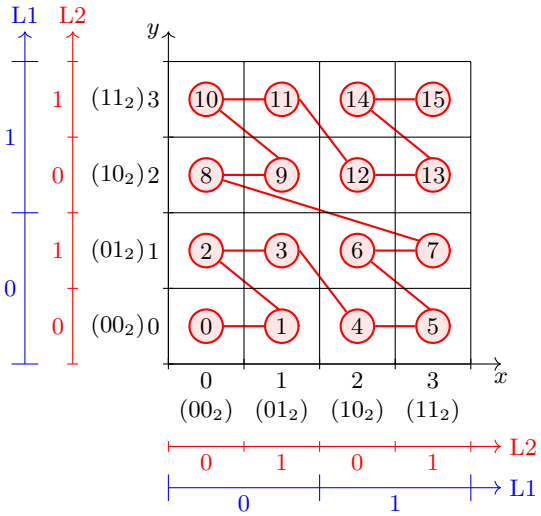


Figure 7.5: Coordinates and rank along the *Z curve* after two refinements of the universe square, compare to the second level in Figure 7.1. Note, how the coordinates in binary can be obtained by concatenating the bisection position in each refinement, starting with the first level (here in blue L1) for the most significant bit. The refinement of the first refinement is here drawn in red and labeled L2 to indicate the second level. The rank of each element within the *Z curve* than can be found by interleaving the bits of the coordinates in each direction.

represented as in equation (7.1) resulting for a given level L in

$$x_i(L) = \sum_{l=1}^{L} \xi_{i,l} \cdot 2^{L-l} \quad \text{for} \quad i = 1, .., d. \tag{7.2}$$

The *Z curve* rank $z(L)$ on this level L can then easily be computed by

$$z(L) = \sum_{l=1}^{L} \left(\sum_{i=1}^{d} \xi_{i,l} \cdot 2^{(L-l)d+i-1} \right). \tag{7.3}$$

As an example consider the quadtree of Figure 7.5. Applying (7.3) for the element at coordinate $(x_1, x_2) = (x, y) = (2, 1) = (10_2, 01_2)$ in this $L = 2$ mesh is equivalent to the following procedure: Group the most significant bit from the y coordinate and the most significant bit from the x coordinate together, yielding 01. Then concatenate this group with the grouping of the next significant bits, which in this example are already the least signficant bits (10). This procedure results in a rank $z(L)$ of $0110_2 = 6_{10}$ along the *Z curve* on level $L = 2$. For the octree in 3D, there will always be $d = 3$ bits grouped together (inner sum in (7.3)), but otherwise the same mechanism applies.

The reverse operation can be found by matching coefficients. Writing $z(L)$ from equation (7.3) as

$$z(L) = \sum_{k=0}^{L \cdot d - 1} \zeta_k \cdot 2^k,$$

we find matching coefficients for

$$k = (L - l)d + i - 1. \tag{7.4}$$

Thus, the l, i pair with l as bit position in a coordinate component x_i from (7.2) matching a given bit position k in the *Z curve* rank $z(L)$ on level L can be found by

$$l = L - \left\lfloor \frac{k}{d} \right\rfloor, \tag{7.5}$$

and

$$i = (k \mod d) + 1. \tag{7.6}$$

In equation (7.5) $\lfloor \cdot \rfloor$ denotes the integer floor rounding to the nearest smaller integer number. Note, that $i - 1 < d$ for all i in (7.4), and can therefore be neglected in (7.5), due to the integer division by d. With equations (7.4), (7.5) and (7.6), it is now possible to match values at the respective bit positions in *Z curve* rank and the coordinate component $\zeta_k = \xi_{i,l}$ in both directions. Though, for the actual implementation some bit shifting operations can be used instead of multiplications and divisions to achieve the bit interleaving and sieving, respectively.

7.3.2 Node identification in the tree

The space filling curve provides us now with a method to map the multiple dimensions on a given level into a single number with a well defined ordering. However, the identification of an arbitrary element in the full

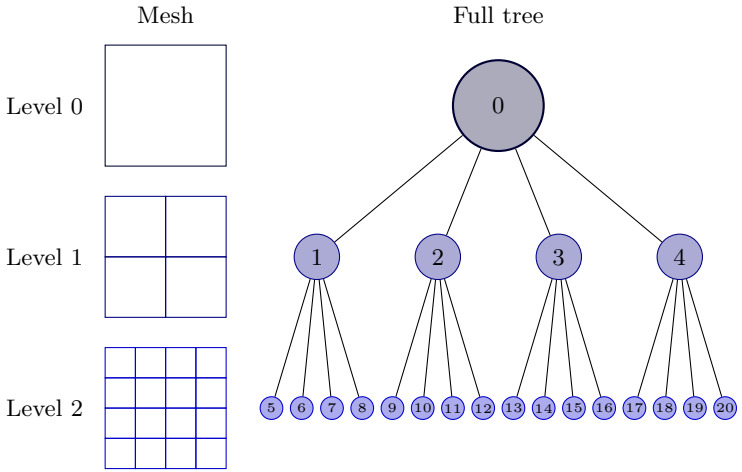


Figure 7.6: Illustration of the *TreeID* to identify each element in the full quadtree with a single integer.

tree still requires two information, the level, the element is found on and its *Z curve* rank on that level. To avoid this need for two numbers to fully describe an element, we introduce a breadth first counting through the complete tree. This counting follows the space filling curve on each level, but keeps adding adding up all elements from level to level. This reduces the address of an element to a single integer value, out of which the position and size can be computed completely. The numbering starts with 0 for the root node, representing the universe square or cube and then follows the space filling curve on each level, but keeps adding up the respective ranks. As this number provides an identification for each element in the tree, it is referred to as *TreeID* in this work. Figure 7.6 shows the quadtree from Figure 7.1, but now with the respective *TreeIDs* added to the elements and nodes. With this method it is possible to fit 20 levels of a 3D octree into a signed 64 bit integer.

The breadth first numbering of the full tree results in a straight forward method to move vertically through the tree. Consider a given node with the *TreeID* t_n. Its children can be found by

$$t_{cj} = 2^d \cdot t_n + j, \tag{7.7}$$

where j indicates the child ranging from 1 to 2^d, and d is the dimensionality of the mesh (3 for the octree, 2 for the quadtree). As explained in the previous section, the spatial arrangement of the children $t_{c1..2^d}$, follows the *Z curve*. Refer to Figure 7.4 and Table 7.1 for an illustration of this ranking in a quadtree and an octree. The ordering for arbitrary dimensions can be found by using $L = 1$ in equation (7.3). Conversely, the parent t_p of a given node t_n can simply be found by

$$t_p = \left\lfloor \frac{t_n - 1}{2^d} \right\rfloor . \tag{7.8}$$

The horizontal movement within a single level of the tree can be achieved by transforming the *Z curve* rank to the multidimensional coordinates via equations (7.5) and (7.6). Then, after applying some arbitrary offsets to obtain a relative neighbor in multiple dimensions, convert the modified coordinates back to its *Z curve* ranking via equation (7.3). What is left to do is the transformation from the *TreeID* to the *Z curve* rank on the level of the element. This is achieved by subtracting the number of elements in the full tree on all coarser levels:

$$s(L) = \sum_{i=0}^{L-1} 2^{d \cdot i} \tag{7.9}$$

The offset s given in equation (7.9) yields the connection between *TreeID* t of a given element and its ranking in the *Z curve* on its level L as

$$z(L) = t - s(L). \tag{7.10}$$

Now we have all the tools at hand to navigate around in a mesh defined on the octree basis. Let us now put this into action for the connectivity search, as it is needed for stencil schemes like finite volume or the Discontinuous Galerkin.

7.3.3 Connectivity search in the Octree

With the node identification at our hands, we can perform lookups of arbitrary elements in the full tree. That is, for any given *TreeID*, the adjacent elements with their respective *TreeID* can be computed. This can be used to compute the stencil as required by the numerical scheme, by exploiting the fully structured Cartesian nature of the mesh on each level. To find neighboring elements in the required stencil, we need to convert the *TreeID* of the element, for which the stencil is to be constructed, to

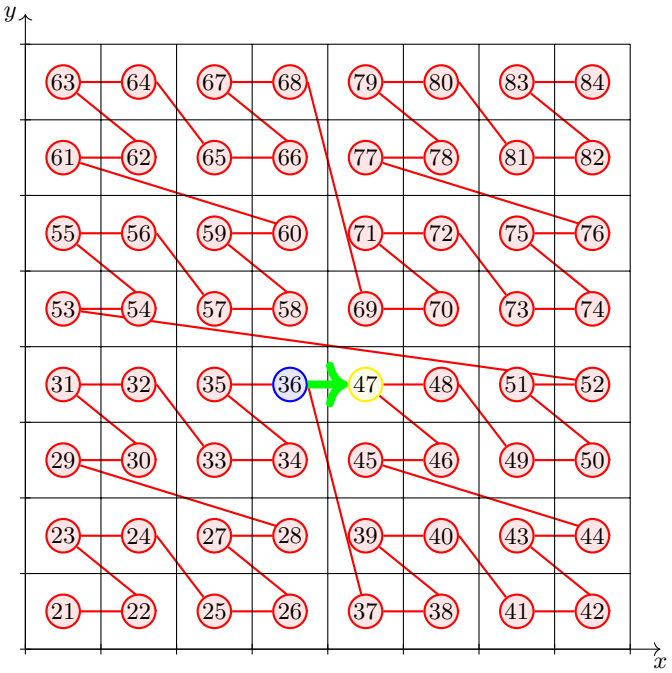


Figure 7.7: Looking up the right neighbor of a given *TreeID* (31), highlighted in blue. The neighbor is indicated in yellow and has a *TreeID* of 47. These elements are on the third level of the full quadtree and the first *TreeID* on this level is 21, all following elements on this level are counted according to the space filling *Z curve*.

its coordinates on its level. Then we can add desired offsets and convert the obtained coordinates back to a *TreeID*.

Let us illustrate this with an example for the two-dimensional quadtree. Assume, we need to find the right (in positive x direction) neighbor to the element with *TreeID* $t = 36$, shown in blue in Figure 7.7. First, we need to identify its coordinates $x_i(L)$, which includes the level and the respective coordinates in the two dimensions. The level L can be found with the help of equation (7.9) by subtracting the level offsets successively from the given *TreeID*. Thus, in this case for $t = 36$, we obtain:

- Level 0: $36 - 4^0 = 35$
- Level 1: $35 - 4^1 = 31$
- Level 2: $31 - 4^2 = 15$
- Level 3: $4^3 > 15$
- *TreeID* 36 is located on level 3 and has a *Z curve* rank of $z(3) = 15$ on this level.

The binary representation of $z(3) = 15$ is 001111_2 and with the bit sieving from equations (7.5) and (7.6) we obtain the coordinates for both dimensions as $x_1 = 011_2 = 3$ and $x_2 = 011_2 = 3$:

$z(3) = 15$	0	0	1	1	1	1
$x_2 = 3$	0		1		1	
$x_1 = 3$		0		1		1

To find the right neigbhor, we simply need to add one to x_1, which yields 4. The coordinate, for which a *TreeID* has to be found, therefore is $(4, 3)$ on level 3. To convert this back into a *TreeID*, we revert the steps taken above, but now for the new coordinate. First, we convert the coordinates $x_1 = 4$ and $x_2 = 3$ into a *Z curve* rank $z(3)$, by interleaving their bits, according to equation (7.3):

$x_2 = 3$	0		1		1	
$x_1 = 4$		1		0		0
$z(3) = 26$	0	1	1	0	1	0

From this interleaving we obtain $z(3) = 26$ as shown above. Finally, we need to add the offset $s(L)$ from (7.9) for level $L = 3$ to obtain the *TreeID*: $t = z(3) + 21 = 47$. Note, that to find this neighbor with *TreeID* 47 to the element with *TreeID* 36, we did not need any more information about

the mesh then the dimensionality of the mesh and the original *TreeID* itself. All topological information is encoded in the *TreeID*. This ability to identify the neighborhood purely based on a given *TreeID* is important for the parallel treatment, which will be discussed later on. One question remains, before we move on to discuss the sparse tree in the next section. When looking at neighbors, it might happen, that we try to access elements outside the universe cube, for example the right neigbhors to elements at the right edge of the mesh shown in Figure 7.7, that is to elements with an x_1 coordinate of 7. To avoid special treatments and always ensure a well defined behavior, a periodic boundary is assumed in each direction of the root element. This is achieved by using a modulo operation around the offset addition for the neighborhood identification, such that we get for the right neigbhor $x_1 = (7 + 1) \mod 8 = 0$. With this definition, we obtain a closed system, where any movements through the tree remain within the tree and result in well-defined element locations. Please note, that this periodicity is merely assumed to assure a well defined neighborhood in the topology of the tree. We will treat boundary conditions in the mesh differently allowing for their arbitrary definition.

7.3.4 The sparse Octree

So far, only the full tree was considered, and the description of the topology, as given by the recursive multidimensional bisection, has been described. The topology allows the identification of each element without a fully explicit description. However, in the meshes we only want to maintain the elements that build the computational domain. These are always leaf nodes in the tree without children. Also not all areas of the universe cube need to be covered by the computational domain. Instead boundaries and obstacles might limit it. Different resolutions in the computational domain itself also give rise to missing nodes on given levels. In this mesh definition which is sparse in contrast to the full tree of the topology description we need to maintain a list of all elements and need to deal with neighbors that are possibly missing.

What we are looking for, are tessalations of arbitrarily shaped domains with non-overlapping elements of variable size. That is, for any element in the mesh, we will have neither its parent nor its children from the tree, as those would overlap the same space. Further, to allow for domains of arbitrary shape, we restrict the elements to those, that actually form the computational domain. Figure 7.8 shows such a mesh, which is refined towards a cubical obstacle in the middle of the domain. The discussion above provided us with a unique identifier for each element in the domain.

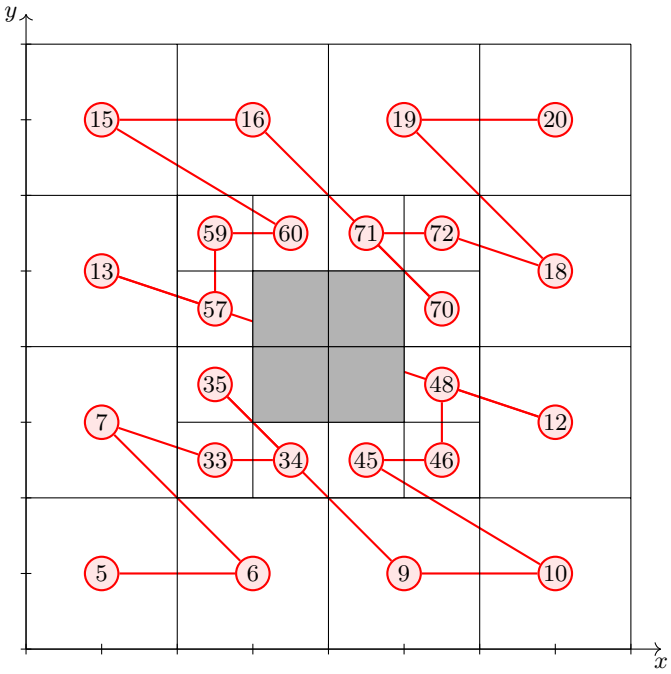


Figure 7.8: Quadtree mesh refined towards an obstacle in the center (indicated in grey). The numbers indicate the *TreeID* for each element.

To describe a mesh, it therefore is sufficient to simply list the identifiers of all the elements, which should assemble the mesh. Thus, a sparse mesh for a complicated computational domain can be obtained noting the *TreeID* for each non-overlapping element that should be part of the mesh discretization.

