Communication-Avoiding Algorithms for a High-Performance Hyperbolic PDE Engine

CHARRIER, DOMINIC,ETIENNE

Communication-Avoiding Algorithms for a High-Performance Hyperbolic PDE Engine

Dominic Etienne Charrier

The study of waves has always been an important subject of research. Earthquakes, for example, have a direct impact on the daily lives of millions of people while gravitational waves reveal insight into the composition and history of the Universe. These physical phenomena, despite being tackled traditionally by different fields of physics, have in common that they are modelled the same way mathematically: as a system of hyperbolic partial differential equations (PDEs). The ExaHyPE project ("An Exascale Hyperbolic PDE Engine") translates this similarity into a software engine that can be quickly adapted to simulate a wide range of hyperbolic partial differential equations. ExaHyPE's key idea is that the user only specifies the physics while the engine takes care of the parallelisation and the interplay of the underlying numerical methods. Consequently, a first simulation code for a new hyperbolic PDE can often be realised within a few hours. This is a task that traditionally can take weeks, months, even years for researchers starting from scratch.

My main contribution to ExaHyPE is the development of the core infrastructure. This comprises the development and implementation of ExaHyPE's solvers and adaptive mesh refinement procedures, it's MPI+X parallelisation as well as high-level aspects of ExaHyPE's application-tailored code generation, which allows to adapt ExaHyPE to model many different hyperbolic PDE systems. Like any high-performance computing code, ExaHyPE has to tackle the challenges of the coming exascale computing era, notably network communication latencies and the growing memory wall. In this thesis, I propose memory-efficient realisations of ExaHyPE's solvers that avoid data movement together with a novel task-based MPI+X parallelisation concept that allows to hide network communication behind computation in dynamically adaptive simulations.

Communication-Avoiding Algorithms for a High-Performance Hyperbolic PDE Engine

Dominic Etienne Charrier

A Thesis Submitted for the Degree of *Doctor of Philosophy* (Ph.D.) in Computer Science

Department of Computer Science

Durham University

June 2019

Table of Contents

1 Introduction **1**

1.1 Exascale Computing Challenges 6

1.2 Challenges for ExaHyPE 9

1.3 Scientific Contributions 11

1.4 Publications 13

1.5 Thesis Structure 13

2 Numerical Methods for Hyperbolic PDEs **15**

2.1 Solving Hyperbolic Partial Differential Equations 16

2.2 Finite Volumes Methods 20

2.3 The ADER-DG Method 25

2.4 A Posteriori Subcell Limiting for ADER-DG 30

2.5 Discussion 36

3 Engine Design and Toolset **38**

3.1 The ExaHyPE Core 40

3.2 Toolkit, Optimised Kernels, and Debugging/Benchmarking Ecosystem 41

4 Example Usage: Solving the Euler Equations **44**

4.1 Adaptivity and Parallelisation 52

5 Related Work **56**

5.1 Numerical Methods for Hyperbolic PDEs 58

5.2 ExaHyPE in the Scientific Computing Landscape 64

6 Adaptive Mesh Refinement **67**

6.1 Parallel Adaptive Spacetree Meshes . . . 69
6.2 The Spacetree as Meta Data Structure . . . 72
6.3 Eliminating Local Master-Worker Communication . . . 87
6.4 Pre-Refinement Techniques . . . 88
6.5 Inter-Grid Transfer Operators . . . 91
6.6 Discussion . . . 97

7 Limiting Hybrid ADER-DG-FVM 99
7.1 A Hybrid ADER-DG-FVM Solver . . . 101
7.2 Dynamic Curing . . . 105
7.3 Limiter-Criteria-Guided Refinement . . . 109
7.4 Discussion and Outlook . . . 111

8 Communication-Avoiding Low-Storage ADER-DG 116
8.1 Memory Analysis of the Baseline . . . 118
8.2 Low-Storage ADER-DG . . . 124
8.3 Communication-Avoiding ADER-DG . . . 129
8.4 Theoretical Comparison . . . 134
8.5 Experimental Comparison . . . 135
8.6 Discussion . . . 141

9 Hybrid Parallelisation 143
9.1 Hybrid Parallelisation via Recursion Unrolling . . . 145
9.2 Enclave Tasking . . . 150
9.3 Experimental Evidence . . . 155
9.4 Summary . . . 158

10 Multi-Node Performance Studies 160
10.1 Applications . . . 161
10.2 Results . . . 165
10.3 Discussion . . . 165

11 Conclusion 172
11.1 Summary . . . 172

11.2 Outlook 174

Declaration

This thesis was conducted as part of the ExaHyPE project. The author's work focused on creating the interface between the Peano framework and the applications developed by the application scientists. This task involved designing the principal program flow and algorithms of ExaHyPE plus major parts of the glue-code generation that plugs user applications into the engine.

Copyright

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). I also acknowledge support and computing resources provided by the Leibniz Supecomputing Centre (grant nos pr48ma & pr83no). This work made use of the facilities of the Hamilton HPC Service of Durham University.

Thanks are due to all members of the ExaHyPE consortium who made this research possible—in particular to Leonhard Rannabauer, Anne Reinarz, Kenneth Duru, Luke Bovard, and Sven Köppel for their work on the geophysics and astrophysics applications and Jean-Matthieu Gallard for his work on the optimised compute kernels for ADER-DG. I am very grateful to the ExaHyPE PIs Michael Bader, Michael Dumbser, Alice Gabriel, Luciano Rezzolla, and Tobias Weinzierl who enabled this research.

A special thank you goes to my supervisor Tobias Weinzierl for his time and dedication throughout the past three and a half years. His skilfulness in rethinking algorithms, his exhaustive knowledge on essentially all aspects of applied maths and computer science, and his hands-on attitude towards performance engineering really impressed and inspired me.

I wrote this thesis in the `reStructuredText` format instead of LaTeX (http://docutils.sourceforge.net/rst.html). The python-based document builder `Sphinx` then generated this document (http://www.sphinx-doc.org/). I am grateful to Jeff Terrace for sharing `Sphinx` extensions that allow to create side-by-side figures with `Sphinx` (http://jterrace.github.io/sphinxtr).

Thank you to all the people that supported me during my time at Durham University. Lei Fan, Konstantinos Krestinitis, Robert Bird, Eléonore Gaudin — I am grateful I had such wonderful office mates and friends. Another big thank you to my flatmate Joshua Roffe and everyone else I met at Ustinov college. Finally, I want to thank my family, my girlfriend Kayla, and her family for their support and love throughout the past three and a half years.