The mesh in Figure 7.8 might not be complicated, but serves as a basic example, showing elements of different sizes and areas in the universe square, that are not to be part of the computational domain. Shown is a quadtree mesh refined towards an obstacle, along with the *TreeID* of each element. Sorting of the elements is indicated by the red line, though some parts of the resulting curve overlap with each other, like the connection

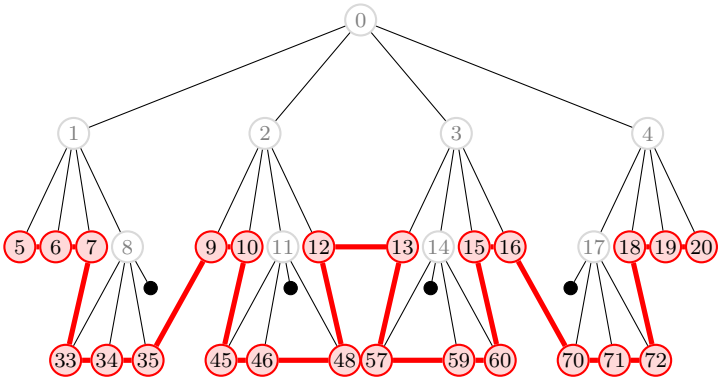


Figure 7.9: Tree to the mesh from Figure 7.8. Red nodes are actual elements in the mesh, while the grey nodes are virtual parents that only exist topologically. The black dots indicate missing elements, where an obstacle is found and no elements are present. The thick red line indicates the space filling curve and the ordering of the elements.

from 12 to 13 overlaps with the connection from 48 to 57. This mesh can be completely described by the list of the shown *TreeID*s and a geometrical definition of the universe square. The corresponding tree is shown in Figure 7.9. In the tree representation of the mesh in Figure 7.9, the connecting curve from element to element gets a little clearer, without the overlapping of the spatial domain.

Figures 7.8 and 7.9 are closely related. While Figure 7.8 shows the spatial arrangement of the elements, Figure 7.9 illustrates the topological arrangement in the quadtree. The relation between these two is defined by the space filling curve. Thus, a sparse mesh is represented by listing all *TreeID*s for the mesh, sorted according to the space filling curve, in the case of Figure 7.9 all numbers in red nodes, following the thick red line of the space filling curve. The obtained list in this example is (5,6,7,33,34,35,9,10,45,46,48,12,13,57,59, 60,15,16,70,71,72,18,19,20).

Finding neighbors in this sparse mesh works basically by the same mechanism as described in Section 7.3.3 for the full tree. Namely, the *TreeID* is converted to coordinates, an offset for the neighbor is added, and the resulting coordinate converted back to a *TreeID*. There is only

one additional step involved, which is the look up of the *TreeID* identified as neighbor in the list of elements of the mesh. For this search, we have two possible approaches at our hand, both resulting in logarithmic time complexity for the lookup of a given element. One approach is to create the virtual structure of the upper part (all parents of actual leaf nodes) of the tree with pointers to the children. The tree could then be traversed accordingly to fastly identify the element in question. An alternative is the usage of a binary search in the list of elements, which is possible, as the elements are sorted according to the space filling curve. The comparison operator, required for a binary search, is given by comparing to elements on their greatest common level. Both approaches offer a fast lookup of actually existing elements. Though the actual storage of the upper part of the tree might be faster, it also requires the mainteanance of a large part of the full tree. The binary search on the other side converges somewhat slower, but has the appeal to work for arbitrary dimensions in the same way and without the need to maintain an explicit tree structure. In the mesh shown in Figure 7.8, the lookup of the upper neighbor of the element with *TreeID* 48, for example requires the identification of the neighboring *TreeID* 70 as described in Section 7.3.3 and then the lookup of the position of this element in the list of all elements in the mesh, which in this case is 19.

One complication in the sparse mesh in comparison to the full tree is the possibility of neighboring elements not existing on the same level as the original element. For example the right neighbor of the element with *TreeID* 70, would be *TreeID* 73. However, this element does not exist, instead only the element with *TreeID* 18 exists. The left neighbor does not exist at all, due to the obstacle in this direction. Finally, the left neighbor of *TreeID* 18, would be 17, which also does not exist. These are the three possible cases, that can occur in the same level neighbor lookup in the sparse tree.

1. The element does not exist (outside the computational domain).
2. Only children of the element exist.
3. Only a parent of the element exists.

The first case does not pose a problem, because there are usually boundary conditions associated with the surface of the computational domain, and therefore no need for elements in this direction. The last two cases need to be treated by interpolation. For this, intermediate elements are created, which only serve as placeholders for the interpolation results, such that

the stencil algorithm can easily access them on each level. In contrast to the mesh elements, the intermediate elements overlap other elements, and they are created as needed. The actual interpolation mechanism needs to be provided by the solver, but the required infrastructure and spatial information is given by the octree layout. Let us refer to these intermediate, newly created elements as *ghost* elements.

Depending on the origin of the interpolated data, we may call these *ghost* elements more specifically *ghosts from finer*, for elements where the actual mesh elements are children of the element, or *ghosts from coarser*, for elements where only coarser elements exist in the actual mesh.

7.4 Distributed Octree

With the previous tools at our hand, we now have the means to describe complex meshes on the basis of an octree topology. In this section, we will uncover how this strategy can be exploited on parallel systems with distributed memory. The space-filling curve used to serialize the mesh provides an ordering that preserves the multidimensional locality in the onedimensional list to some extend. A partitioning, therefore, can be simply achieved by splitting the list of elements into chunks of equal size. Due to the locality preserving property of the space-filling curve, these chunks build partitions of reasonable shape. While the surface might not be as small as with a graph partitioning approach, it yields a sufficiently small surface, especially for high order methods.

For the illustration of the mechanism in parallel, we will use the quadtree mesh from Figure 7.8 and distribute its $N = 24$ elements to $P = 5$ partitions. Each partition has to contain at least

$$n = \left\lfloor \frac{N}{P} \right\rfloor = 4$$

elements. The remaining 4 elements are distributed to the first 4 partitions, such that the first four partitions each consist of 5 elements, while the fifth partition only has to compute 4 elements. With this distribution, there is always at most a difference of one in the number of elements between any two partitions.

An important property of this partitioning approach is the fact that each process locally can decide which part of the mesh it should compute. The only required information is the total number of elements in the mesh N and the total number of processes P. As the mesh is stored in the described linear array, each process can read just its part from it independently from the other processes.

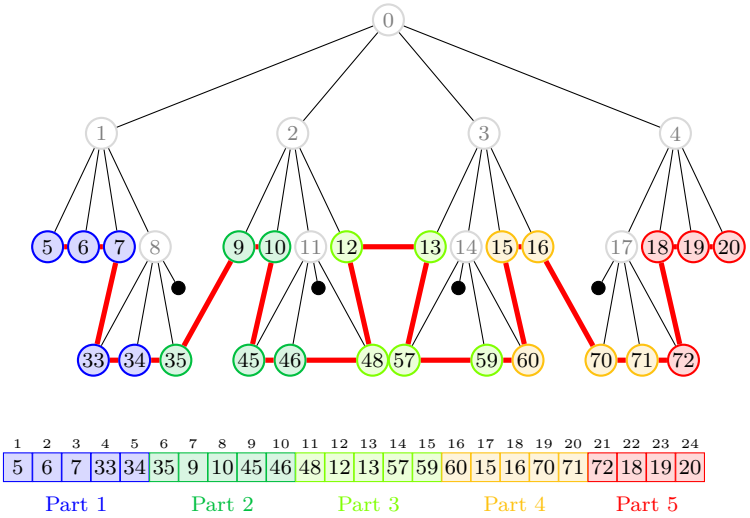


Figure 7.10: Partitioning with 5 parts of the mesh from Figure 7.9. Colors indicate the partition, each element belongs to. Below the tree, the serialized list of elements and their split into partitions is shown.

7.4.1 Distributed connectivity search

The connectivity search in parallel works basically as described in Section 7.3.3. There is only one major complication compared to the serial sparse octree. In addition to the three cases for neighbors outlined above (not existing, on a coarser or a finer level), it is possible in the distributed mesh, that an element is found on a remote partition. As a searched element might be refined further, and only its children exist in the actual mesh, it may also be that the element is spread across multiple partitions. Consider for example the element with *TreeID* 59 in Figure 7.8 with the partitioning outlined in Figure 7.10. The right neighboring element would be the one with *TreeID* 60, which is found on a different partition. For an example, where the searched element is distributed across multiple partitions, consider the right neighbor to *TreeID* 13, which is the not existing, but refined element 14. Element 14 has three children $(57, 59, 60)$, of which the first two are located in partition 3 and the last one is located on partition 4, as can be seen in Figure 7.10.

Thus, there is one additional step involved in the creation of the connectivity in the distributed tree: locating the elements in the partitions. To allow this, the first and last element of each partition is acquired on each process. The identification of the partition for a given *TreeID* can then be done by a binary search across those partition boundaries. With the partition bounds known to each process, this can be done completely independent by all processes concurrently. As a result of this identification, the following three cases can arise:

1. The searched element is *found locally* and the serial method outlined in the previous section can be applied.

2. The searched element is *found on a remote process* and has to be communicated.

3. The searched element is *found on multiple processes* that is, it does not actually exist, but is further refined.

Elements that do not exist at all because they are outside the computational domain are not taken into account here. Usually this is not an issue as there are boundary conditions at the surface of the domain and no neighbors are searched in this direction. In other words, we never expect to look for elements outside the computational domain. The first and second case from the partition identification are straight forward. If the element is local, the algorithm described before for the serial identification of the element position in the sparse mesh can be followed. In the second case,

when the element is on a remote partition, a request has to be sent to the corresponding process and a *halo* element is created for the solver locally. The remote process itself simply performs the serial lookup of a given *TreeID* in its part of the mesh and establishes a connection of the corresponding mesh element and the communicated element.

Only in the third step it might not be obvious, how the situation should be dealt with. However, by relying on the recursive topology of the octree, this case also can be resolved with little extra work. First, it should be noted, that this case can only happen if the requested element is further refined and only its children exist in the sparse, non-overlapping mesh. For these, a virtual element, serving as a placeholder, needs to be created as in the serial algoritm. When the children are all on a single partition, this virtual element obviously should reside on the same partition itself. However, in the case where the children are spread across multiple partitions, it is not obvious anymore where the virtual element should reside. Instead any process encountering this situation creates the virtual ghost element itself. While this might result in multiple processes executing the interpolation and filling of the ghost element, it avoids the need for some agreement algorithm with additional communication to find a suitable process for this task. Thus, if the third case above is encountered, the process looking for the distributed element, creates this element locally as a placeholder and then looks instead for its children. This is done recursively until none of the children is distributed across more than a single process anymore. It should be noted, that this case is rather rare for most mesh configurations, and children of an element can only be distributed across more than two partitions if some are enclosed completely within the searched coarse element. Usually, there are only two partitions involved and the recursion above has only to be done once. Yet, by employing this algorithm, it is possible to deal with any mesh and partitioning configuration without any special cases.

The most important features of the presented approach are (i) the little information, that is needed to describe remote partitions, and (ii) the mostly independent local computation of relations between elements in the mesh. In terms of *MPI* operations the complete communication in the neighborhood search algorithm is composed of the following steps:

- An allgather to collect the first and last *treeIDs* from all partitions.
- An all-to-all with a single integer to indicate the number of elements to be exchanged between all processes.
- Point-to-Point communications for the actual exchange of elemental

data.

7.4.2 Scalability measurement

For the assessment of the presented approach, the time consumption of the initialization in the solver is measured. To stress the applicability for large stencils, a neighborhood of 18 elements in 3D is searched, as for example required in many Lattice Boltzmann methods. The computational domain is a cube, periodic in all directions, and allows for an ideal weak scaling, as each partition has exactly the same workload, both in computation and communication, for a suitable number of processes.

These measurements where done on the Cray XE6 system *Hermit* located at the High Performance Computing Center HLRS in Stuttgart (Germany). It is equipped with *AMD Interlagos* processors, which are grouped in shared memory nodes with 32 cores each. A pure *MPI* parallelization with one process per core is used, resulting in 32 processes per node. The analysis ranges from a single node up to 1024 nodes. Thus, covering a range of up to 32768 processes. Due to the full cubes (uniformely refined universe cubes to a certain level), deployed in this assessment, the total element counts of all conducted runs are powers of 8. The largest evaluated problem had roughly 8.5 billion elements.

Due to the described binary search across partition boundaries, a running time scaling, that asymptotically behaves like $\mathcal{O}(\frac{N}{P} \cdot \log(P))$ is expected, where N denotes the overall number of elements and P the number of processes. To see how well this is achieved, two measurement series are performed. The first is a strong scaling, with fixed overall problem size N, which should scale up to the point, where the $\log(P)$ term or some constant overheads dominate the $\frac{N}{P}$ term. Whereas in the second series, a weak scaling with a fixed problem size per process $\frac{N}{P}$ is investigated.

7.4.3 Strong scaling

In the strong scaling analysis, the largest cube, that fits onto a single node is used and the measurement is run for this mesh on more and more processes. The problem has 8^8 or roughly 16 million elements. In Figure 7.11 the resulting running times for the identification of all neighbors are plotted in seconds with logarithmic axes. This initialization step scales quite fine up to 8192 processes. Beyond this process count, the the running times get larger again. Yet, even for these small problem sizes per process of less than 2000 elements, the parallelization overhead is not large and the running time remains below one second. The observed peak at 2^{14}

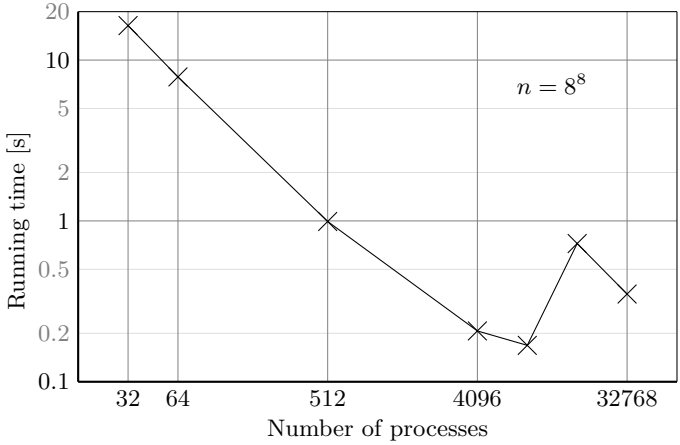


Figure 7.11: Strong scaling of neighbor identification on *Hermit*.

processes might be due to the less optimal partitioning and therefore an increased number of *halo* elements. However, these times still remain on a low level at less than one second. The strong scaling shows that it is possible to speed up the neighbor identification by a factor of at most roughly 100. As we will see from the weak scaling, this translates also to higher node counts and any given problem size can be speed up by parallelization by this factor of roughly 100.

7.4.4 Weak scaling

For the weak scaling, about 8 million elements per node are used. To ensure exactly replicated problems on each process, only powers of eight are used for the process counts. Thus, the smallest machine partition in this analysis are two nodes or 64 processes. The measured running times are plotted in Figure 7.12 over the same axis for the process counts as the strong scaling. Note, that the timings here are not plotted with a logarithmic scale, the linear increasing running time therefore indicates a $\mathcal{O}(\log(P))$ behavior. This corresponds to the expected behavior for the binary search over all partition boundaries.

The initialization thus does not build a bottleneck for the overall simulation and enables the computation of large problems. This scalable

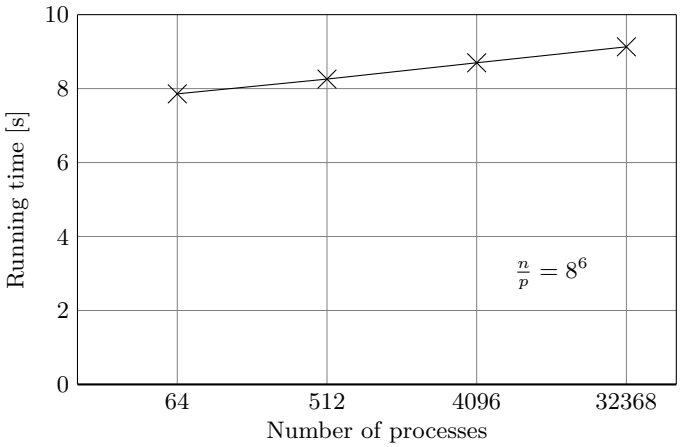


Figure 7.12: Weak scaling of neighbor identification on *Hermit*.

behavior of our new initialization enabled the connectivity creation for 8.5 billion elements in less than 10 seconds with 32768 processes on the *Hermit* system.