Dominic Etienne Charrier
Durham, 20 June 2019

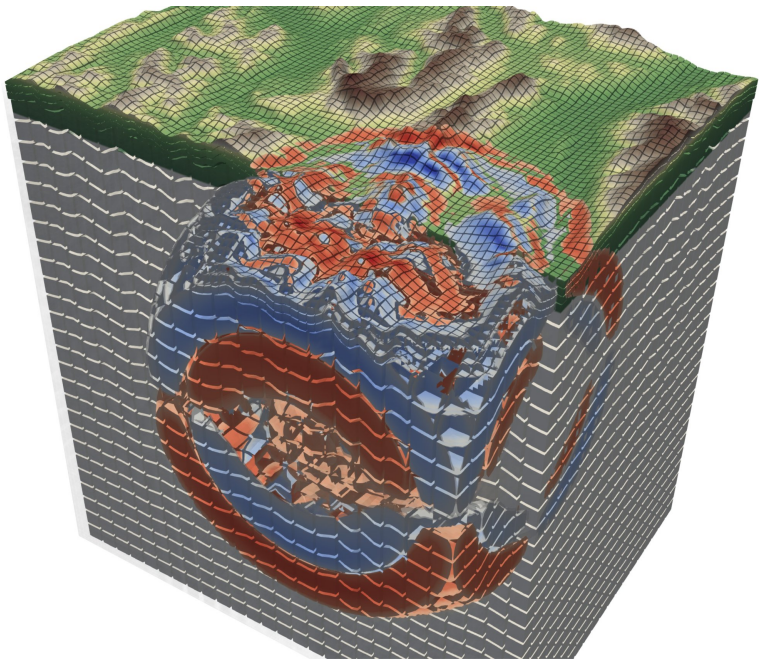
1 Introduction

Understanding wave phenomena has enabled advance in science and engineering. It has allowed us to develop new technology, to understand and reduce risks to our life and property, and to gain insight into the creation process of the Universe. This list is not comprehensive. Mathematically, waves are modelled as hyperbolic PDE (partial differential equation) systems. The PDE systems considered in this thesis are specialisations of this type. We focus on first order systems:

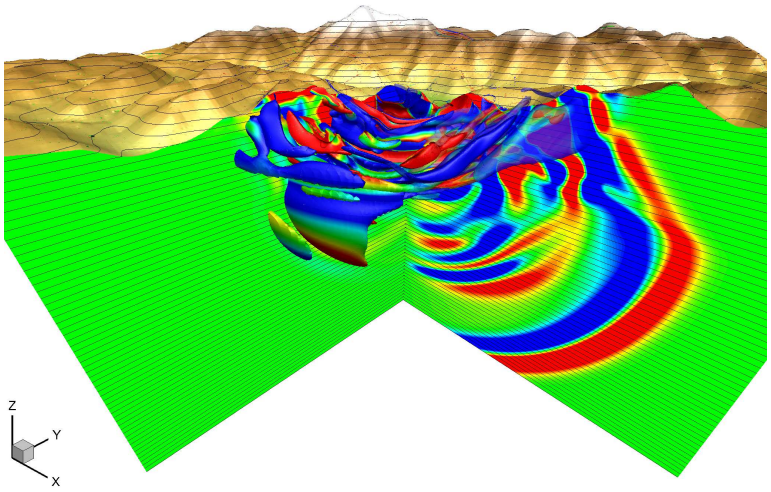
$$\frac{\partial q}{\partial t} + \nabla \cdot F(q) + \sum_i A_i(q) \frac{\partial q}{\partial x_i} = S(q), \tag{1.1}$$

where $q = q(x,t)$ are the typically vector-valued *state variables*, F is the *conservative flux* (a tensor), the A_i are the *coefficient matrices* for computing non-conservative products with i being the space dimension, and S is the *source term*.

Solutions q that satisfy (1.1) can be derived analytically under idealised conditions. However, for problem formulations that consider less ideal conditions such as heterogeneous material, complicated domains, or nonlinearities, it becomes very difficult or impossible to find analytic solutions. Computer simulations are employed to obtain approximate solutions. The insight that can be obtained with them is limited by the used algorithms and computer hardware. To gain the maximum insight with finite compute power, algorithms must take into account



(a) Geometry-Aligned Mesh



(b) Diffuse Interface Method

Fig. 1.1: Velocity magnitude of an earthquake wave travelling through the Alps: In these ExaHyPE simulations, complicated topography is modelled via (a) a geometry-aligned mesh or (b) a diffuse interface. *Reprinted from* [83] and [92], respectively.

that physical processes span over multiple spatial and temporal scales. Moreover, they must be tailored to the hardware to reduce the runtime and cost of complex simulations to realistic levels.

> The primary outcome of this thesis is the development of the core components of the software engine ExaHyPE ("An Exascale Hyperbolic PDE Engine") for simulating waves. ExaHyPE solves large-scale scientific problems with dynamically adapted mesh resolution on supercomputers. The engine allows the rapid development of simulation codes that solve hyperbolic PDE systems in first-order form. ExaHyPE's strategy is to create efficient algorithms that are tailored to the hardware while it hides this complexity from the engine users. Engine users are only required to specify the application-specific part, i.e. the physics.

A wide range of applications have been implemented with ExaHyPE; see [83]. In this thesis, I will focus on applications from seismology and fluid dynamics.

Seismology

The linear elastic wave equations can be used to model the propagation of earthquake waves [67]. They take the form:

$$\begin{aligned} \frac{\partial \sigma}{\partial t} - E(\lambda, \mu) \cdot \nabla v &= S_\sigma, \\ \frac{\partial v}{\partial t} - \frac{1}{\rho} \nabla \cdot \sigma &= S_v, \end{aligned} \tag{1.2}$$

where σ is the stress tensor, ρ and v are mass density and velocity, respectively. The stress tensor is symmetric, i.e. the PDE has the 9 unknowns $q = (\sigma_{xx}, \sigma_{xy}, \sigma_{xz}, \sigma_{yy}, \sigma_{yz}, \sigma_{zz}, v_x, v_y, v_z)^T$. Furthermore, S_ρ and S_v are volume sources, and $E(\lambda, \mu)$ is the stiffness tensor which relates material strain to the stress tensor. All quantities depend on the spatial coordinates; the stiffness tensor depends on them via the material parameters, the Lamé coefficients λ and μ. Equation (1.2) does not make any assumptions on the topography, i.e. the shape of the interface between solid and surrounding air. Topography and material distribution are often approximated with a geometry-aligned computational mesh (Fig. 1.1 (a)). Alternatively,

topography information can be integrated into the PDE system via additional equations [92]:

$$\begin{aligned} \frac{\partial \sigma}{\partial t} - E(\lambda,\mu) \cdot \frac{1}{\alpha} \nabla(\alpha\, v) + \frac{1}{\alpha} E(\lambda,\mu) \cdot v \otimes \nabla\alpha = S_\sigma, \\ \frac{\partial \alpha v}{\partial t} - \frac{\alpha}{\rho} \nabla \cdot \sigma - \frac{1}{\rho} \sigma \nabla\alpha = S_v, \\ \frac{\partial \alpha}{\partial t} = 0, \qquad \frac{\partial \lambda}{\partial t} = 0, \qquad \frac{\partial \mu}{\partial t} = 0, \qquad \frac{\partial \rho}{\partial t} = 0, \end{aligned} \tag{1.3}$$

where the scalar diffuse interface parameter α models topography as a smooth function. This approach allows solving the equations numerically with computational meshes that are not aligned with the topography. I refer to [92] for additional details on the derivation of the model.

Leonhard Rannabauer kindly provided the implementation of the solvers for the linear elastic wave equations that I use in this thesis. The diffuse interface method relies on more advanced features of ExaHyPE; however, it might require less memory than the geometry-aligned method, which has to store Jacobian matrix and determinant for every mesh element. A comparison of both methods in terms of their performance and memory footprint is thus a very interesting study.