7.5 *TreElM* File Format

As already introduced in the previous sections, the octree mesh is represented by the serialized leaf nodes building the computational domain without overlap. The storage of the mesh on disk is basically the sequence of *TreeID*s building the computational domain. However, this is in general insufficient to fully describe the mesh and an additional elemental information is introduced to attach further data to the elements. 64-bit signed integers are used to encode the *TreeID*s, which ensures a great portability, but could be extended to 128-bit signed integers, when needed. In addition to the *TreeID*, a second 64-bit signed integer is used to build a bit-mask for properties like boundary conditions, that might be attached to each element. Each bit in this second integer indicates, whether the corresponding element possesses a certain property or not. For example one bit indicates, wether the element is adjacent to boundaries and further information on boundary conditions are attached to it. This data is written

in native binary format to disk, resulting in a file with just 16 bytes per element. Due to this straightforward format, the data might be accessed on an elemental basis with Fortran's direct IO, it can be easily converted between big and little endian representations and allows fast reading and writing.

This binary data is accompanied by a header file with descriptive information about the mesh, like the total number of elements, the physical extent and origin of the universal cube and descriptions for the attached properties. The data in this header is provided in the form of a *Lua* [35] script, to allow for a flexible handling of additional data and the future evolution of the format. With the help of the *Aotus* library [41] data can be easily retrieved from and written to such *Lua* scripts.

7.5.1 Additional elemental properties

As 64 bits are available to describe additional properties, there could be 64 different kinds of data attached to each element and each one combinable with all other properties. The most important data that has to be attached to elements in nearly every mesh are the boundary conditions, others are for example material information. As the boundary conditions are such a basic part of the general mesh description, let us use their implementation as example with greater detail. But keep in mind, that the described method is also suitable to attach other information to the elements. In Chapter 8 for example colors and subresolution information for the elements are introduced. Their attachment to the mesh follows the same principles as described in this section for the boundary elements.

Each property is indicated by a bit in the property bit-mask. Which bit is attached to what kind of property is defined in the header file of the mesh. The property might then be linked to further information stored in additional files. To attach boundary conditions to an element, a bit is set for an element to indicate that at least one of its sides has a boundary condition attached to it. For all elements with this property and only those elements, boundary conditions in all directions are stored. This data is put into a separate file. The information on boundary conditions can therefore be found in this additional binary file for all directions of every element with any boundary condition attached to it. The binary file has the same number of integer entries for all elements and can be easily accessed by distributed processes, as element locations in the file can be computed solely by the position of the boundary element. In this list of boundary elements, the ordering from the complete mesh is preserved. That is, while only elements with the boundary property are stored in the file, they are

still sorted according to the space filling curve.

A boundary condition is indicated by a positive integer number. But to ease the handling of boundaries, there is a header file provided with a list of labels corresponding to the integers. These labels can be used in the configuration of the solver to assert actual properties to the boundary conditions. Only positive integers are considered as boundary entries, zeros indicate no boundary in the corresponding direction, and negative numbers are reserved for references to *TreeID*s as neighboring elements. Directly referencing a certain *TreeID* instead of a boundary condition on one of the sides of an element is used to implement periodic interfaces within the universe cube. A zero on the other hand indicates no boundary condition in that direction. Due to this layout of the boundary description, the boundaries can be read in parallel with the following procedure:

- Count the local number of elements with boundary conditions (given by the property bits defined for all elements).
- Build a prefix summation across all partitions to find the offset of the local set of boundary conditions in the sparse set of global boundary conditions.
- Read this set of elemental boundary conditions from the file accordingly from the offset up to the number of local elements with boundary conditions.

With the strict ordering and uniform elemental treatment of data, the sparse property information can be accessed efficiently in parallel. Just a single collective prefix operation is required in the parallel system, while the consumed disk storage is kept at a minimum.

The amount and kind of additional data that might be attached to the elements by the properties is arbitrary and could involve several more indirections. For example, in the representation of colors and subresolutions, there needs to be a file with subresolution data for each color.

7.6 Overview to the Implementation of the APES Framework

With the methods described in this chapter, we now have the utilities at hand to efficiently describe meshes on the basis of the octree topology. The algorithms are implemented in the *TreElM* library [42], but of course this is only one building block towards a simulation tool that allows the computation of large scale problems on highly distributed parallel systems.

The layout of the overall framework with the central *TreElM* library is shown in Figure 7.3. Following components build the *APES* framework:

- *Seeder*, the mesh generator. This application creates mesh files in the *TreElM* format from geometry description. A discussion on the generation of high order representations within this mesh generator is provided in Chapter 8.

- *Ateles*, a Discontinuous Galerkin solver, dedicated to cubical elements.

- *Harvester*, a post-processing tool to visualize high order data from *Ateles*.

In this section, we will now have a brief glimpse at these other parts of the *APES* framework and how they interact with each other. The goal is to provide parallel tools for each necessary step of the overall simulation. Tu et al. refer to this as an end-to-end parallel simulation [81]. It should be noted, that the *TreElM* library not only provides the means to identify the halos to be exchanged between partitions but also provides various communication patterns that can be used for the actual exchange. Due to this encapsulation of the communication, the communication layout can be easily exchanged. It is even possible to replace the complete parallel paradigm and for example employ Fortran Coarrays instead of MPI with only few changes to the code.

The first tool in the chain, covered by *APES*, is the mesh generation tool *Seeder*. It generates meshes in the *TreElM* format, as described above. Input for the mesh generator are geometrical descriptions of the domain boundaries in the form of STL files. An octree mesh approximation of this geometry is found by iterative bisections towards the geometry, up to a user defined refinement level. For the identification of the computational parts, the user has to define seeding points from which the domain will be flooded. Only flooded elements are considered to be part of the computational domain and will be written to the mesh file. We will look into mesh generation in some more detail in Chapter 8, though with a focus on the high order representation of geometries. A more general description is given in [29].

Ateles then takes up the generated mesh and uses it to perform simulations of partial differential equations. Several equation systems have been implemented in this solver, including linearized Euler equations, Euler equations and compressible Navier-Stokes equations but also Maxwell's equations. The solver implements the Discontinuous Galerkin scheme and

is specifically tailored towards cubical elements with hanging nodes, as they appear in octree meshes. By relying on simple cubical elements, it is possible to represent the solution by the tensor product of onedimensional polynomials, yielding a dimension by dimension approach for the integration. Legendre polynomials are chosen as basis functions, which allows for evaluation of the mass matrix. Moreover, due to the three-term recursion of the basis it is also possible to achieve a fast integration of the stiffness matrix. These properties result in an efficient scheme for arbitrary high orders. High order methods open the path to highly scalable and efficient solutions on modern, distributed systems. They achieve the same accuracy with less degrees of freedom, thereby reducing the required memory, which is a scarce and expensive resource. At the same time, a Discontinuous Galerkin scheme requires only the exchange of surface information between elements, and as this is one dimension less then the volume of the elements, this communication diminishes relatively to the computational effort for sufficiently high scheme orders. The solver writes restart files to disk, which follow the same principles as the *TreElM* mesh format. That is, the polynomial solution of each element is written to disk according to the ordering of the mesh and the identification of the respective element location is given by the position of the element.

Finally, the post-processing tool *Harvester* is used to generate files for visualization tools like Paraview out of the high order results of the simulation. Since *Harvester* is a stand alone tool, it can be deployed on specialized machines, that are more suited for visualization. The post-processing enables the computation of derived quantities and the generation of visualization files for high order polynomial data. This is achieved by a voxelization of the polynomial data up to the required resolution. An advantage here is the possibility to use an adaptive method and stop the voxelization as early as possible, reducing the amount of data in the visualization file.

Further it is also possible to write only subsets of the mesh with the attached simulation data to disk. With the help of tracking objects this restriction of the considered data can be achieved in the solver as well as in *Harvester*. In the solver such writes can be configured for arbitrary intervals that can be chosen independent of the restart data.

We now have run through all the necessary tools for the simulation setup and all parts are capable of dealing with large amounts of data. This is achieved by the common infrastructure built around the octree mesh representation with its simple, yet sufficiently flexible, topology. Only one more block in Figure 7.3 has not been covered so much so far. The *Aotus* library provides an interface to the Lua scripting language and enables

its deployment as a configuration tool in Fortran applications. It is used exensively throughout *APES*, not only as configuration of the individual tools, but also as meta data to describe binary files. This part is described in a little more detail in the following section.

7.6.1 Usability

All features become void, if they can not be used. Therefore all developments ultimately should serve the usability of the system. To make massively parallel computing architectures usable for simulations, we need to deploy scalable algorithms, and to make these algorithms usable we have to embed them in appropriate programs. A main criteria for usability of programs is their user interface. The user interface is the part of the application that by definition becomes most apparent to the user and defines the interaction between the user and the application. To ensure a good usability, the user interface has to be accessible and understandable to the user. It has to provide all features of the software, but should not overwhelm the user by too many details. The target audience of the *APES* tools are users of HPC systems who are familiar with command line interfaces and scripting. Scripts are an expressive method to describe settings and actions, they provide a flexible option to configure and run simulations. Therefore, they are a natural choice for the user interface in the *APES* framework.

Usage of Scripts for the configuration of simulations yields readable input files, as it might be augmented with comments, and mathematical expressions can be directly used to describe given settings. There is a wide range of scripting languages available, that might serve this purpose. However, one of the most important aspects of the scripting implementation in this case is its ease of interoperation with the hosting language. For the wide variety of specialized HPC platforms another necessety is high portability. These driving criteria led to the choice of Lua, which is specifically designed to work well with a hosting language and is provided in a highly portable ANSI-C library. To provide a full featured Fortran interface to Lua, the *Aotus* wrapper library has been written around the Lua C-API. It encapsulates the Lua functionality and exposes it in interfaces, that are compatible with Fortran semantics. Throughout the *APES* tools it is used to provide the bridge between the application developer and the user. However, this description method is so convenient, that it is also used in header files to provide meta data in the file formats consumed and produced by the simulations.

The scriptable input enables the grouping of parameters into logical

blocks, such that each application feature can provide its own self-contained set of parameters to be set. With this organization, a flexible development is possible, where each feature introduces its own settings independent of the other ones. Together with sane defaults this provides a smooth evolution for the application without disrupting its usage by newly introduced configuration parameters. Furthermore, the scripted configuration also allows the definition of functions to be used in the simulation. This is especially useful to describe boundary and initial conditions, without relying on pre-defined functions offered by the application itself. By the definition of space-time functions it is therefore possible to describe arbitrary initial conditions and transient boundary conditions.

Besides the user interface an accurate documentation is essential to ensure usability. The need for up to date documentation has soon be realized in software engineering and various tools to help in this aspect have been developed. A main point is here to keep documentation and actual code tightly entangled, ideally within the same files. Code documentation systems like Doxygen extract interface information from the source code and augment it with explanations provided by the programmer in comments to the code. This minimizes the risk for outdated documentation.

8 Generating the Octree Mesh

While the previous chapter introduced a mesh description that is suitable for scalable parallel processing on distributed systems, it still leaves the question on how such a mesh should be generated. Let us now turn to this subject and especially consider the generation of mesh information for high-order schemes. The high-order Discontinuous Galerkin Finite Element Method (*DG*) has many advantages on modern distributed and parallel supercomputing systems but one drawback is the need for an appropriate representation of the geometrical setup.

One path towards high order boundary representation is the deformation of elements and their superparametric description. A method specifically designed for Discontinuous Galerkin discretizations is offered for example in [33]. This way of describing higher order geometries provides the greatest flexibility in the kind of boundary conditions to apply. But it also introduces additional costs, as the numerical scheme in those elements needs to incorporate the necessary transformations. Furthermore, such deformed elements are subject to varying mesh quality and prone to issues with geometrical constraints.

Instead we will have a look at another option to represent boundaries within elements by some penalization and how this internal representation is obtained in the mesh generation step for the octree. An implementation of this approach is available as open source in the mesh generator *Seeder* [43]. It takes surface triangulations in form of STL [88] files and produces Octree based meshes with cubical elements and the high order polynomial geometry description. This polynomial geometry information is then attached to the elements in the *TreElM* formatted file. The solver than can use it as a description of materials, for example for the permittivity $\epsilon(x, y, z)$ and permeability $\mu(x, y, z)$ in Maxwell's equations (2.8) - (2.11). *Ateles*, the *DG* solver in *APES* uses polynomial basis functions. Therefore, *Seeder* will also generate polynomial representations of the geometry locally in each element.

The fundamental idea to obtain high-order polynomial representations is the use of a simple first-order voxelization within the coarse elements of the mesh for our *DG* solver *Ateles*. This volume information can then be translated into polynomials by a suitable transformation method. A

major feature of the first-order method is its robustness that allows the treatment of complex setups. A drawback is its low accuracy and need for a large number of voxels. However, this is alleviated by using the Octree strategy of recursive refinement.

8.1 Related Work

Our approach is most closely related to embedded boundaries, as used in spectral discretizations. Examples of such approaches are the *Spectral Smoothed Boundary Method* [4] and the *Fourier Spectral Embedded Boundaries* [67]. These methods rely on a representation of the irregular domain by a function. This function, also referred to as *phase-field*, is usually constructed with the help of a global rectangular Cartesian grid. We extend this concept with an Octree mesh, where we apply this method in each of the elements of the mesh. The constructed geometry representation is then defined locally within each mesh cell. This matches the function space of the *DG* and can be directly used by *DG* solvers. Within the elements, we also make use of the Octree bisection algorithm to achieve a fast voxelization of surfaces. In comparison to a global rectangular domain in spectral methods, the composition of multiple elements in a mesh allows for a greater flexibility in the definition of the embedding domain. By fitting the embedding domain to the actual computational domain of interest, the computational effort can be minimized in this approach.

Other methods, where irregular meshes are deployed with an internal geometry representation are typically referred to as *Immersed Boundary Methods*. Introduced by [63] for elastic walls in incompressible flows, this approach has been improved and extended since by various authors. An overview of these methods is for example provided by [52]. Similarly to the strategy we follow here, these methods make use of meshes for the computational domain. However, they rely on surface representations to describe objects within the meshes and directly enforce boundary conditions on these. They are popular for flows with moving geometries but do not provide a direct method to describe varying materials, like the change in permeability and permittivity in electrodynamics. In contrast to the embedded boundaries in spectral methods, the immersed boundaries are usually employed in lower order schemes.

Our goal is the generation of material representations for high-order *DG* solvers. As such, we need a mesh like in the immersed boundary methods, but a high-order functional representation of the geometry as in the embedded boundary methods. This method enables the exploitation of

the fast convergence of spectral approximations but still allows for complex computational domains.

The traditional approach to boundaries in unstructured, irregular meshes is the fitting of the mesh to the geometry. Here, a high-order can be obtained by deforming the elements with curved surfaces. [33] offers a method in this direction specifically designed for discontinuous Galerkin discretizations. However, the identification of such curved boundaries is much more complex and usually not used for internal interfaces like material changes, as both sides of the interface need to be considered. Such deformed, unstructured elements are also subject to varying mesh quality and prone to issues with geometrical constraints. As we will show, the embedding description of materials provides a viable and robust approach to the body fitting of meshes. It comes at the cost of volumetric information that needs to be stored but avoids the need for expensive transformations during the simulation.

8.2 The Seeder Mesh Generator

Seeder [29] is an Octree mesh generator. It produces voxelizations of complex geometries defined by surface triangulations in the form of STL files. Some geometric primitives, like spheres and cylinders, are also available, and we will make use of them in the examples considered here. By voxelization, we refer to the process of subdividing a given volume into smaller cubical elements (voxels) in three-dimensional space. With the Octree approach, these cubical elements are successively split into 8 smaller cubes, where needed. The use of Octree representations in mesh generations is not new [71] but as explained in Chapter 7, we extend the usage of the Octree to our solvers, and need to provide a mesh that preserves the Octree topology instead of discarding it in favor of a general unstructured representation. An example for the voxelization of a sphere is shown in Figure 8.1. The yellow surface indicates the sphere and the red color indicates the voxels completely inside the sphere. Note, how the voxels build a staircase that approximates the smooth surface. Also, the Octree refinement is visible in Figure 8.1. Our concept of voxels does not imply equally sized voxels. Instead, as outlined by the white lines, different voxel sizes arise from the bisection rule of the Octree approach. Thus, we only need to create a large number of small voxels close to the smooth surface while covering the rest of the volume with just a few large ones. The mesh format generated by *Seeder* exploits the topology information from the Octree and is designed for the parallel processing on distributed, parallel

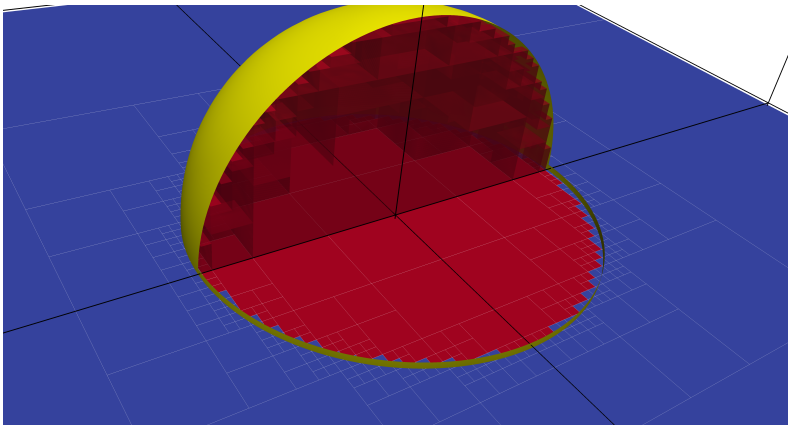


Figure 8.1: Illustration of the voxelization of a sphere within coarse mesh elements. The sphere is indicated by the yellow surface while the thick black lines outline the elements of the actual mesh. The voxelization within elements follows the Octree refinement towards the sphere and is indicated by the thinner white lines. Inside the sphere, voxels have been colorized by the flood-fill mechanism with a seed in the center. Flooded elements are shown in red; other elements are blue.

systems. *Seeder* is freely available online [43] under a permissive BSD license and has been successfully compiled and run on many computing architectures. In the following section, the general voxelization method is briefly outlined. Afterward, we explain the extensions to enable high-order material definitions within the mesh elements.

8.2.1 Basic mesh generation procedure

To produce the voxelization, *Seeder* deploys an approach similar to the *building cube method* described in [36]. The basic idea is an iterative refinement towards geometry surfaces, followed by a flooding of the computational domain starting from a user defined seed. This flooding is limited by elements intersected by boundary objects and all flooded elements finally constitute the actual computational domain. For the refinement, a bisection in each direction is used in each step, resulting in an Octree mesh. Such tree structures are well established and widespread in mesh

generators to identify and sort geometrical objects fast, see for example [91] for an early adoption.

In *Seeder*, each geometry has some refinement level, defined by the user, attached to it. This level describes how many bisection steps should be done to resolve the surface. The higher this level, the smaller the voxels to approximate the surface. Elements are refined iteratively if they intersect a geometry, until the desired resolution is reached. After this step of boundary identification, the actual computational domain is identified by a 3D flood filling algorithm. All elements intersected by a geometry bound this flooding. To avoid unintentional spills, the flooding only considers the six face neighbors (*von Neumann neighborhood*). This mechanism, even though it requires the definition of seeds by the user, is chosen as it provides a high robustness and indifference towards the triangle definitions in STL files. Small inaccuracies in the geometry definition are automatically healed, as long as they are below the resolution of the voxelization. This approach has proven to be robust and applicable to a wide range of complex geometries.

8.3 Generation of Polynomial Geometry Approximations

The cubical elements obtained by the mesh generation procedure described in the previous section, provide the frame wherein we now can construct the high-order surface representation. We want this representation in the function space of the *DG* solver, which often are polynomials and in our solver specifically Legendre polynomials. Legendre polynomials build an orthogonal basis with respect to a weight of one on the interval $[-1, 1]$, and they adhere to the three-term recurrence relation

$$L_i(x) = \frac{2i-1}{i} x L_{i-1}(x) - \frac{i-1}{i} L_{i-2}(x). \tag{8.1}$$

With $L_0(x) = 1$ and $L_1(x) = x$ the higher order polynomials can be recursively computed by (8.1). The first Legendre polynomial $L_0(x) = 1$ is the only one with an integral mean, all higher ones are mean free on the interval $[-1, 1]$.

For material interfaces we usually need to deal with discontinuities as the material property jumps at the interfaces. Thus, we need to project a step function

$$\Xi(x) = \begin{cases} 0 & \text{if } x \leq \xi \\ 1 & \text{if } x > \xi \end{cases} \tag{8.2}$$

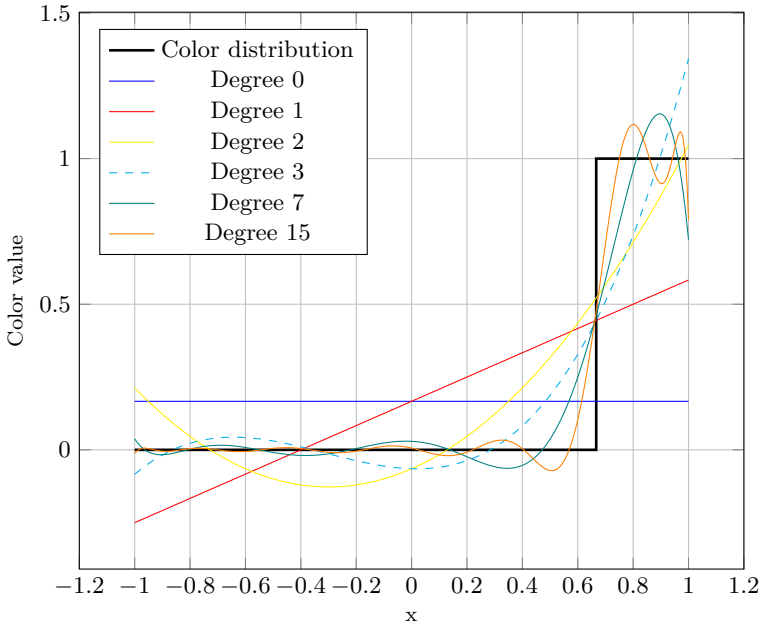


Figure 8.2: Projection of a step function (8.2), jumping at $x = \frac{2}{3}$ onto the space of Legendre polynomials. Shown is the step function along with its approximation by more and more Legendre basis functions obtained by analytical integration.

to our polynomial space and find a suitable expansion

$$P_n(x) = \sum_{i=0}^{n} a_i L_i(x) \tag{8.3}$$

that approximates (8.2).

In Figure 8.2 such a discontinuity with $\xi = \frac{2}{3}$ in (8.2) is shown along with its approximation $P_n(x)$ from (8.3) with various maximal degrees up to $n = 15$. While in this simple 1D example, the projection can be computed analytically, this is not possible anymore for higher dimensions with arbitrary jump definitions. We, therefore, introduce an algorithm in this section to find approximations of the projection numerically.

However, before polynomials can be computed, the material distribution itself needs to be identified. Namely, we need to find regions of the domain where a specific material should be present. We achieve this by selectively attaching attributes to elements in the mesh. To define, which elements should be attributed and which not, we can exploit the flood filling algorithm explained above by enabling multiple fillings instead of just a single one. Individual surface descriptions confine each flooding, which allows the identification of distinct regions in the mesh. By ascribing a particular material definition to each of these regions, the voxelized spatial material distribution can be obtained. This approach is similar to coloring an image, and we refer to those floodings as colors. In the following, we briefly discuss the coloring concept and then move on to the generation of high-order surface representations within the Octree mesh.

8.3.1 Coloring

Seeder takes a surface triangulation along with a seed definition to construct the computational domain with non-overlapping cubical elements. The seeds are usually points, but might also be other geometrical objects and are used as the starting point for the flood-filling algorithm. The surface description builds the confinement for the flood-filling. These two parts together are therefore building the volumetric geometry definition in the mesh. By using multiple of such pairs, it gets possible to describe different regions within the same mesh. We refer to this as coloring, and each seed and surface needs to have a color attached to it.

The flooding spreads from the seeds and is limited by surfaces of the same color. Boundaries of other colors do not affect the flooding and it is possible to have elements flooded by multiple colors. Differently colored regions might, therefore, overlap. The color information is then attached to the elements and provide a method to distinguish specific areas in the mesh. Afterward, the solver can associate individual material properties to given colors.

8.3.2 Sub-resolution

With the coloring principle described above, we are now able to define arbitrary material areas, but we still need to obtain high-order surface approximations inside the elements of the Octree. As this provides information beyond the resolution of the actual mesh, we refer to this as sub-resolution. Figure 8.3 sketches the overall workflow of the algorithm to construct this information.

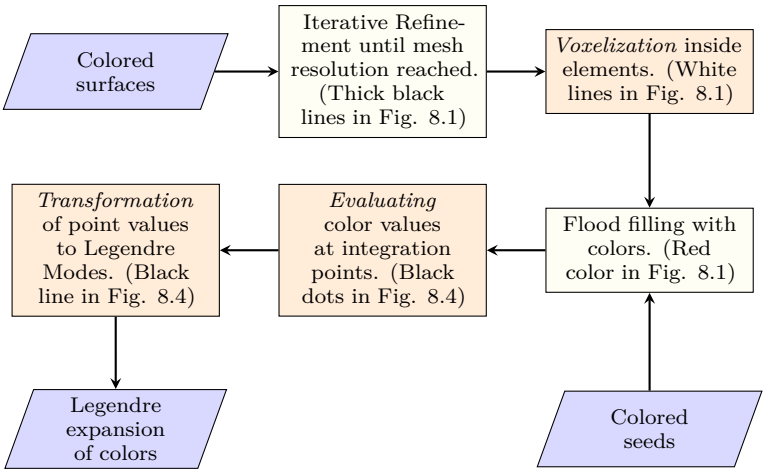


Figure 8.3: Chart of the overall workflow. Required inputs are the surface descriptions and seeding points to start the flooding. The resulting output is the expansion of the color distribution in Legendre modes for each element.

To obtain the sub-resolution as an expansion in Legendre polynomials in each element, we need to perform the following three steps:

- **Voxelization** (and flooding) within elements of the final mesh to identify color distributions
- **Evaluation** of color values at integration points
- **Transformation** of point data to polynomial modes

These three steps are indicated by a darker orange color in Figure 8.3. The other two processes are already described above. A first step creates the coarse mesh with the desired resolution, and after resolving the surfaces within the elements by the voxelization process, all elements and voxels are flood filled with colors.

The first additional step we introduce is the identification of color boundaries inside the coarse mesh elements. Luckily, we already have a robust and fast method to identify color boundaries in volumes by the **voxelization** method described above for the mesh. We can deploy this mechanism also inside elements and recursively refine the voxels towards surfaces. The final mesh will not contain the voxels within the coarse elements but rather the polynomial approximation of the color distribution. We limit the refinement inside elements by setting the number of additional levels ℓ. This number can be freely chosen in the configuration and all elements intersecting a boundary will be refined accordingly, independent of the size of the coarse element.

We observe that even though the voxelization is only a first order approximation, it is feasible to represent the surface accurately due to the exponential nature of the bisection approach in the Octree. After all voxels are known, the mesh generation algorithm proceeds with flooding as described in the previous section. The flood filling does not heed the intersected coarse elements of the final mesh. Instead, all voxels are flooded down to the finest level. With this approach, we do not require a separate algorithm for the flooding of voxels inside intersected elements.

Figure 8.1 illustrates the mesh status after refinement and flooding for eight coarse elements intersected by a sphere. The sphere is indicated by the yellow surface and cut open to reveal the voxelization within. Thick black lines indicate the coarse mesh elements and the fine white lines show the sub-resolution voxels within them. As described above, elements in the interior of the sphere are flooded, which is indicated by the red coloring. The flooding is limited by the sphere and all voxels outside the geometry are not flooded with this color.

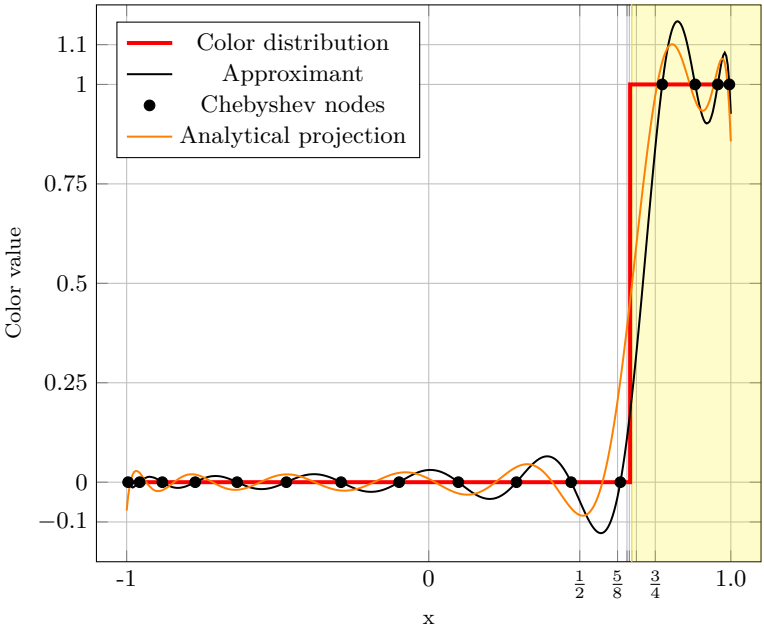


Figure 8.4: Illustration of the approximation method in 1D. For a single element and one discontinuity. The color value jumps from 0 to 1 at $x = \frac{2}{3}$ and is indicated by the red line. The grid lines indicate the bisection sequence and the yellow area highlights the region, where the color value is identified to be 1 by the bisecting approximation. An approximant polynomial of degree 15 is constructed from the 16 shown Chebyshev nodes (black dots). The orange polynomial shows the analytical projection with degree 15, also depicted in Figure 8.2.

Two processes remain to be done at this stage, the *evaluation* of color values and their *transformation* to Legendre modes. These two are hard to illustrate in three dimensions, and we will instead make use of the one-dimensional setup in Figure 8.4. It makes use of the same target step function Ξ with $\xi = \frac{2}{3}$, as the one in Figure 8.2, but outlines the individual numerical steps we take to arrive at an approximation of the analytical projection. Vertical grid lines indicate the recursive voxelization towards the surface point. We assume the flooding to happen right of the surface, that is, the seed is at some location $x > 1$. This flooding is indicated by the yellow shaded area in Figure 8.4. Thus, the numerical method has to approximate a step function (8.2) with $\xi = \frac{2}{3}$, illustrated in the figure by the thick red line. With eight bisections in the voxelization, this results in an approximation of the jump location at $\hat{\xi} = 0.671875$. This state corresponds to the three-dimensional case outlined in Figure 8.1.