Fluid Dynamics

The compressible Euler equations take the form of a hyperbolic conservation law as only the flux tensor $F(q)$ is non-trivial:

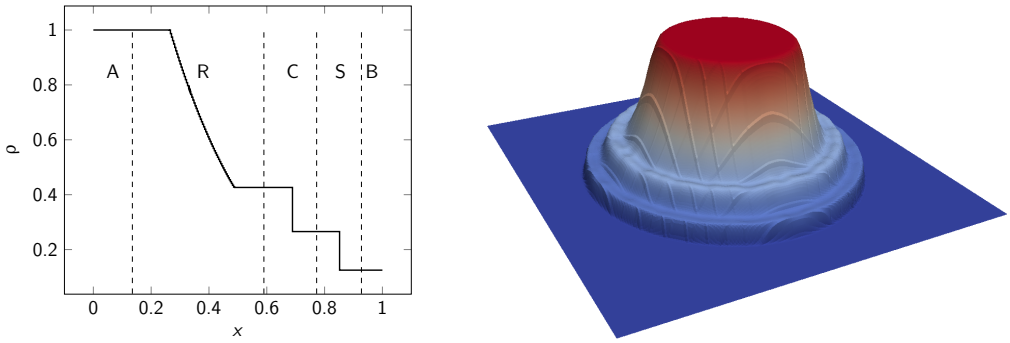
$$\frac{\partial q}{\partial t} + \nabla \cdot F(q) = 0. \tag{1.4}$$

The conserved state variables $q = (\rho, \rho\, v, \rho\, E)^T$ are constructed from the primitive variables ρ, v, and E. These symbols denote mass density, velocity, and energy density, respectively. The nonlinear flux is given as

$$F = \begin{pmatrix} \rho v \\ \rho\, v \otimes v + P\, I_{d\times d} \\ v(\rho E + P) \end{pmatrix}, \tag{1.5}$$

To close (1.4) and (1.5), I use a pressure P according to the EOS (equation of state) of a perfect gas with adiabatic index γ,

$$P = (\gamma - 1)\,(\rho E - \frac{1}{2}\,\rho v \cdot v).$$



(a) Sod's Shock Tube

(b) 2D Explosion ("2D Sod")

Fig. 1.2: (a) Sod's shock tube for the compressible Euler equation describes a Riemann problem with initial states $q_A = q(x < 0.5, 0) = (1.0, 0, \ldots, 1.0)^{\mathrm{T}}$ and $q_B = q(x > 0.5, 0) = (0.125, 0, \ldots, 0.1)^{\mathrm{T}}$. The analytically computed profile of the density is shown at $t = 0.2$ non-dimensional time. In region R a smooth rarefaction wave is present. In region C, a contact discontinuity is present. In contrast to region C, a pressure jump aligns with the density jump in region S (not shown). Therefore, in region S, a shock wave is present. (b) shows a snapshot of a simulation of a spherical explosion at $t = 0.1$ non-dimensional time. The 2D density profile exhibits the same three characteristic wave types.

The square matrix $I_{d\times d}$ of size d is the identity matrix. The compressible Euler equations are used to model flow in compressible gases and fluids. Note that (1.4) states the compressible Euler equations in the strong form, which assumes that the flux is differentiable everywhere (via the divergence operator). More general formulations of the problem are *integral* or *weak* fomulations One integral formulation is the following, which can be recovered from the strong formulation (1.4) via a Green's theorem:

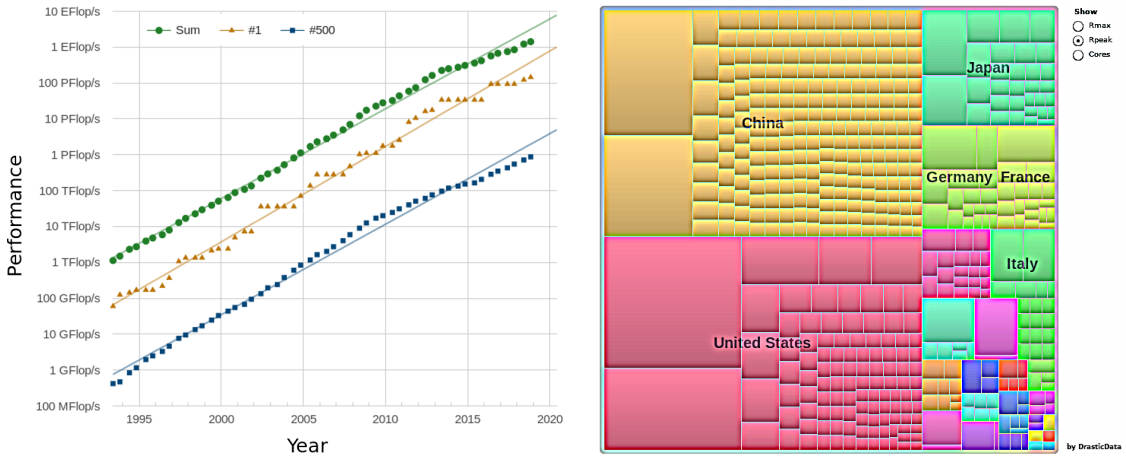
$$\int_\Omega \frac{\partial q}{\partial t} \mathrm{d}x + \int_{\partial\Omega} F(q)\, n \,\mathrm{d}s = 0, \tag{1.6}$$

where Ω is a volume, $\partial\Omega$ its hull, n is the outward directed normal vector and $\mathrm{d}s$ is the infinitesimal surface element. This integral formulation of the problem relaxes the assumption on the differentiability of the solution in space (via the flux). Aside from the smooth waves that the strong fomulation admits too, e.g. rarefaction waves, (1.6) admits a second class of solutions (Fig. 1.2): Propagating discontinuities such as contact discontinuities and shock waves. Compared to contact discontinuities, not only a density but also a pressure jump is present at shock fronts. Therefore, fluid particles flow from the low pressure domain into the high pressure domain. Shock waves can be generated even from smooth initial conditions. Both discontinuities, especially shock waves, are challenging for computer simulations.

All features of ExaHyPE are necessary to solve the compressible Euler equations: High-order approximation techniques with low numerical diffusion are required to model the flow in smooth areas, while nonlinear stability mechanisms are necessary to prevent spurious features in the solution around shock waves and other discontinuities. The latter add numerical diffusion that must be localised with AMR (adaptive mesh refinement). As they are comparably small with 5 coupled equations, the compressible Euler equations have been a particularly useful application for the initial development stages of ExaHyPE. The small number of equations implies low memory requirements and low computational cost when solving these equations with computer simulation codes. Important features of ExaHyPE such as AMR and treatment of shock waves could be developed and tested on a laptop or small workstation.

1.1 Exascale Computing Challenges

The term *Exascale* in ExaHyPE's name refers to exascale computing. *Exa* is the decimal unit prefix for $1000^6 = 10^{18}$, i.e. for a quintillion or a "billion billion". This is the number of floating point operations (flop) that the coming generation of supercomputers will be able to perform



(a) Projected Performance Development

(b) Supercomputing Countries

Fig. 1.3: (a) Projected development of theoretical peak performance of supercomputers. (b) Number of supercomputers per country and their theoretical peak performance (Rpeak), which is visualised by the size of the boxes. *Graph and diagram taken from TOP500 webpage and adapted* [90] .

per second. Exascale supercomputers promise to be the key to ground-breaking new research. The leading countries in HPC (High Performance Computing) (Fig. 1.3 (b)), are investing billions of dollars to become first in the race to exascale [116][114]. Having such compute power at hand will allow them to take leadership positions in many scientific disciplines and industries. The United States lead the race at the moment: Their supercomputer Frontier [115] is expected to be ready in year 2021. It is designed to have a peak performance of 1.5 exaflop per second.