The number of additional refinement levels ℓ determines the spatial accuracy of the surface approximation. However, they only build one half of the numerical approximation; the other half is governed by the quality of the polynomial representation that we construct in the final two steps. The accuracy of our polynomial approximation is determined by the number of Chebyshev nodes N at which the color distribution is evaluated. For a given N, Chebyshev nodes are given by

$$x_c = cos\left(\frac{2c-1}{2N}\pi\right), c = 1, ..., N. \tag{8.4}$$

Both factors, ℓ and N, can be set independently by the user. However, they both limit the accuracy of the overall approximation, and for optimal results, they need to be correlated. The minimal distance between the first Chebyshev node and the element boundary is proportional to N^{-2}, and the length of the smallest voxel within the element is proportional to $2^{-\ell}$. To resolve all node distances, it is, therefore, necessary to choose the number of additional levels ℓ according to

$$\ell \geq \lceil 2\log_2(N) \rceil. \tag{8.5}$$

Once the flooding situation of all voxels is known, we can **evaluate** the color at each Chebyshev node. For this, numerical values need to be associated with the flooding status for each color. Usually, we assume a value of 1 for flooded voxels and a value of 0 for non-flooded voxels. Due to the spatial discretization by an Octree, the color value at each Chebyshev node x_c can be found fast with logarithmic computational complexity. In Figure 8.4 the Chebyshev nodes (8.4) for $N = 16$ are indicated by black

dots and their color value by the elevation of the dots. The approximated step function provides the color values, and we obtain

$$f_c = \Xi(x_c) \text{ with } \hat{\xi}, c = 1, ..., N \tag{8.6}$$

for the polynomial values at the Chebyshev nodes x_c. It is notable that the position of the surface is only significant up to the interval between two neighboring nodes. A variation of the jump location within the interval between two neighboring nodes does not change the final approximation. The only option to increase the accuracy of the approximation is to use a larger number of integration points N.

Finally, the **transformation** of the nodal information (8.6) to a suitable function space for the solver has to be done. A typical choice for discontinuous Galerkin methods is the orthogonal Legendre basis. To obtain the Legendre modes a_i in (8.3) from the values at the Chebyshev nodes f_c in (8.6), we apply the fast polynomial transformation proposed in [1]. However, other target functions with different point sets could also be plugged into the described machinery. The obtained polynomial recovers the function values f_c at the nodes exactly and is drawn in Figure 8.4 by the black line running through all dots. Due to the discontinuity of the step function, the representation in polynomial space is an infinite series. The finite approximation suffers from the Gibbs phenomenon [89]. However, besides this fundamental problem, also inaccuracies due to the numerical integration can be seen as the numerical (black) and the analytical (orange) projections do not coincide in Figure 8.4. These result in a distorted location of the jump. In the next section, we will have a closer look at these numerical issues.

With this step, we now have a volumetric description that yields a high-order approximation of the surface. Note that only intersected coarse mesh elements need to get this information added, all other elements have constant colors. Thus, the need for volume information is limited to a small area at the surface.

8.4 Numerical Properties

In this section, we investigate the numerical properties of the described approximation method. Though, the one-dimensional problem with a single jump is much simpler than a real three-dimensional geometry; it is still instructive for the fundamental properties of the algorithm. Let us recall that the goal is an appropriate representation of the geometry in a high-order discontinuous Galerkin solver. Typically, the deployed functions

in the solver are smooth within elements. Here we consider specifically Legendre polynomials, which are attractive due to their orthogonality. The representation of a non-smooth material distribution in the finite smooth function space, therefore, can only be approximate.

Degree	L^2-Error
0	0.527046
1	0.402538
3	0.226028
7	0.167786

Degree	L^2-Error
15	0.122322
31	0.086937
63	0.060674
127	0.043065

Degree	L^2-Error
255	0.030481
511	0.021513
1023	0.015218
2047	0.010763

Table 8.1: The convergence of the series of Legendre polynomials towards the step function with the jump at $x = \frac{2}{3}$.

Table 8.1 shows the convergence behavior for the series of Legendre polynomials, obtained by L^2 projections of the step function at $x = \frac{2}{3}$ in the interval $[-1, 1]$. A slow convergence can be observed, which is well known for high-order approximations of discontinuous functions. However, the error outside a small band around the discontinuity can be improved later on by a post-processing step, as shown in [25]. The error is bound to this band around the discontinuity, and the width of that band decreases with the number of degrees of freedom. While we can compute this analytic projection for the simple setup with a single discontinuity in one dimension, this is not possible anymore for multiple dimensions and more complex geometries. Thus, we need the previously described numerical approach to approximate the projection. In the following, we first analyze how well the numerical scheme recovers the optimal solution given by the L^2 projection for the simple one-dimensional discontinuity.

Figure 8.5 shows the convergence of the numerical procedure towards the analytical projection onto a polynomial space with a maximal degree of 15. Plotted is the error over the spatial resolution, where the spatial resolution is given by the number of Chebyshev nodes used for the numerical integration. The difference in terms of the L^2 norm to the analytical projection is represented by the blue line and covers all modes of the polynomial. With the red line, the absolute difference in the volume is provided. Note that the volume equals to the first mode of the Legendre polynomial and thus, is easily obtained. Nevertheless, the two curves show a similar behavior, and the volume error can be used as a good indicator of the qualitative behavior of this numerical approximation. Apparently, the error does not converge uniformly, but on average we observe a convergence rate of roughly 1. This error is comparable to the

one due to the voxelization, and so these two resolutions (number of voxels and number of integration points) should be in the same range.

To investigate the mechanism in 3D, we look at three different geometrical objects: a sphere, a cube, and a tetrahedron. In each case the overall domain is a cubical box $[-1,1] \times [-1,1] \times [-1,1]$ subdivided by 8 cubical elements. The sphere is put in the middle of the domain, with its center at $(0,0,0)$ and has a radius of $\frac{1}{3}$. Similarly, the cube has an edge length of $\frac{2}{3}$, and its barycenter is placed at the center of the domain at $(0,0,0)$. As a third basic geometry, the tetrahedron again is similarly defined with its barycenter in the center of the domain and an edge length of 1.

The error in the volume approximation is used to assess the quality of the polynomial representation. Figure 8.6 plots this measure for the sphere over the two available parameters voxelization resolution and numerical integration points. It can be observed, that the error is mostly bounded by either one of the parameters and for a minimal computational effort it indeed is necessary to change them according to the relation (8.5).

Figure 8.7 illustrates the projection of the sphere on a 3D polynomial representation in the eight elements of the mesh. From left to right it shows improved accuracies. In yellow the isosurface of a color value of 0.5 is shown and in comparison the half of the reference sphere is shown in blue. The leftmost image shows the sphere embedded in the eight elements, indicated by the black wireframe. In this, a very rough estimation of the sphere is shown with a polynomial of degree 15 and a low voxelization resolution. Clearly, the staircases from the voxelization are visible in the polynomial representation here. The next image shows a zoom in for finer voxelization, but still a polynomial degree of 15, in the left half the reference sphere is again depicted in blue. While the finer resolution in the voxelization yields a better approximation now, there are relatively strong oscillations visible, especially close to the element boundaries. To allow for a better representation of the sphere we move to a polynomial degree of 31 in the third image. The voxelization is chosen with an appropriate resolution, in this case, but the numerical integration results in aliasing issues, exhibiting a staircase-like effect for the isosurface of the polynomial. Finally, in the rightmost image, we see the impact of a higher number of points for the numerical integration, resulting in a much smoother geometry for the shown polynomial of degree 31.

Figure 8.8 illustrates the approximation of a cube. All images show the isosurface for polynomial representation of degree 15, but the accuracy of the numerical approximation increases from left to right. The number of integration points increases from 16 in the leftmost image over 32 and 48 to 64 in the rightmost image and the voxelization is chosen according to

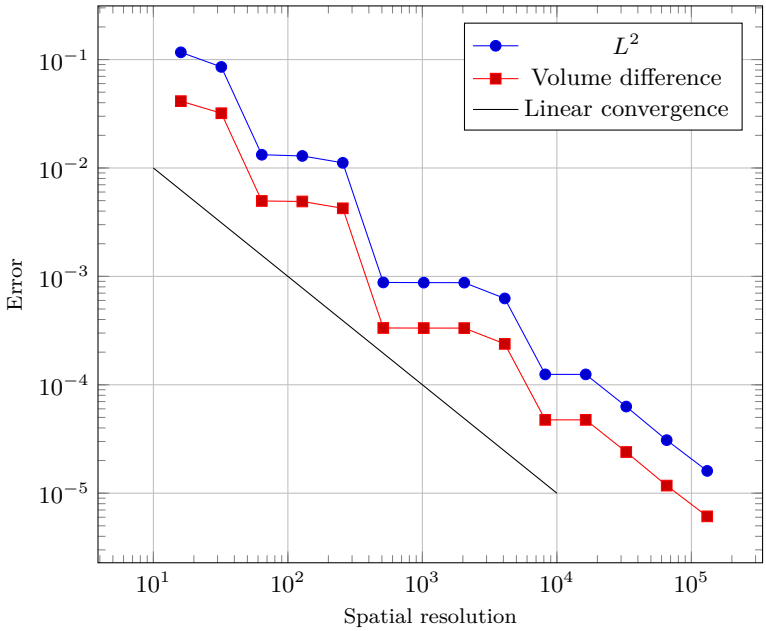


Figure 8.5: Error convergence of the numerical approximation towards the analytical projection for a polynomial of degree 15. The blue line shows the L^2-error over all 16 modes while the red line shows the absolute error in the first mode, which represents the volume. On average, a convergence rate of 0.973 is achieved. Keep in mind that in comparison to the actual step function, the error from Table 8.1 always remains.

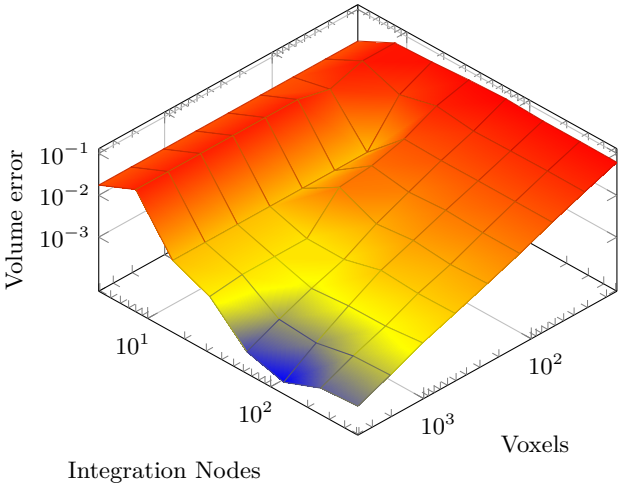


Figure 8.6: Error in the volume approximation for a sphere with a radius of $\frac{1}{3}$. The mesh consists of 8 elements with a common vertex in the center of the sphere.

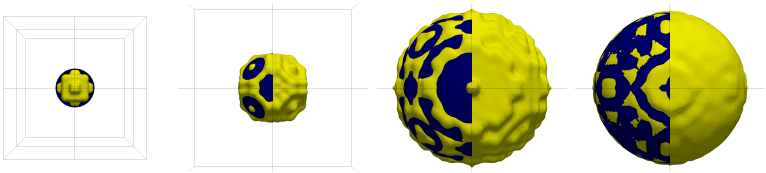


Figure 8.7: Illustration of sphere approximations, with increasing accuracy. The sphere is blue, and the isosurface of the color value at 0.5 is yellow. On the left, the sphere is shown in the embedding domain with the 8 elements. Voxelization and integration points increase from left to right.

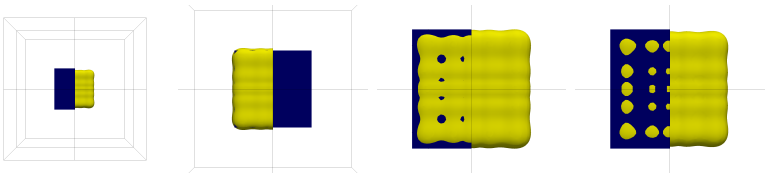


Figure 8.8: Representation of the cube in 8 elements with polynomials of degree 15. From left to right an increasing number of integration points is used. The leftmost image shows the cube with the 8 elements of the mesh. The reference geometry is drawn in blue, and the isosurface of the color value 0.5 in yellow. We cut the reference in the middle to enable a better view for the comparison, except for the second image, where it is the other way around, and the isosurface is cut.

Equation 8.5. Edges and corners get smoothed out, but we observe only little oscillations for this simple, axis aligned, geometry.

Finally, Figure 8.9 depicts a study on the 3D polynomial approximation of a tetrahedron. The maximal polynomial degree increases from 7 up to 63, and twice as many integration points as polynomial modes are used in each approximation. We observe that the sharp edges and corners are smoothed out at low orders but are increasingly well recovered in the higher resolved polynomials. Also, oscillations in the planes of the tetrahedron get smaller in amplitude. With a polynomial of degree 63, the original shape is well captured. This shows that a high-order polynomial can nicely be constructed, even for objects with sharp corners and edges.

Figure 8.10 shows the application of the described method to a more complex geometry. Depicted is in yellow the isosurface of a polynomial approximating a porous medium, described by a triangulation in an STL file shown in dark blue. This geometry features small bridges and holes. Those are well recovered by the polynomial approximation; only the edges get a little bit smoothed out. Keep in mind, that we are only using cubical elements, which can be exploited by the numerical scheme. Also, no bad elements arise resulting in an extremely robust mesh generation.

8.5 Mesh Generation Summary and Future Work

Seeder implements a robust first order method to obtain polynomial representations of geometrical objects to describe nonsmooth material

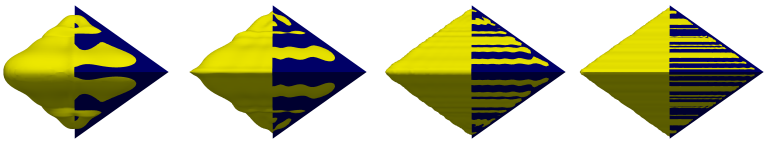


Figure 8.9: Approximation of the tetrahedron with an increasing polynomial degree from left to right. Starting on the left with a polynomial degree of 7 and increasing over 15 and 31 to 63 in the rightmost image. Shown is the isosurface of the polynomial at a value of 0.5 in yellow and for comparison the reference geometry cut in half with a blue coloring.

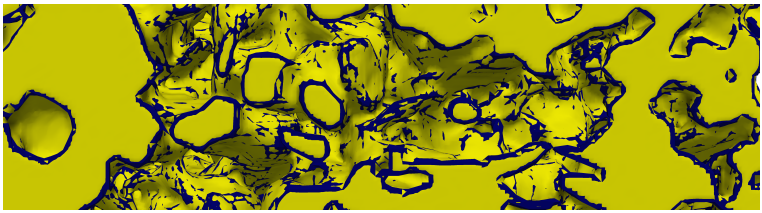


Figure 8.10: Isosurface of a porous medium (yellow) in comparison to the original STL data (blue). The geometry is well recovered; only edges are smoothed out a little.

distributions. The robustness is due to the low order geometry approximation and the usage of a flood filling algorithm, eliminating the need for high requirements on the quality of the geometry representation. By employing the bisection strategy of the octree, the generation is also fast and allows the detailed discretization of complex geometries. It has been shown, that the numerical approximation indeed converges towards the optimal L^2 projection of the nonsmooth distribution onto the polynomial function space, used in high order discontinuous Galerkin solvers. A possible improvement over the current staircase representation of the surface could for example be achieved by computing an approximate plane for the geometry within intersected voxels. While such a computation would introduce additional complexity and potentially expensive computations, it would reduce the number of voxels required to resolve the shortest distances between integration nodes. Nevertheless, the pure voxelization scheme currently deployed is already capable to discretize large an complex settings and has been successfully used for highly detailed simulations. Though the errors from the Table 8.1 remain, it is proven in [92] that high order information can be recovered for such nonsmooth setups by an appropriate post-processing.

9 Results with Ateles

Ateles implements the *DG* scheme on the Octree provided by *TreElM*. It specifically exploits the fact, that all elements are simple cubes and uses a modal or nodal representation as needed. This results in the need for polynomial transformations but allows for fast integrations with the appropriate polynomial basis. Methods that combine high-order approximations with the concept of finite elements are also commonly referred to as spectral element methods [62]. They provide the flexibility of mesh discretizations and combine them with the fast error convergence of high-order approximations.

Ateles uses Legendre polynomials as basis functions. It relies on cubical elements and exploits them with a dimension-by-dimension approach. This basis yields a fast application of the mass matrix in the *DG* scheme. Because of its three-term recurrence relation it also yields a fast application of the stiffness matrix. In a purely modal setting that can be used for linear equations with constant material, we obtain a numerical complexity that scales only linearly with the number of degrees in freedom. Thus, increasing the resolution by increasing the polynomial degree or refining the mesh has the same computational complexity attached to it. Only, the time step size restriction for the explicit time integration is more severe for the high-order approximation. For nonlinear equations or varying materials, a nodal approximation is required, and *Ateles* has to perform the conversion

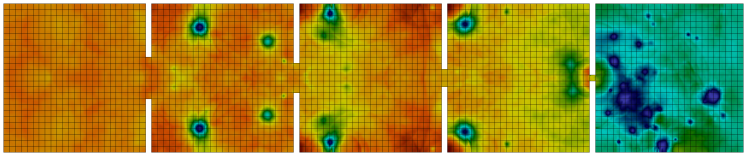


Figure 9.1: Coarse mesh configuration for a eleventh order simulation of an aerodynamic lens. The colors show an instanteneous pressure field on a scale from $\frac{1}{2}$ bar to 2 bar.

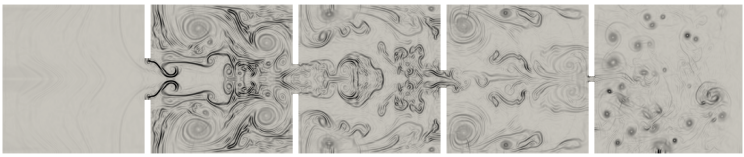


Figure 9.2: Schlieren-like visualization of the flow in the aerodynamic lens. This plot of the magnitude of the density gradient illustrates how well small scale structures and contact discontinuities are preserved by the high-order discretization (maximal polynomial degree of 10 was used in this simulation).

between modal and nodal representations. This is an expensive operation, but fast transformations exist that bound the computational complexity by $\mathcal{O}(m \log(m))$. With these *Ateles* is well equipped to solve a broad range of setups with high-order approximations.

Figures 9.1 and 9.2 illustrate a flow simulation with *Ateles* for an aerodynamic lens [72] setup. The simulation was done with an eleventh order discretization and the mesh resolution illustrated in Figure 9.1. By plotting the magnitude of the density gradient, a Schlieren-like visualization of the flow is obtained in Figure 9.2. It can be nicely seen, how small scale structures are still well resolved in this relatively coarse resolution. An remarkable feature is the conservation of contact discontinuities in the numerical solution.

9.1 High-Order Efficiency

As outlined earlier, high-order approximations yield a small error with few degrees of freedom. Fewer degrees of freedom are equal to smaller amounts of required memory. *Ateles* is built to deal exceptionally well with high-order approximations, because of this insight. The modal approach used in *Ateles* is especially well suited for linear equations like Maxwell's equations or the linearized Euler equations, as the modal space never has to be left. However, the high-order polynomial representation in the *DG* scheme leads to a tight stability restriction on the time step size in explicit time integration schemes. To ensure stability, the time step size has to be

proportional to the inverse of the maximal polynomial degree squared:

$$\Delta t \ \frac{h}{m^2}. \tag{9.1}$$

Thus, a high-order discretization results in a larger number of time steps and, therefore, also in an increased computational effort. We have to ask whether, this increased cost actually pays off, or if a lower order scheme with a smaller element size would be the better option.

Let us have a look at Figure 9.3. This graph shows the error convergences for different discretization schemes with the error on the y-axis and the computational effort on the x-axis. We are simulating a simple sinusodial wave in a periodic domain, that gets transported as an acoustic wave from left to right. A full period of the wave is simulated, and the numerical solution is compared against the analytical solution at that point in time. The same physical time is simulated by all schemes and high-order simulations suffer from the small time step.

Now the error is shown in Figure 9.3, for the orders $2, 4, 8$ and 12. For each order a grid-refinement study is done, where the computational effort obviously grows with the number of elements. This is visible in the monotoneously increasing computing time in each of the series. Each series starts off with a single element and the doubles the number of elements from data point to data point. Clearly the error convergence order can be observed in the gradient of the curves in this double logarithmic plot. And even with the increased number of time steps, the highest order obtains the most accurate result with the smallest amount of computational effort.

9.2 Scalability of *Ateles*

To assess the parallel performance of our solvers, we use a performance measure that allows us to intuitively recognize the quality of the execution. For *Ateles* we use million degree of freedom updates per second, as this is what we are interested in. Thus, the faster the execution the more updates per second we will obtain and the larger our performance measure will be. In short we refer to this number as MDUPs. This section shows some scaling results on the HLRS system *Hermit* in Stuttgart.

The performance not only depends on the machine but usually also on the problem size. There might be a large variation in performance over the problem size due to effects like caching, overheads in calls or communication to computation ratios. To captures this behavior, we employ plot that shows the MDUPs per parallel execution unit over the problemsize for various counts of parallel execution units. Because this

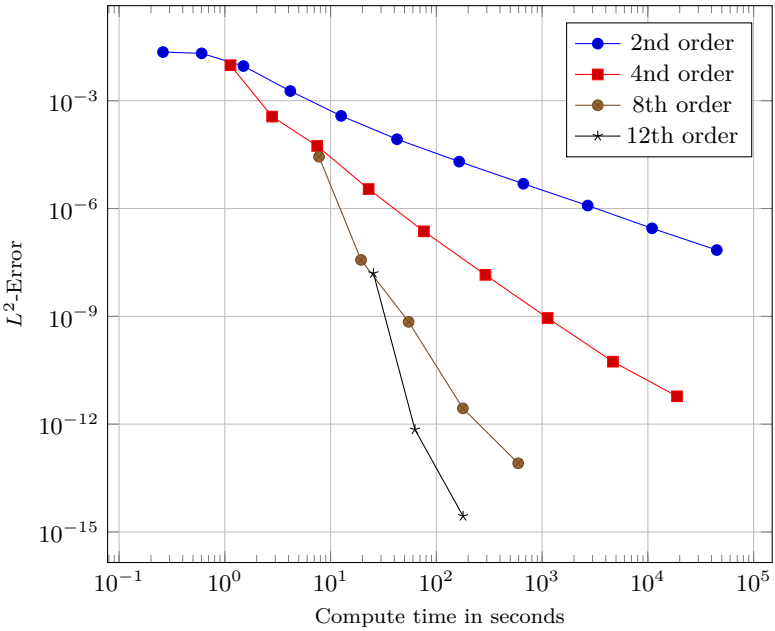


Figure 9.3: Convergence plot in terms of computing time for a single acoustic sine wave in a periodic domain.

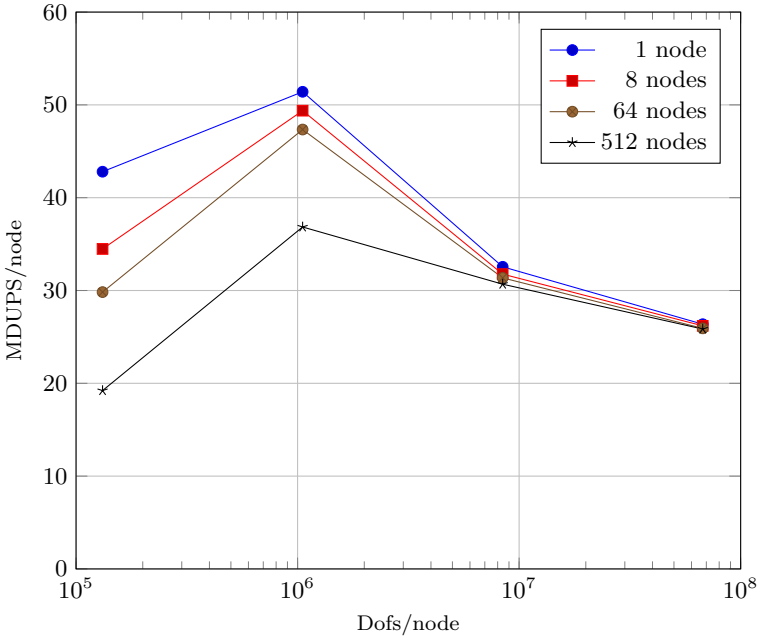


Figure 9.4: Performance map for *Ateles* solving Maxwell's equations with 7th order spatial discretization on up to 512 nodes of *Hermit*.

kind of plot provides a global overview on the parallel execution behavior, we refer to this as a *performance map*. Figure 9.4 shows such a performance map for *Ateles* and a seventh order spatial discretization. The single node performance provides something like a reference line for the achievable performance. More nodes result in network communication and, therefore, in reduced performance per node. We observe a cache effect in Figure 9.4, which results in a peak performance at around one million degrees of freedom per node. This problem size fits completely into the cache of the processors in the node. For too small problems left of the peak we observe a smaller performance, due to various overheads that weigh in at these tiny problems. For larger problems, right of the peak, we observe also a decrease performance, which is due to the required memory accesses in this region. The performance map also highlights, that there is a large difference in performance in the cache region for different node counts but for larger problems per core, they converge and provide nearly the same performance per node as the single node execution. By neatly summarizing parallel executions with problem size dependent behavior, the performance map also explains weak and strong scaling figures. Weak scaling is simply obtained by the vertical distance between the data series for different node counts. In the example of Figure 9.4, the performance map reveals that the weak scaling will be bad for problems per node ins the cache region. However, it will be pretty fine in the memory region for sufficiently large problem sizes per node. This weak scaling for different problem sizes per node is shown in Figure 9.5. Observe the bad scaling in the region of small problems per process but the nearly ideal scalability for sufficiently large problem sizes.

The weak scaling behavior gets better with increasing scheme order because the relation between required computation within the elements and the communication with other elements becomes smaller. Figure 9.6 shows the corresponding parallel efficiency for a weak scaling of *Ateles* with a 31st order discretization. Note that there are only two different problem sizes per node are present in that plot. This is due to the minimal problem size imposed by a single element. While the 7th order incurs 343 degrees of freedom per element, the 31st order already requires $29,791$. With this imposed minimal problem size we can not observe the badly scaling small problem sizes anymore.

The strong scaling is a little harder to derive from the performance map, as it not only involves the jump from data set to data set as in the weak scaling but also movement along the x axis to adjust for the changed problem size per node. Thus, to obtain the strong scaling with a fixed overall problem size, we start with a large problem and a small node count

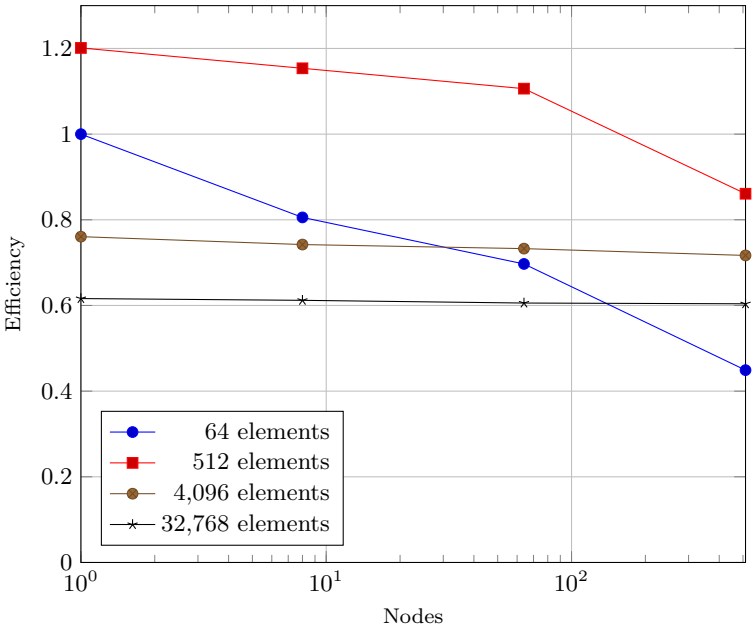


Figure 9.5: Weak scaling for 7th order *Ateles* on up to 512 nodes of *Hermit*. This plot provides a different view on the data from Figure 9.4. It shows the parallel efficiency for different number of elements per node over increasing node counts.

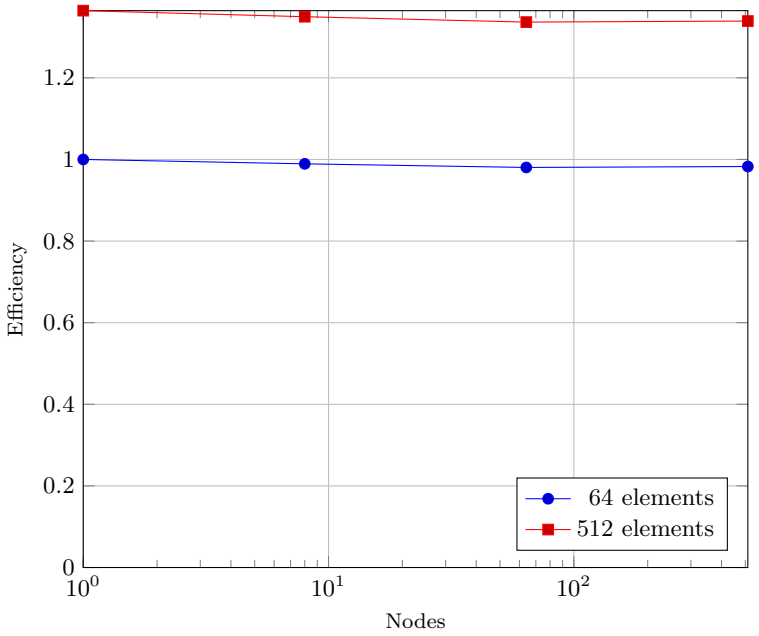


Figure 9.6: Weak scaling for 31st order *Ateles* on up to 512 nodes of *Hermit*.

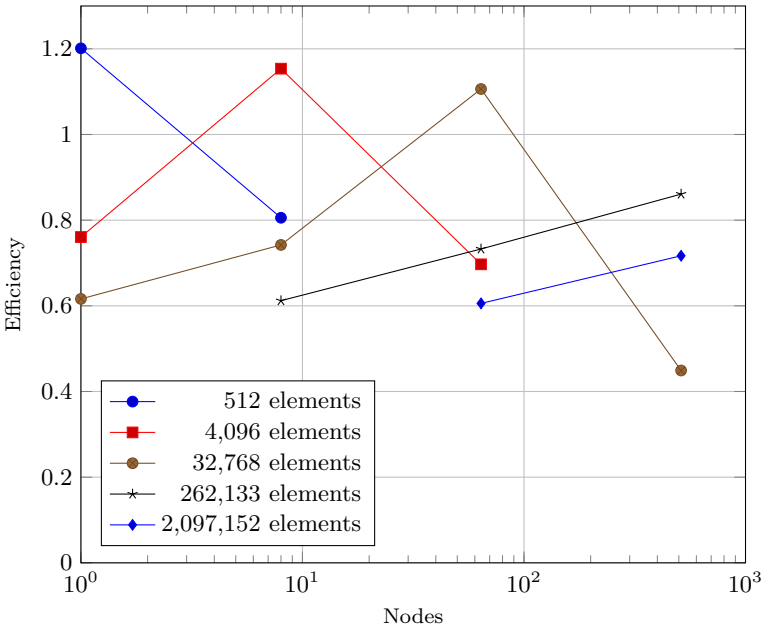


Figure 9.7: Strong scaling for 7th order *Ateles* on up to 512 nodes of *Hermit*. This plot provides a different view on the data from Figure 9.4. It shows the parallel efficiency for different numbers of total element counts.

on the right. Then we move to larger and larger node counts but at the same time move along the x axis to the left. Figure 9.7 shows the strong scaling behavior for the seventh order discretization from the performance map for various fixed total problem sizes.