Building such a massively parallel exascale machine is not the only isolated challenge. Software must be prepared for the exascale era, too. Already with today's petascale machines, it is difficult to write scientific software that reaches performance levels close to their theoretical peak. The majority of HPC applications cannot exploit more than a fraction of it. Take for example the HPCG (High-Performance Conjugate Gradient) benchmark [46]. It models the requirements of state-of-the-art solvers for elliptic PDE systems. When running HPCG on the fastest machines in the TOP500 list, they deliver only 1.5 % of their theoretical peak performance [90].

The picture looks completely different if a dense linear algebra code such as HPL (High-

Performance Linpack) [45][110] is run on these machines. Most machines achieve around 80 % of their theoretical peak performance in this benchmark. Not coincidentally, HPL is the gold standard for comparing the performance of installed machines against the theoretical peak performance that system integrators promise [90]. It is natural to ask why HPL performs so much better than HPCG on today's supercomputer architectures. The answer lies in the memory access patterns the codes exhibit. HPL's computations are dominated by dense matrix-matrix products, which result in very regular data access patterns [46]. In this case, modern hardware is very efficient in preloading data before it is used by the processing units. Moreover, the arithmetic intensity that HPL exhibits is high, i.e. HPL performs many floating point operations per byte that it loads from memory.

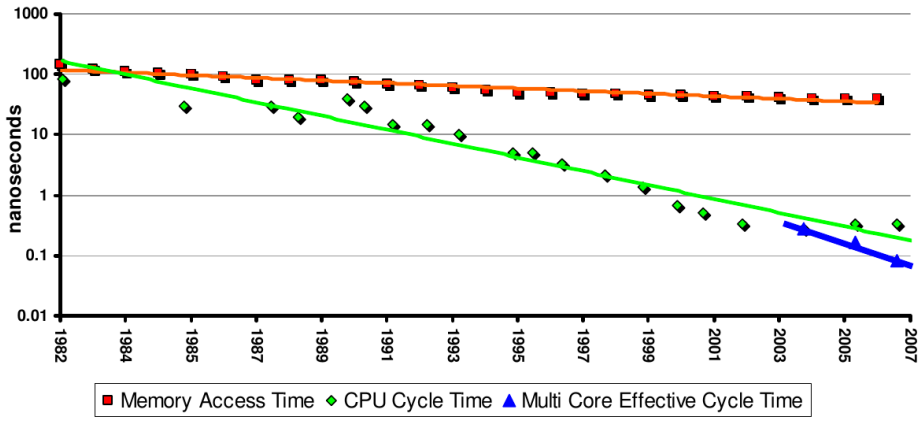


Fig. 1.4: The memory wall. Memory access times did not improve at the same rate as CPU cycle times in the past years. Although CPU frequency scaling has hit a wall now too, the gap is far from closed. In fact, modern multi-core CPUs increase the pressure on the memory subsystem even further (blue line). *From* [73].

HPCG, in contrast, performs mostly linear algebra operations with low arithmetic intensity, notably scalar products. The code spends proportionally more of its runtime with loading data from memory than HPL. At the same time, data access is less regular due to the usage of sparse matrices. Compared to HPL, the applications performance is more shaped by memory latency and bandwidth. Hence, HPCG is more affected by the *memory wall* than HPL [73]. This term was coined to describe that memory access times did not decrease at the same rate as the duration of a CPU (central processing unit) cycle over the past years (Fig. 1.4). As a consequence, the CPU has to wait hundreds of cycles until data that is not found in the

caches is loaded from main memory. Regular data access patterns that allow preloading data before they are used are essential to keep the CPU cores busy. The majority of scientific applications have runtime behaviour that is more similar to that of HPCG than to HPL, while today's supercomputer architectures are tailored to codes such as HPL that exhibit very regular memory access patterns and have high arithmetic intensity.

> As exascale supercomputer designs such as Frontier [115] do not present radical new approaches to close the gap between CPU and memory performance, software must be tailored to the hardware characteristics instead. Algorithms must be (re-)designed such that arithmetic intensity is high and inter-process communication is hidden or avoided. This is already true for the current petascale era but will be of even higher importance in the exascale era [73].

1.2 Challenges for ExaHyPE

One key idea of ExaHyPE is that the users focus on the physics while all HPC aspects such as parallelisation and compute kernel optimisations are taken care of by the engine. Users build their applications on a code base that is maintained by HPC specialists who tackle the challenges the exascale era brings. Starting from this goal, ExaHyPE has to tackle multiple challenges to gain acceptance in the targeted scientific communities. On the one hand, there are algorithmic and performance engineering challenges which are closely related to the challenges any scientific software has to face to successfully enter the exascale era. On the other hand, there are software engineering challenges due to ExaHyPE's aspirations to be a useful generic and extensible tool that can be applied to hyperbolic PDE problems from a large variety of scientific disciplines. Due to its engine character, simulation codes built upon ExaHyPE inherit the characteristics of ExaHyPE's building blocks. If the building blocks are not there, applications built upon them will not perform.

ExaHyPE is built around the ADER-DG (Arbitrary DERivative Discontinuous Galerkin) method, a discretisation and time stepping scheme for the evolution of hyperbolic PDE systems that performs only a single neighbour communication step per time step. ADER-DG is considered a good fit to HPC as it has dominating algorithmic steps with high arithmetic

intensity that allow to hide communication well [51]. Highly optimised realisations of the method are able to achieve up to 60 % of the peak performance on petascale supercomputers [63]. ADER-DG is a predictor-corrector method. It has a large memory footprint as it has to store a *predictor* vector in addition to the solution vector. To save memory and computational cost, ExaHyPE employs dynamic AMR, i.e. the method's accuracy is only increased in areas where interesting features are present in the solution. Moreover, ExaHyPE does not realise the plain ADER-DG method. Instead, it adds further stability mechanisms on top: To increase the robustness of ADER-DG in the vicinity of shock waves and other discontinuities, ExaHyPE teams it up with a more robust finite volume method. After the ADER-DG time step, the solver rolls back in time locally where those solution features cause numerical instabilities and employs a more robust (though less computationally efficient) finite volumes method. The resulting method is called *a posteriori limiting* ADER-DG [54] as problems are cured a posteriori, i.e. after the ADER-DG time step. Dynamic AMR and limiting increase the complexity of ADER-DG and make it more sensitive to memory and network latencies. At the same time, computational cost and memory demands become difficult to predict.

I see the following challenges that must be solved to prepare ExaHyPE's algorithms for the exascale era:

1. ExaHyPE's ADER-DG method must perform aggressive AMR to keep the large persistent memory footprint and bandwidth demand of ADER-DG under control.
2. ExaHyPE's numerical methods must be implemented such that communication phases are overlapped with computation phases.
3. ExaHyPE's numerical methods must be formulated such that their access to the caches and main memory is minimal. Tasks and eventually loops must be fused where possible in order to increase arithmetic intensity and scalability of the methods.