9.3 Seeder Generated Material for Electrodynamics

To show the behavior of the obtained geometry, we look at an electrodynamic setting, governed by equations 2.8 - 2.11, and solve it with *Ateles*. For time integration, a classical explicit fourth order Runge-Kutta integration is deployed. The simulation setup is an infinite cylinder, impinged by a planar wave. In this setting, the scattering of the wave can be described

by a Mie series [51], which we use as the reference solution. Our simulation setup has the following dimensionless parameters:

- Permittivity and permeability in the surrounding: $\epsilon_S = \mu_S = 1$
- Permittivity and permeability in the cylinder: $\epsilon_C = \mu_C = 2$
- Simulated domain: $[-1, 1] \times [-1, 1] \times [0, 0.125]$
- Radius of the cylinder $r = 0.21$
- Center of the cylinder: $(0.0009765625, 0.0009765625)$
- The impinging planar wave has a wave length of $\lambda = 0.25$

The domain is discretized with $16 \times 16 \times 1 = 256$ elements, and polynomials with a maximal polynomial degree of 15 are used to represent the solution in each element. For the initial condition of the numerical simulation, we also employ the Mie series. We compare the numerical result with the reference solution after the impinging wave has traveled once by the diameter of the cylinder ($\Delta t = 0.42$).

In Figure 9.8 the instanteneous exact solution for the Z-component of the displacement field $\mathbf{D} = \epsilon\mathbf{E}$ is shown on the left. Right to this reference solution, the difference between the numerical solution and this reference after a simulation time of 0.42 is depicted. We use a range in ± 10 % of the maximal amplitude in the exact solution for the scale of the difference. Figure 9.9 shows the numerical result itself. A de-aliasing is applied in the numerical scheme here. Thus, while the scheme uses 16 modes per direction to represent the solution, we use twice as many modes (32) to compute the multiplication of the material distribution with the solution. In this numerical simulation, no voxelization is employed. Instead, the exact definition of the cylinder is used to determine the material values at the 32 Chebyshev nodes per direction that are used to construct the polynomials with a maximal polynomial degree of 31. This corresponds to $\ell = \infty$ with $aa = 1$ in Table 9.1. As can be seen, the reference solution is recovered quite accurately in the largest part of the domain. Only close to the actual interface, there are larger deviations observed. Please note that no smoothening post-processing was applied here, and all Gibbs oscillations are visible in the deviations.

We now replace the exact definition of the cylindrical geometry by the polynomial representation obtained by the method described in Chapter 8. All other parameters of the numerical setup remain the same. For all simulations, the cylinder geometry is approximated by polynomials with a

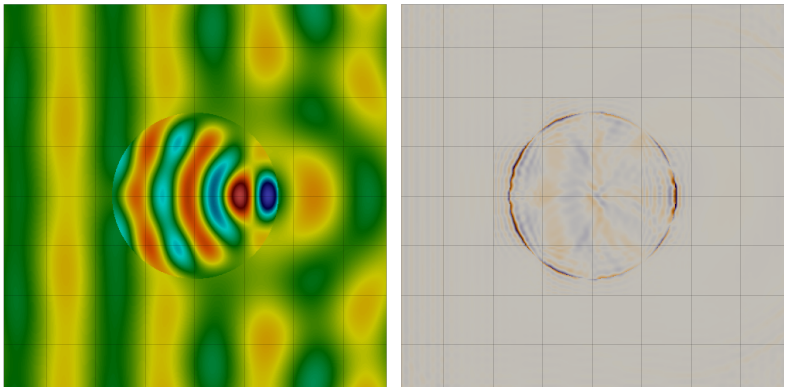


Figure 9.8: Scattering of a planar wave at a cylindrical object. The grid lines indicate the mesh of the *DG* solver. For the numerical solution, a basis with a maximal polynomial degree of 15 is used. On the left, the reference solution is shown. On the right, the difference between the numerical solution and the reference can be seen for a de-aliasing by 32 points. The color scale for the difference is chosen with a range of ±10 % of the maximal amplitude in the reference.

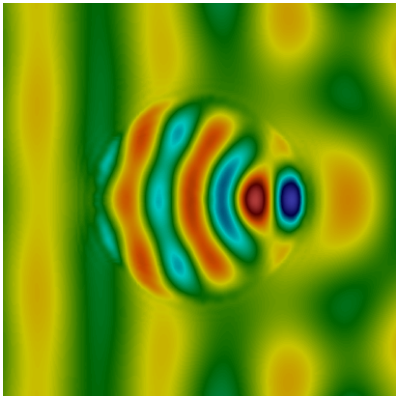


Figure 9.9: Numerical result of the Mie scattering experiment. While Figure 9.8 shows the reference solution along with the error of the numerical approximation, this image now shows the actual numerical solution. It shows how the numerical solution gets a little distorted at the material discontinuity but otherwise the solution is well approximated.

maximal polynomial degree of 31 in each element. Similarly to Figure 8.6, we can vary the number of integration points and the size of the smallest voxels in each element for the construction of these polynomials.

To judge the quality of the thereby obtained cylinder approximations for the DG scheme, we build the L^2-error across the 8×8 elements enclosing the cylinder. For the comparison, we consider the instantaneous solution after simulating a time interval of 0.42 (the time it takes the impinging wave to move once by the diameter of the cylinder). The errors are measured in the Z component of the displacement field $\mathbf{D}$ and are shown in Table 9.1. We increase the voxel resolution (ℓ) from row to row in Table 9.1. In the columns we increase the anti-aliasing, that is the number of Chebyshev nodes to construct the polynomials of degree 31. The aa factor is to be understood as a multiplicator, such that with $aa = 1$ we use 32 points per direction to construct the polynomials and with $aa = 4$ we use 128.

As can be seen in Table 9.1, the error always improves with the voxel resolution ℓ. A higher anti-aliasing, however, does not always improve the solution quality. This behavior matches the observations in Figure 8.6 and emphasizes the necessity for voxel resolutions that resolve the smallest

ℓ	$aa = 1$	$aa = 2$	$aa = 4$
5	3.452e-03	3.345e-03	2.973e-03
6	1.949e-03	2.066e-03	1.944e-03
8	1.491e-03	1.329e-03	1.318e-03
10	1.401e-03	1.235e-03	1.241e-03
∞	1.339e-03		

Table 9.1: L^2-error in the Mie scattering simulation after a simulation time of 0.42 for different geometry approximations. Simulations were done with a spatial order of 16. The polynomial representation of the geometry uses a maximal polynomial degree of 31. For the time integration, a classical explicit fourth order Runge-Kutta scheme is used.

distances between Chebyshev nodes.

10 Future Work and Summary

In this final chapter, some outlines for further work are provided, and a short summary of the presented work is provided. This work set out to enable heterogeneous, coupled simulations for multi-scale problems on large, distributed and parallel computing systems. Over the course of this challenging task it become obvious that some fundamental changes had to be implemented to overcome scalability issues, not only in the solver, but also in the supporting tools like mesh generation and post-processing. The complete scope of this huge task could not be covered in this single thesis, so a lot of work remains open and further paths will be followed from here on.

10.1 Summary

This work is concerned with the simulation of multi-scale problems like aero-acoustics on large parallel and distributed computing systems. Major bottlenecks, inhibiting scalability, were overcome and a completely new approach has been implemented. The focus in the numerical methods is put on high-order schemes, as they can provide the required accuracy for detailed simulations with less degrees of freedoms. We covered the challenge to bring heterogeneous, coupled simulations to massively parallel systems with distributed memory. By utilizing *PACX-MPI*, we even were able to run this kind of setup on heterogeneous computing infrastructure. It was shown that matching the heterogeneity of the software to the heterogeneity of the hardware is advantageous and yield lower times to solution than either system alone. An ultimately non-scaling and therefore limiting factor was identified in the mesh generation and adjacency search. This bottleneck can only be overcome by more dramatic changes, and sacrifices in the generality of the mesh representation. A new framework revolving around this idea has been established and opened the path to truely scalable simulations on distributed systems. The work on this new framework is ongoing and not only involves the solvers themself but also pre- and post-processing tools.

Especially the mesh generation with high-order geometry representation in this new framework has been elaborated. The assembled tools and

methods constitute a new environment to address multi-scale problems on massively parallel systems of the future. The advantages of high-order methods for modern computing systems were discussed and a brief discussion on the efficient implementation has been conducted. The new framework offers already great usability and enables the utilization of large scale computing systems for a broad range of simulation setups. Developments that still need to be done are those related to dynamic adaptations of the discretization. Such dynamic adaptations will advance the new implementation beyond the scope of the original coupling idea and enable the addressing of new questions.

10.2 Future Work

With the status of this thesis, the *APES* framework is not as adaptable as it should be. It is still lacking dynamic mesh adaptation, though the Octree representation would be highly suitable for this feature. Work is ongoing to enable the dynamic mesh adaptation in *TreElM* and the solvers and will allow a further limitation of the numerical effort to those regions, where it is actually needed. However, *Ateles* provides an implementation of a high-order method, which allows the consideration of individual elements as spectral domains in themselves. Such high-order schemes in combination with mesh discretizations are also referred to as spectral element methods [62]. By viewing each element as a single spectral domain, we can think of the overall *DG* mesh as a coupling of individual spectral domains. A vision for further developments is to allow here adaptations in the equations to solve, the spatial resolution in terms of element size and polynomial degrees and the temporal resolution in terms of time steps. This would combine the *KOP* concept with h/p-Refinement and utilize the Octree infrastructure for high scalability. Instead of a static setup that needs to be defined a-priori, the dynamic setup could develop an appropriate discretization in the course of the simulation itself and thereby not only reduce the computational effort for a given problem, but also increase the usability. In this concept, the individual elements are less parts of a mesh but more independent agents that solve the partial differential equations in the space, they are covering. The high dynamics of such a setup and the detailed interaction between such element agents pose a challenging problem. However, the Octree infrastructure and high-order *DG* discretization provide a solid basis to tackle it.

The concept of independent agents interacting with each other, matches well with modern computing systems, where resources are more and more

distributed and most operations need to be local. Thus, even so the dynamics of the system might be hard to handle and decrease the sustained performance, such a concept may well be feasible on modern high performance computing systems and can allow us to solve larger problems in a more automated way. That is, we could shift the work of finding a proper domain decomposition from the user to the computer. This only becomes possible by employing spectral discretizations in the *DG* elements, as only in this case the effort in each element is sufficiently large and the communication between elements is negligble small.

An important building block on the way to such a highly adaptive simulation, is a more advanced time integration scheme, like for example the local space-time discontinuous Galerkin method by Dumbser et al. [13]. These time integrators also have the additional benefit that they might yield a parallelization in time. They would therefore allow the utilization of large parallel systems for long running transient simulations. By employing a global space-time discontinuous Galerkin method, as proposed by van der Vegt and van der Ven [85], it might even become possible to parallelize multiple small time steps that might be adjacent to an element with a larger time step. This strategy would result in a time slice that needs to be solved across all elements but with the possibility to adapt the discretization locally in space and time. Therefore, the implementation in *APES* now provides both, a more integrated and a more flexible framework to achieve multi-scale simulations like aero-acoustic problems.

Bibliography

[1] Bradley K. Alpert and Vladimir Rokhlin. “A Fast Algorithm for the Evaluation of Legendre Expansions”. en. In: *SIAM Journal on Scientific and Statistical Computing* 12.1 (Jan. 1991), pp. 158–179. ISSN: 0196-5204, 2168-3417.

[2] Michael Bader. *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. 2012th ed. Springer Berlin Heidelberg, Sept. 2012. ISBN: 3642310451.

[3] David Blackstock. *Fundamentals of Physical Acoustics*. English. 1 edition. New York: Wiley-Interscience, Feb. 2000. ISBN: 978-0-471-31979-5.

[4] Alfonso Bueno-Orovio, Víctor M. Pérez-García, and Flavio H. Fenton. “Spectral Methods for Partial Differential Equations in Irregular Domains: The Spectral Smoothed Boundary Method”. In: *SIAM J. Sci. Comput.* 28.3 (Mar. 2006), pp. 886–900. ISSN: 1064-8275. DOI: 10.1137/040607575.

[5] John Butcher. *Numerical Methods for Ordinary Differential Equations*. Englisch. 2. Auflage. Chichester, England ; Hoboken, NJ: John Wiley & Sons, Mar. 2008. ISBN: 978-0-470-72335-7.

[6] R.K. Cavin, P. Lugli, and V.V. Zhirnov. “Science and Engineering Beyond Moore’s Law”. In: *Proceedings of the IEEE* 100.Special Centennial Issue (May 2012), pp. 1720–1749. ISSN: 0018-9219. DOI: 10.1109/JPROC.2012.2190155.

[7] W.L. Chen, F.S. Lien, and M.A. Leschziner. “Local mesh refinement within a multi-block structured-grid scheme for general flows”. In: *Computer Methods in Applied Mechanics and Engineering* 144.3–4 (1997), pp. 327–369. ISSN: 0045-7825. DOI: 10.1016/S0045-7825(96)01187-5.

[8] C. W. Clenshaw and A. R. Curtis. “A method for numerical integration on an automatic computer”. en. In: *Numerische Mathematik* 2.1 (Dec. 1960), pp. 197–205. ISSN: 0029-599X, 0945-3245. DOI: 10.1007/BF01386223.

[9] R. Courant, K. Friedrichs, and H. Lewy. "Über die partiellen Differenzengleichungen der mathematischen Physik". de. In: *Mathematische Annalen* 100.1 (Dec. 1928), pp. 32–74. ISSN: 0025-5831, 1432-1807. DOI: 10.1007/BF01448839.

[10] Kiril Dichev. *PACX-MPI | hlrs.de*. Mar. 2009. URL: http://www.hlrs.de/organization/av/amt/research/pacx-mpi/ (visited on 17/16/2015).

[11] Michael Dumbser. "Arbitrary high order PNPM schemes on unstructured meshes for the compressible Navier-Stokes equations". In: *Computers & Fluids* 39.1 (Jan. 2010), pp. 60–76. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2009.07.003.

[12] Michael Dumbser and Martin Käser. "Arbitrary high order non-oscillatory finite volume schemes on u nstructured meshes for linear hyperbolic systems". In: *Journal of Computational Physics* 221.2 (Feb. 2007), pp. 693–723. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2006.06.043.

[13] Michael Dumbser et al. "A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes". In: *Journal of Computational Physics* 227.18 (Sept. 2008), pp. 8209–8253. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2008.05.025.

[14] Rodger W Dyson. "Technique for very high order nonlinear simulation and validation". In: *Journal of Computational Acoustics* 10.02 (2002), pp. 211–229.

[15] Herbert Edelsbrunner and Ernst Peter Mücke. "Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms". In: *ACM TRANS. GRAPH* 9 (1990), pp. 66–104.

[16] E.S.T. Fernandes, V.C. Barbosa, and F. Ramos. "Instruction usage and the memory gap problem". In: *Computer Architecture and High Performance Computing, 2002. Proceedings. 14th Symposium on.* 2002, pp. 169–175.

[17] J. E. Flaherty et al. "Adaptive Local Refinement with Octree Load Balancing for the Parallel Solution of Three-Dimensional Conservation Laws". In: *Journal of Parallel and Distributed Computing* 47.2 (1997), pp. 139–152. ISSN: 0743-7315. DOI: 10.1006/jpdc.1997.1412.

[18] M.J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. ISSN: 0018-9219.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. English. Stuttgart: High-Performance Computing Center Stuttgart, Sept. 2009.

[20] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. English. Stuttgart: High-Performance Computing Center Stuttgart, Sept. 2012. URL: http://mpi-forum.org/docs/docs.html.

[21] F. Franchetti et al. "Efficient Utilization of SIMD Extensions". In: *Proceedings of the IEEE* 93.2 (2005), pp. 409–425. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840491.

[22] Edgar Gabriel et al. "Distributed Computing in a Heterogeneous Computing Environment". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, 1998, p. 494.

[23] Jean-Loup Gailly and Mark Adler. *zlib*. May 2012. URL: http://zlib.net/ (visited on 07/15/2012).

[24] Bernhard Gatzhammer. *Efficient and Flexible Partitioned Simulation of Fluid-Structure Interactions*. Englisch. Dr. Hut, Feb. 2015. ISBN: 978-3-8439-1990-6.

[25] D. Gottlieb and J.S. Hesthaven. "Spectral methods for hyperbolic problems". In: *Journal of Computational and Applied Mathematics* 128.1–2 (2001). Numerical Analysis 2000. Vol. VII: Partial Differential Equations, pp. 83–131. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/S0377-0427(00)00510-0.