In addition, the following performance engineering challenges must be addressed:

4. ExaHyPE is built upon the PDE framework Peano [103], which has been mainly employed for programming solvers for low-order elliptic PDE systems. In contrast to ExaHyPE's numerical methods, these methods use significantly larger computational meshes. Design decisions made during the development of frameworks are often biased by the requirements of the primarily targeted applications. Therefore, Peano's applicability

and performance must be assessed with respect to ExaHyPE's requirements.

5. A sole geometrically inspired load balancing scheme is not suitable for ExaHyPE as the computational work and memory footprint of ExaHyPE's numerical methods is difficult to predict. Moreover, performance fluctuations among compute nodes are expected to be common in exascale machines. A second load balancing layer must be added on top that allows to move work from overloaded ranks temporarily to ranks with less work. ExaHyPE must be designed to support such a feature.

6. The compute kernels of ExaHyPE's numerical methods must be optimised towards the targeted supercomputing architectures. This must be done in a generic way that is applicable to most PDE systems and architectures targeted by the engine.

This last performance engineering evergreen will not be addressed further in this thesis as it is tackled by ExaHyPE collaborators at Technical University of Munich. I rely on their work in this thesis to ensure that the single-core performance of ExaHyPE is at a competitive level that allows conducting scalability studies. Lastly, I see the following classic software engineering challenges:

7. The engine must be able to serve the PDE modelling needs of different communities: Not all applications require all PDE terms; it must be possible to switch certain compute kernels off. Certain applications do not require limiting at all (linear elasticity with geometry-aligned meshes) while it is essential for others (compressible Euler).

8. A workflow must be developed that allows users to realise new hyperbolic PDE solvers quickly. Users should only be concerned with the physics. Building applications with ExaHyPE must hide the complexities of numerical methods and their parallelisation from its users.

1.3 Scientific Contributions

In order to tackle the challenges formulated in the previous section, I make the following scientific contributions:

1. I developed all high-level algorithms and solvers (which includes their parallelisation) plus a major part of the user interface of a generic software engine for simulating hyperbolic PDEs. The software is presented in [83] and has been successfully applied to

applications of practical importance [92][75]. The engine offers dynamically adaptive mesh refinement and a hybrid distributed-memory shared-memory parallelisation. User applications can conveniently switch on features and add PDE terms via a configuration file.

2. I proposed efficient realisations of ExaHyPE's ADER-DG method that significantly reduce memory footprint and memory access of the method (Chapter 8). A preprint on these realisations is available from [39]. In [37], we present a node-level performance analysis for them in the context of systems with deep memory hierarchy, where one additional layer of persistent but fast memory is placed between main memory and hard drive.

3. We developed a tasking technique for dynamically adaptive simulations that prioritises tasks along partition boundaries and mesh resolution transitions [38]. The technique allows to explicitly overlap communication and computation without sacrificing a whole CPU core for communication. Furthermore, we avoid a task graph assembly completely. In this thesis, I describe how we implement the technique for ExaHyPE's ADER-DG method.

4. I implemented a posteriori subcell limiting ADER-DG as a hybrid ADER-DG-FVM (finite volume method) solver. (Chapter 7). This reduces the communication steps of the method by up to a factor of 4. Moreover, the solver reuses the limiter criteria as refinement criterion. We use this hybrid solver in [92].

5. I developed a mesh data structure based on Peano plus intra-grid transfer operators that allow unconstrained mesh adaptivity for the ADER-DG method. Moreover, I transferred the a posteriori solution correction idea of the a posteriori subcell limiting ADER-DG method to adaptive mesh refinement (see Chapter 6 and Chapter 7). The resulting mesh adaptation procedure might be an alternative to pre-refinement techniques that rely on estimates to decide where to pre-refine the mesh.

6. I assess the meshing framework Peano [103] for parallel dynamically adaptive mesh refinement in the context of high-order finite element type discretisations. (see Chapter 6 and Chapter 9)

1.4 Publications

The following peer-reviewed articles are related to this thesis:

1. D. E. Charrier and T. Weinzierl, "An Experience Report on (Auto-)tuning of Mesh-Based PDE Solvers on Shared Memory Systems," in Parallel Processing and Applied Mathematics, vol. 10778, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer International Publishing, 2018, pp. 3–13.
2. M. Tavelli et al., "A simple diffuse interface approach on adaptive Cartesian grids for the linear elastic wave equations with complex topography," Journal of Computational Physics, vol. 386, pp. 158–189, 2019.
3. D. E. Charrier et al., "Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver", The International Journal of High Performance Computing Applications, Apr. 2019, 2018, pp. 3–13.
4. A. Reinarz et al., "ExaHyPE: An Engine for Parallel Dynamically Adaptive Simulations of Wave Problems," Submitted to Computer Physics Communications (accepted with minor corrections), Preprint: arXiv:1905.07987 [cs, math], May 2019.

The following preprints cover aspects of this thesis:

1. D. E. Charrier and T. Weinzierl, "Stop talking to me – a communication-avoiding ADER-DG realisation," arXiv:1801.08682 [cs], Jan. 2018.
2. D. E. Charrier, B. Hazelwood, and T. Weinzierl, "Enclave Tasking for Discontinuous Galerkin Methods on Dynamically Adaptive Meshes," arXiv:1806.07984 [cs], Jun. 2018.
3. P. Samfass et al., "Tasks Unlimited Lightweight Task Offloading Exploiting MPI Wait Times for Parallel Adaptive Mesh Refinement", May 2019.

This thesis contributed to the development of the open-source software ExaHyPE, which is freely available from www.exahype.org.

1.5 Thesis Structure

Chapter 2 introduce vanilla versions of ExaHyPE's numerical methods, which I use as foundation for discussing the optimisations that I propose in the research chapters of this

thesis. A brief overview of ExaHyPE's architecture is then given in Chapter 3. Chapter 4 quickly demonstrates the principal usage of ExaHyPE by realising a parallel adaptive solver for a nonlinear problem with less than 100 lines of code. It follows a discussion how ExaHyPE relates to other software in Chapter 5. In Chapter 6, I present ExaHyPE's dynamic AMR implementation before I dive into the implementation of ExaHyPE's a posteriori limiting ADER-DG method, which I implement as a hybrid ADER-DG-FVM method in Chapter 7. I propose communication-avoiding low-storage variants of ADER-DG in Chapter 8 before I present ExaHyPE's distributed-memory and shared-memory parallelisation in Chapter 9. I employed all developed techniques to run large-scale simulations. The results are examined in Chapter 10. Finally, Chapter 11 concludes this thesis with a summary and a discussion of major findings plus ideas for future research directions.

2

Numerical Methods for Hyperbolic PDEs

In this chapter, I give an overview of the numerical methods that ExaHyPE uses to solve hyperbolic PDE systems. Written from a computer science perspective, I focus on their program flow and communication patterns. This chapter provides the foundation for discussing variants of these numerical methods that minimise two aspects of communication, network communication and data movement between processing units and memory. Communication is considered the main road block for software to reach peak performance on exascale machines.

Structure

The first section briefly reiterates the equation systems solved by ExaHyPE and the optimal flow that ExaHyPE's numerical time stepping methods should realise to minimise communication. In the next sections, I formalise the numerical methods FVM (Section 2.2) and ADER-DG (Section 2.3) that we use in ExaHyPE. Section 2.4 discusses the coupling of ADER-DG and FVM to construct the a posteriori subcell limiting ADER-DG method. I clarify what the strengths of all three methods are with respect to the modelled wave phenomena and express straightforward realisations of the methods as pseudocode. This allows a comparison of the algorithms against optimal algorithms and among themselves in terms of communication requirements.

2.1 Solving Hyperbolic Partial Differential Equations

ExaHyPE provides numerical methods for solving hyperbolic PDE systems that can be expressed in the following strong form:

$$\frac{\partial q}{\partial t} + \nabla \cdot F(q) + \sum_{i=1}^{d} A_i(q) \frac{\partial q}{\partial x_i} = S(q), \tag{2.1}$$

where $q\colon \mathbb{R}^d \times \mathbb{R}_0^+ \to \mathbb{R}^v$ are the vector-valued *state variables*, $F(q)\colon \mathbb{R}^d \times \mathbb{R}_0^+ \to \mathbb{R}^{v\times d}$ is the *conservative flux*, and $A_i(q)\colon \mathbb{R}^d \times \mathbb{R}_0^+ \to \mathbb{R}^{v\times v}$ are the coefficient matrices of the *non-conservative product*, where i is a space dimension and d the number of space dimensions. Furthermore, $S(q)\colon \mathbb{R}^d \times \mathbb{R}_0^+ \to \mathbb{R}^v$ is the *algebraic source term*. The argument brackets indicate that $A_i(q)$, $S(q)$, and $F(q)$ may depend on q. In (2.1), $F(q)$ is subject to the tensor divergence $\nabla \cdot (\cdot) = \sum_{j=1}^{d} \frac{\partial (\cdot)_{ij}}{\partial x_j}$.

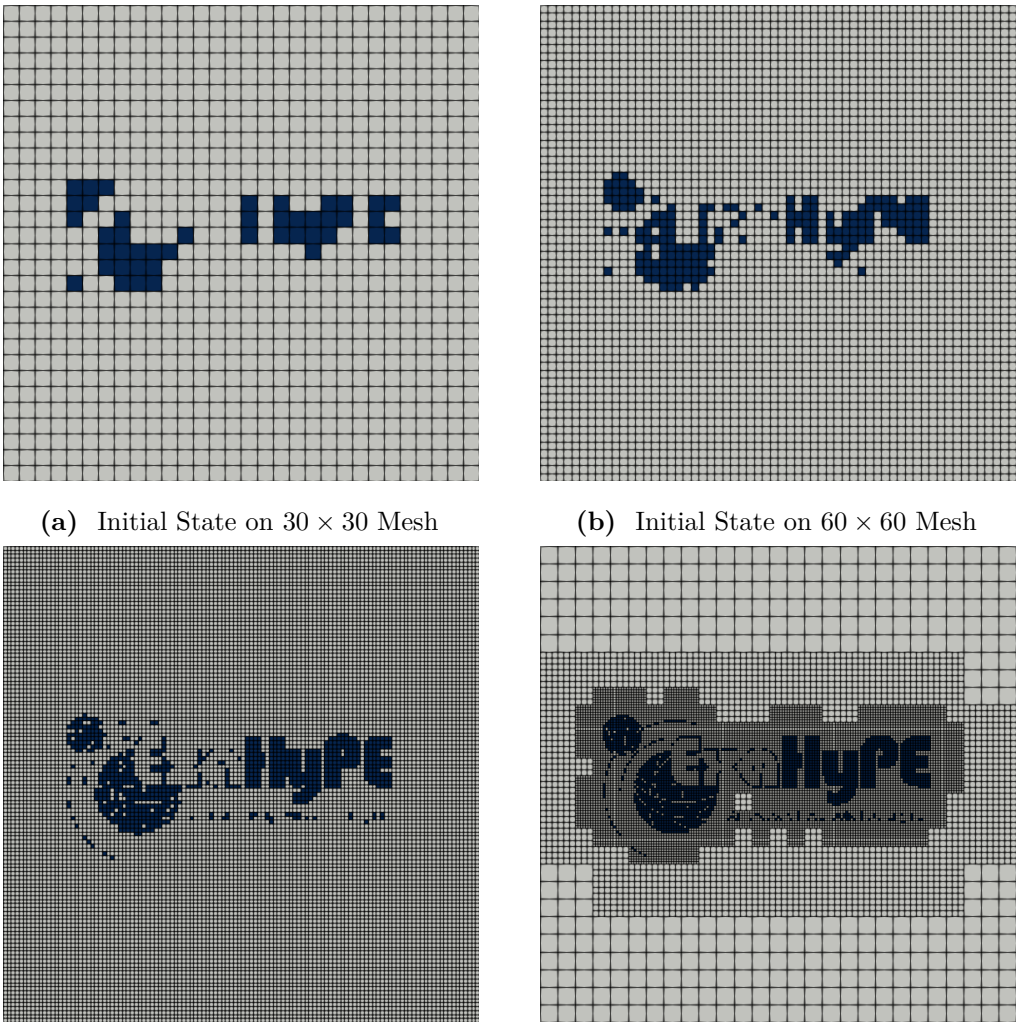
Given a computational domain Ω, a simulation end time T_{final}, and suitable initial and boundary conditions, ExaHyPE's numerical methods solve *weak* formulations of (2.1), which also admit discontinuities in q. They do not compute the exact solution q but an approximation q_h. ExaHyPE uses two numerical methods and couples them with each other to obtain a highly accurate and robust solver: It employs a FVM method with high spatial resolution in regions where the solution exhibits discontinuities, and the ADER-DG method which builds upon high order polynomials to obtain very high accuracy where the solution is sufficiently differentiable. I describe both algorithms and their coupling in the subsequent sections.

The Computational Mesh

Both methods, FVM and ADER-DG, perform a decomposition of the computational domain into smaller parts (*cells*) yielding a *mesh* $\mathcal{T}$. (Mesh and grid are used interchangeably in this thesis.) Both methods minimise their approximation error with increasing mesh resolution.

The simplest mesh family considered by ExaHyPE are uniformly spaced (*regular*) Cartesian meshes (Fig. 2.1 (a)–(c)). From a comparison of meshes (c) and (d) in Fig. 2.1, it is apparent that it often suffices to increase the resolution locally (*adaptively*). ExaHyPE supports dynamically adaptive Cartesian meshes via the PDE framework Peano [102][103][105].