[26] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. en. CRC Press, Dec. 2010. ISBN: 9781439811931.

[27] John Lawrence Hammond, J. E. Brown, and S. S. Liu. *Development of a transmission error model and an error control model*. en. Rome Air Development Center, Air Force Systems Command, 1975.

[28] Daniel F. Harlacher et al. "Dynamic Load Balancing for Unstructured Meshes on Space-Filling Curves". In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*. May 2012, pp. 1661–1669. DOI: 10.1109/IPDPSW.2012.207.

[29] Daniel F. Harlacher et al. "Tree Based Voxelization of STL Data". In: *High Performance Computing on Vector Systems 2011*. Ed. by Michael Resch et al. Springer Berlin Heidelberg, 2012, pp. 81–92.

[30] Manuel Hasert et al. “Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi”. In: *Journal of Computational Science* 5.5 (Sept. 2014), pp. 784–794. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2013.11.001.

[31] John Hennessy et al. *Computer Architecture: A Quantitative Approach*. 1st. Morgan Kaufmann Publishers, Jan. 1996.

[32] William D. Henshaw and Donald W. Schwendeman. “Parallel computation of three-dimensional flows using overlapping grids with adaptive mesh refinement”. In: *J. Comput. Phys.* 227.16 (Aug. 2008), pp. 7469–7502. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2008.04.033.

[33] F. Hindenlang, T. Bolemann, and C-D. Munz. “Mesh Curving Techniques for High Order Discontinuous Galerkin Simulations”. In: *IDIHOM: Industrialization of High-Order Methods-A Top-Down Approach*. Springer, 2015, pp. 133–152.

[34] T. Hoefler, A. Lumsdaine, and W. Rehm. “Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI”. In: *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. Reno, USA: IEEE Computer Society/ACM, Nov. 2007.

[35] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual*. Roberto Ierusalimschy, Aug. 2006. ISBN: 8590379833.

[36] Takashi Ishida, Shun Takahashi, and Kazuhiro Nakahashi. “Efficient and Robust Cartesian Mesh Generation for Building-Cube Method”. In: *Journal of Computational Science and Technology* 2.4 (2008), pp. 435–446.

[37] W. Joppich and M. Kürschner. “MpCCI tool for the simulation of coupled applications”. en. In: *Concurrency and Computation: Practice and Experience* 18.2 (Feb. 2006), pp. 183–192. ISSN: 1532-0634. DOI: 10.1002/cpe.913.

[38] J. Kačur. “Method of Rothe in evolution equations”. en. In: *Equadiff 6*. Ed. by Jaromír Vosmanskỳ and Miloš Zlámal. Lecture Notes in Mathematics 1192. DOI: 10.1007/BFb0076049. Springer Berlin Heidelberg, 1986, pp. 23–34. ISBN: 978-3-540-16469-2 978-3-540-39807-3.

[39] Yehuda E. Kalay. “Determining the spatial containment of a point in general polyhedra”. In: *Computer Graphics and Image Processing* 19.4 (Aug. 1982), pp. 303–334. ISSN: 0146-664X. DOI: 10.1016/0146-664X(82)90019-3.

[40] H. Klimach and S. Roller. "Distributed Coupling for Multi-Scale Simulations". In: *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*. Ed. by P. Ivanyi and B.H.V. Topping. Civil-Comp Ltd., 2011. ISBN: 978-1-905088-44-7.

[41] Harald Klimach. *Aotus*. 2011. URL: `https://bitbucket.org/apesteam/aotus/` (visited on 10/17/2015).

[42] Harald Klimach et al. *Treelm*. 2012. URL: `https://bitbucket.org/apesteam/treelm` (visited on 10/17/2015).

[43] H. Klimach et al. *Seeder*. https://bitbucket.org/apesteam/seeder. Last accessed on 2015-06-04. 2015.

[44] Sophie von Kowalevsky. "Zur Theorie der partiellen Differentialgleichung." In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 80 (Jan. 1875), pp. 1–32. ISSN: 0075-4102. DOI: `10.1515/crll.1875.80.1`.

[45] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Leipzig: B. G. Teubner, 1901.

[46] Lew Landau and Evgeny Lifshitz. *Fluid Mechanics, Second Edition: Volume 6*. English. 2 edition. Butterworth-Heinemann, Jan. 1987. ISBN: 978-0-7506-2767-2.

[47] Jeff Lane, Bob Magedson, and Mike Rarick. "An efficient point in polyhedron algorithm". In: *Computer Vision, Graphics, and Image Processing* 26.1 (Apr. 1984), pp. 118–125. ISSN: 0734-189X. DOI: `10.1016/0734-189X(84)90133-6`.

[48] Frank G. Lether. "On the construction of Gauss-Legendre quadrature rules". In: *Journal of Computational and Applied Mathematics* 4.1 (1978), pp. 47–52. ISSN: 0377-0427. DOI: `10.1016/0771-050X(78)90019-0`.

[49] Z. Lu and A. Jantsch. "Trends of terascale computing Chips in the next ten years". English. In: *IEEE 8th International Conference on ASIC, 2009. ASICON '09*. IEEE, Oct. 2009, pp. 62–66. ISBN: 978-1-4244-3868-6. DOI: `10.1109/ASICON.2009.5351607`.

[50] Robert L. Meakin. "Composite Overset Structured Grids". In: *Handbook of Grid Generation*. CRC Press, Dec. 1998. ISBN: 978-0-8493-2687-5.

[51] Gustav Mie. "Beiträge zur Optik trüber Medien, speziell kolloidaler Metallösungen". en. In: *Annalen der Physik* 330.3 (Jan. 1908), pp. 377–445. ISSN: 1521-3889. DOI: `10.1002/andp.19083300302`.

[52] Rajat Mittal and Gianluca Iaccarino. “Immersed boundary methods”. In: *Annu. Rev. Fluid Mech.* 37 (2005), pp. 239–261.

[53] B. Moon et al. “Analysis of the clustering properties of the Hilbert space-filling curve”. In: *Knowledge and Data Engineering, IEEE Transactions on* 13.1 (2001), pp. 124–141. ISSN: 1041-4347. DOI: 10.1109/69.908985.

[54] Gordon E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/jproc.1998.658762.

[55] Philip J. Morris. “Scattering of Sound by a Sphere: Category 1, Problems 3 and 4”. In: *Proceedings of Second Computational Aeroacoustics (CAA) Workshop on Benchmark Problems*. Oct. 1997, pp. 15–17.

[56] Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. rep. IBM Ltd., 1966.

[57] C. -D. Munz et al. “Divergence Correction Techniques for Maxwell Solvers Based on a Hyperbolic Model”. In: *Journal of Computational Physics* 161.2 (July 2000), pp. 484–511. ISSN: 0021-9991. DOI: 10.1006/jcph.2000.6507.

[58] Tobias Neckel. “The PDE Framework Peano: An Environment for Efficient Flow Simulations”. Dissertation. Institut für Informatik, Technische Universität München, June 2009. ISBN: 978-3-86853-147-3.

[59] Peter J. Olver. *Applications of Lie Groups to Differential Equations*. en. Springer Science & Business Media, Jan. 2000. ISBN: 978-0-387-95000-6.

[60] Stanley Osher, Tony Chan, and Xu-dong Liu. “Weighted Essentially Non-oscillatory Schemes”. In: *Journal of Computational Physics* 115.1 (Nov. 1994), pp. 200–212. ISSN: 00219991.

[61] Brynjulf Owren and Marino Zennaro. “Order barriers for continuous explicit Runge-Kutta methods”. In: *Mathematics of Computation* 56.194 (1991), pp. 645–661. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1991-1068811-2.

[62] Anthony T Patera. “A spectral element method for fluid dynamics: laminar flow in a channel expansion”. In: *Journal of computational Physics* 54.3 (1984), pp. 468–488.

[63] Charles S. Peskin. “The immersed boundary method”. In: *Acta Numerica* 11 (Jan. 2002), pp. 479–517. ISSN: 1474-0508.

[64] L. F. Richardson. "The Approximate Arithmetical Solution by Finite Differences of Physical Problems Involving Differential Equations, with an Application to the Stresses in a Masonry Dam". In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 210.459-470 (Jan. 1911), pp. 307–357. DOI: 10.1098/rsta.1911.0009.

[65] Sabine Roller et al. "An Adaptable Simulation Framework Based on a Linearized Octree". In: *High Performance Computing on Vector Systems 2011*. Ed. by Michael Resch et al. 10.1007/978-3-642-22244-3_7. Springer Berlin Heidelberg, 2012, pp. 93–105. ISBN: 978-3-642-22244-3.

[66] P.E. Ross. "Why CPU Frequency Stalled". In: *Spectrum, IEEE* 45.4 (2008), p. 72. ISSN: 0018-9235.

[67] Feriedoun Sabetghadam, Shervin Sharafatmandjoor, and Farhang Norouzi. "Fourier Spectral Embedded Boundary Solution of the Poisson's and Laplace Equations with Dirichlet Boundary Conditions". In: *J. Comput. Phys.* 228.1 (Jan. 2009), pp. 55–74. ISSN: 0021-9991.

[68] Hans Sagan. *Space-Filling Curves*. 1st ed. Springer New York, Sept. 1994. ISBN: 0387942653.

[69] Rahul S. Sampath et al. *Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees*. 2009.

[70] Stefan Schamberger and Jens-Michael Wierum. "Graph partitioning in scientific simulations: Multilevel schemes versus space-filling curves". In: *Parallel Computing Technologies*. Springer, 2003, pp. 165–179.

[71] R. Schneiders, R. Schindler, and F. Weiler. "Octree-based Generation of Hexahedral Element Meshes". In: *Proceedings of the 5th International Meshing Roundtable* (1996), pp. 205–215.

[72] J Schreiner et al. "Aerodynamic lens system for producing particle beams at stratospheric pressures". In: *Aerosol science and technology* 29.1 (1998), pp. 50–56.

[73] T. Schwartzkopff. "Finite-Volumen Verfahren hoher Ordnung und heterogene Gebietszerlegung für die numerische Aeroakustik". In: (2005).

[74] T. Schwartzkopff, C. D. Munz, and E. F. Toro. "ADER: A High-Order Approach for Linear Hyperbolic Systems in 2D". en. In: *Journal of Scientific Computing* 17.1-4 (Dec. 2002), pp. 231–240. ISSN: 0885-7474, 1573-7691. DOI: 10.1023/A:1015160900410.

[75] Alan Silverstein. *Judy IV Shop Manual*. 2002. URL: http://judy.sourceforge.net/application/shop_interm.pdf (visited on 10/17/2015).

[76] Joseph L. Steger and John A. Benek. "On the use of composite grid schemes in computational aerodynamics". In: *Computer Methods in Applied Mechanics and Engineering* 64.1 (Oct. 1987), pp. 301–320. ISSN: 0045-7825. DOI: 10.1016/0045-7825(87)90045-4.

[77] The HDF Group. *Hierarchical data format version 5*. 2000. URL: http://www.hdfgroup.org/HDF5 (visited on 03/16/2013).

[78] Joe F. Thompson, Frank C. Thames, and C. Wayne Mastin. "Automatic numerical generation of body-fitted curvilinear coordinate system for field containing any number of arbitrary two-dimensional bodies". In: *Journal of Computational Physics* 15.3 (July 1974), pp. 299–319. ISSN: 0021-9991. DOI: 10.1016/0021-9991(74)90114-4.

[79] V. A. Titarev and E. F. Toro. "ADER schemes for three-dimensional non-linear hyperbolic systems". In: *Journal of Computational Physics* 204.2 (Apr. 2005), pp. 715–736. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2004.10.028.

[80] Tiankai Tu, David R. O'Hallaron, and Omar Ghattas. "Scalable Parallel Octree Meshing for TeraScale Applications". In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. SC '05. Washington, DC, USA: IEEE Computer Society, 2005. ISBN: 1-59593-061-2. DOI: http://dx.doi.org/10.1109/SC.2005.61.

[81] Tiankai Tu et al. "From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC '06. New York, NY, USA: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188551.

[82] Jens Utzmann. "A domain decomposition method for the efficient direct simulation of aeroacoustic problems". PhD thesis. Universität Stuttgart, 2008.

[83] J. Utzmann et al. "Heterogeneous domain decomposition for computational aeroacoustics". In: *AIAA journal* 44.10 (2006), pp. 2231–2250. ISSN: 0001-1452. DOI: 10.2514/1.18144.

[84] A. Van Oosterom and J. Strackee. "The Solid Angle of a Plane Triangle". In: *Biomedical Engineering, IEEE Transactions on* BME-30.2 (1983), pp. 125–126. ISSN: 0018-9294.

[85] J.J.W. van der Vegt and H. van der Ven. "Space–Time Discontinuous Galerkin Finite Element Method with Dynamic Grid Motion for Inviscid Compressible Flows: I. General Formulation". In: *Journal of Computational Physics* 182.2 (Nov. 2002), pp. 546–585. ISSN: 0021-9991. DOI: 10.1006/jcph.2002.7185.

[86] J. H. Verner. "On deriving explicit Runge-Kutta methods". en. In: *Conference on Applications of Numerical Analysis*. Ed. by John Ll Morris. Lecture Notes in Mathematics 228. Springer Berlin Heidelberg, Jan. 1971, pp. 340–347. ISBN: 978-3-540-05656-0, 978-3-540-36976-9.

[87] Michael Weiss. "Strip mining on SIMD architectures". In: *Proceedings of the 5th international conference on Supercomputing*. ACM. 1991, pp. 234–243.

[88] Elizabeth White. *What Is An STL File?* Text. Nov. 2013. URL: http://www.3dsystems.com/quickparts/learning-center/what-is-stl-file (visited on 06/04/2015).

[89] H. Wilbraham. "On a Certain Periodic Function". In: *The Cambridge and Dublin Mathematical Journal* 3 (1848), pp. 198–201.

[90] Hung-Hsi Wu. "Historical development of the Gauss-Bonnet theorem". In: *Science in China Series A: Mathematics* 51.4 (Apr. 2008), pp. 777–784. DOI: 10.1007/s11425-008-0029-8.

[91] Mark A. Yerry and Mark S. Shephard. "Automatic three-dimensional mesh generation by the modified-octree technique". en. In: *International Journal for Numerical Methods in Engineering* 20.11 (Nov. 1984), pp. 1965–1990. ISSN: 1097-0207.

[92] Jens Zudrop and Jan S. Hesthaven. "Accuracy of high order and spectral methods for hyperbolic conservation laws with discontinuous solutions". In: *SIAM Journal on Numerical Analysis* 53.4 (2015), pp. 1857–1875.

[93] Gerhard Zumbusch. "On the quality of space-filling curve induced partitions". In: *Sonderforschungsbereich 256* (2000), pp. 25–28.

List of Figures

List of Tables

Simulation Techniques in Siegen
Vol. 1

This issue presents the development of scalable simulation tools for fluid flows and especially methods to compute direct aeroacoustic simulations.
Aero-acoustic phenomena pose a multi-scale problem. To tackle this class of problems, a general coupling tool for structured and unstructured meshes is first parallelized to scale on thousands of processes. However, as there remains a principle bottleneck in the treatment of meshes with this approach, an entirely new framework on the basis of octrees is then developed.
This restriction in the mesh representation allows for fully parallel simulations of arbitrary large setups.

Harald Klimach studied aerospace engineering at the University of Stuttgart; worked at the HLRS on porting and optimization of user applications on a wide range of supercomputers and started the development of the APES framework at the GRS in Aachen. Since 2013 he is working in research and teaching at the chair for Simulation Techniques and Scientific Computing of the University of Siegen.

The series ***Simulation Techniques in Siegen*** presents contributions to the field of scientific computing with a focus on the utilization of large-scale computing systems for highly resolved simulations. Applications, as well as numerical methods and their efficient implementation on modern supercomputers, are investigated and described.

ISBN 978-3-936533-82-8

universi
UNIVERSITÄTSVERLAG SIEGEN