Whilst the ADER-DG and FVM methods allow for very general cell shapes, ExaHyPE only considers quadratic or cubic cells. Complicated domain boundaries are treated by incorporating additional terms into the governing PDE system [92] or via shape transformations that are



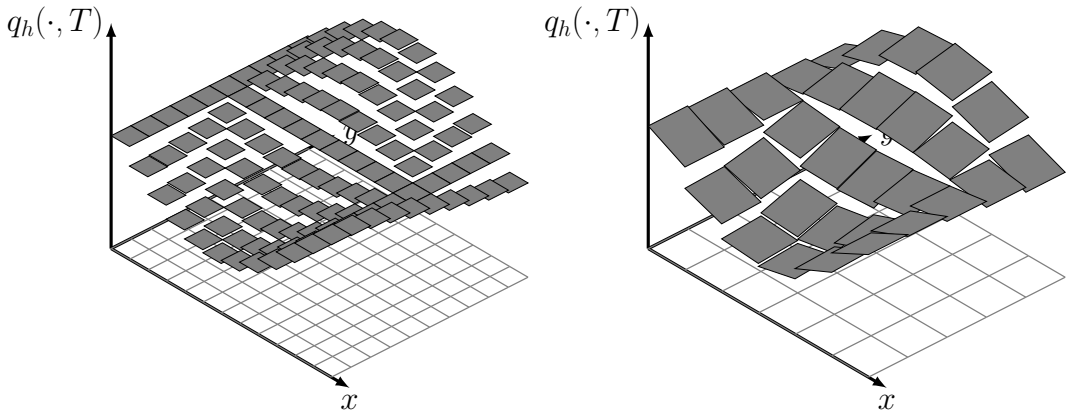
(a) Initial State on 30×30 Mesh

(b) Initial State on 60×60 Mesh

(c) Initial State on 120×120 Mesh

(d) Initial State on Adaptive Mesh

Fig. 2.1: The project logo set as initial state of a 2D simulation. (a)–(c) For increasing mesh resolution, the initial state begins to resemble the project logo. (d) It is sufficient to increase the resolution locally.



(a) Constant per Cell **(b)** Bilinear Function per Cell

Fig. 2.2: Approximation of a smooth scalar-valued solution q in 2D at time T by discontinuous, cell-wise polynomial approximations: (a) each cell holds a constant, (b) each cell holds a bilinear function where the mesh resolution was halved. The respective 2D mesh used is shown on the bottom.

applied to the mesh cells. ExaHyPE's algorithms represent the discrete solution $q_h = q_h(x, t)$ in each cell K at a given time T as a polynomial (Fig. 2.2). Continuity of the discrete solution at the interface between two cells is only weakly enforced. Therefore, the solution exhibits jumps at these interfaces.

One-Step Schemes vs Single-Touch Algorithms

Not only is the spatial resolution of the initial state important for scientific simulations but also its evolution. The ADER-DG and FVM methods that ExaHyPE implements are explicit time stepping schemes, i.e. the discrete solution $q_h(\cdot, T + \Delta T)$ at the next time step depends solely on the discrete solution at the current time step, $q_h(\cdot, T)$. Moreover, ExaHyPE's ADER-DG and FVM schemes are one-step schemes [59]. Such schemes require only a single exchange of information between neighbouring cells. Therefore, their program flow is similar to that of explicit Euler:

Algorithm 2.1 (Explicit Euler)**.** *The current simulation state at time T is evolved to a new state at time $T + \Delta T$ according to an update operator $L_h(T, \Delta T)$. This operator models the physics of the simulated system and is applied to the current simulation state $q_h(\cdot, T)$.*

```
1 q_h(·, 0) ← approximate q(·, 0)                          # set initial simulation state
2 T ← 0                                                    # set initial simulation time
3 while T < T_final do                                     # run time steps until final time is reached
4   q_h(·, T + ΔT) ← q_h(·, T) + ΔT L_h(T) q_h(·, T)       # compute next simulation state
5   T ← T + ΔT                                             # increment (current) simulation time
6 end while
```

The *time step size* ΔT in Algorithm 2.1 can typically not be chosen freely. It depends on the physics via the update operator $L_h(T)$ and the resolution of the used computational mesh. If ΔT is chosen too large, the time stepping becomes unstable and the simulation states lose their physical meaning [32]. Typically, the simulation crashes with numerical errors shortly after. The update operator $L_h(T)$ evolves the simulation state within each cell and couples the individual cells of the computational mesh. If it acts linearly on $q_h(\cdot, T)$, then $L_h(T)$ can be represented as a matrix. ExaHyPE's algorithms never assemble and store such a matrix representation of $L_h(T)$. They directly evaluate the action $L_h(T)\, q_h(\cdot, T)$ instead.

However, applying the action of $L_h(T)$ onto the previous simulation state $q_h(\cdot, T)$ requires that straightforward realisations of ExaHyPE's algorithms traverse the computational mesh multiple times. The solution is read multiple times per time step. A *single-touch* algorithm describes a one-step scheme that accesses the solution data only once per time step. While one-step schemes have the minimum number of neighbour communication steps per time step, single-touch algorithms additionally minimise communication between processing units and main memory. They are an ideal fit to exascale computers, whose extreme compute power can only be exploited if communication is minimal [73]

In this thesis, I rephrase ExaHyPE's numerical methods as single-touch algorithms or if that is not possible, as weak single-touch algorithms that access solution data more than once only in a fraction of time steps. In order to highlight my modifications to the original numerical methods, I introduce typical straightforward algorithmic realisations of these methods in the remainder of this chapter.

2.2 Finite Volumes Methods

In this section, I briefly outline the unsplit FVM methods upon which ExaHyPE's FVM implementations are based. To this end, I consider a formulation of (2.1) where only the flux tensor F is non-trivial. The FVM method is derived from a more general integral formulation [78][95] that can be recovered from the strong formulation (2.1) by integrating over a cell (or volume) V:

$$\int_V \left(\frac{\partial q}{\partial t} + \nabla \cdot F(q) \right) \mathrm{d}x = 0.$$

Applying Green's theorem to the term involving F results in the more general weak formulation:

$$\frac{\mathrm{d}\bar{q}_V}{\mathrm{d}t} = -\frac{1}{|V|} \oint_{\partial V} F(q)\, n \,\mathrm{d}s(x),$$

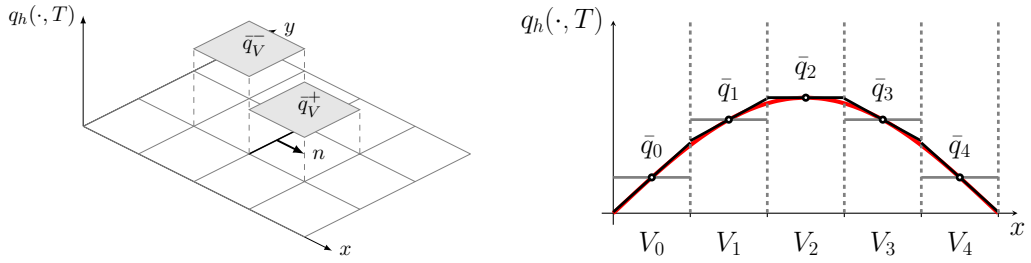
where $\bar{q}_V = \bar{q}_V(t) = \int_V q \,\mathrm{d}x$ is the volume average of q in V and $n = (n_1, n_2, \ldots)^{\mathrm{T}}$ is the normal vector to the volume hull ∂V. While the divergence operator in the strong formulation only admits solutions that are pointwise differentiable (in space), i.e. "smooth" solutions, this more general formulation admits additional solutions such as propagating discontinuities.

A spatial semi-discretisation is then derived by approximating the normal flux $F(q)\, n$ at every interface. Evaluating the normal flux at the interface requires to evaluate the FVM solution at the interface. Let q^+ denote the interface state of the cell in direction of the interface's normal vector n and q^- the one of the cell in opposite direction (Fig. 2.3). The interface states $q^\pm$ can be chosen directly as the volume averages of the neighbours, i.e $q^\pm = \bar{q}_V^\pm$. However, if the solution q is sufficiently differentiable, better estimates of $q^\pm$ can be obtained by reconstructing a higher order polynomial interpolant from the volume averages of a volume V and its neighbours (Fig. 2.3 (b)).

With either approach, the limit values of the normal flux from both sides of an interface, $F(q^+)\, n$ and $F(q^-)\, n$, typically differ. Therefore, in the FVM method, a unique numerical normal flux $G(q^+, q^-)\, n$ is defined on each interface:

$$\frac{\mathrm{d}\bar{q}_V}{\mathrm{d}t} = -\frac{1}{|V|} \oint_{\partial V} G(q^+, q^-)\, n \,\mathrm{d}s(x). \tag{2.2}$$

The numerical normal flux $G(q^+, q^-)\, n$ is typically chosen as the exact or approximate solution to a *Riemann* problem; see [95] for a comprehensive overview. ExaHyPE's applications often



(a) Interface Discontinuity

(b) Linear Slope Reconstruction

Fig. 2.3: (a) At the interface (bold) of two 2D mesh cells, the scalar-valued discrete solution q_h at time T is discontinuous. The example shows the volume averages of the neighbour in direction of the interface's normal vector n ("+") and of the neighbour in opposite direction ("-"). The volume averages of other cells are not shown. (b) A similar scenario is shown in 1D. The Godunov FV method directly uses the volume averages (grey) while the MUSCL-Hancock method reconstructs linear functions (black) for extrapolating a cell's volume averages to the cell's faces (dashed). Therefore, the MUSCL-Hancock method's interface states agree better with the reference solution (red).

use a *Rusanov* flux,

$$G(q^+, q^-) = \frac{1}{2}\left(F(q^+) + F(q^-)\right) - \frac{\lambda_{\max}}{2}\left(q^+ - q^-\right),$$

where $\lambda_{\max}$ denotes the maximum eigenvalue chosen from both the Jacobian of $F(q^+)$ and the Jacobian of $F(q^-)$. They either discretise the time derivative with a forward Euler rule yielding the Godunov method [61] or according to a modified version of the MUSCL (Monotonic Upwind scheme for Conservation Laws)-Hancock procedure [99][95]. Per volume V, the Godunov method requires performing an `update` of the form:

$$\bar{q}_V(T + \Delta T) = \bar{q}_V(T) - \frac{\Delta T}{|V|} \oint_{\partial V} G(q^+, q^-)\, n \, \mathrm{d}s(x). \tag{2.3}$$

All volume averages $\bar{q}_V(\tau)$ together form the volume-wise constant solution representation of $q_h(\cdot, \tau)$ at time $\tau \in \{T, T + \Delta T\}$ on the computational domain. Putting equations (2.3) for all volumes V of a mesh in a single equation system, results in an algorithm alike Algorithm 2.1.

The FVM method's CFL (Courant–Friedrichs–Lewy) condition dictates which time step size choice results in a stable algorithm; e.g. [78][95]:

$$\Delta T < \frac{1}{d} \frac{\Delta x_V}{\lambda_{\max,V}}, \tag{2.4}$$

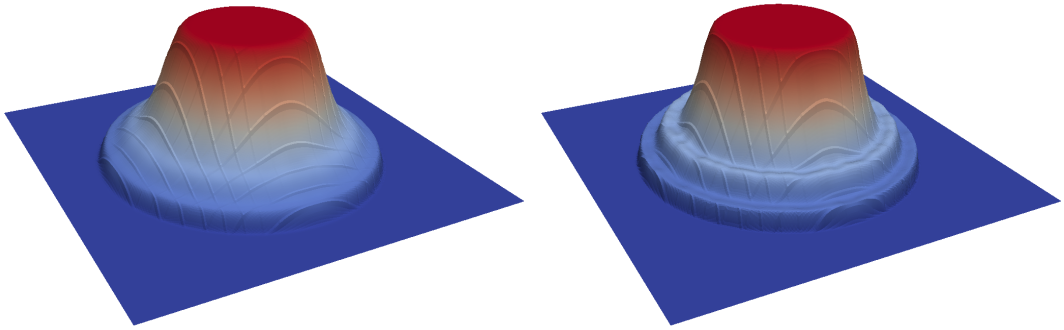
where $\lambda_{\max,V}$ denotes the maximum absolute eigenvalue of the Jacobian of $F(q)$ over the volume V and $\Delta_{x,V}$ the volume's edge length. For PDE systems where F depends nonlinearly on q, the stability region for ΔT might change after every step of the algorithm. Typically, it also changes if the mesh is adapted.

Higher-Order Methods, Patches, and Ghost Layers

The notion of "first-order" and "second-order" accuracy of FVM methods refers to the leading term of the local truncation error of the respective FVM method applied to linear advection problems with sufficiently regular solution [29]. The first-order Godunov method is very dissipative while the second-order MUSCL-Hancock scheme employs significantly less numerical dissipation in regions where the solution is sufficiently differentiable (Fig. 2.4). In each cell, the scheme reconstructs a linear polynomial from the volume averages of neighbouring cells in order to obtain more accurate interface states. A second-order time integration procedure then renders the whole scheme second-order accurate with respect to the mesh spacing [95].

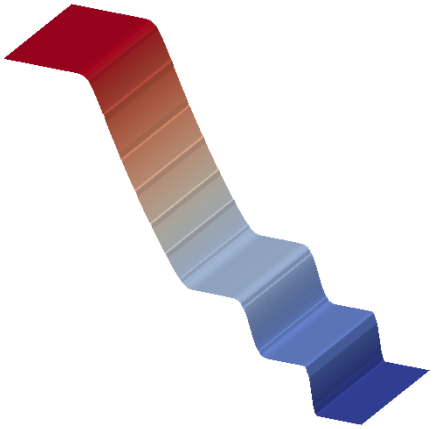
Low-order FVM methods require a high mesh resolution due to their low accuracy while high order FVM methods require a number of neighbours to reconstruct high-order polynomials. Therefore, it is a common pattern to not associate single volumes V with mesh cells but regular subgrids (or, *patches*) that group multiple volumes together. ExaHyPE implements such patch-based FVM schemes, too. Using patches further allows to group the corresponding volume averages into regular blocks. These can then be fed more efficiently to the hardware processing units as these read memory in fixed size chunks and offer SIMD (single instruction multiple data) instructions to perform the same operation on multiple floating point numbers at once; see e.g. [106][104]. In addition, the regular blocks allow to decompose FVM substeps such that they can be processed in parallel by multiple cores of a CPU or GPU (graphics processing unit).

In each regular block of volume averages that corresponds to a subgrid, ExaHyPE's FVM implementations keep storage for additional volume averages. These *ghost cells* are reserved for storing volume averages from the boundary layers of neighbouring patches (Fig. 2.5). In its original form, the MUSCL-Hancock scheme requires two communication steps as it performs a substep where it evolves the interface data by half a time step [95]. The updated interface data has to be exchanged between neighbouring cells before the time step can be finished

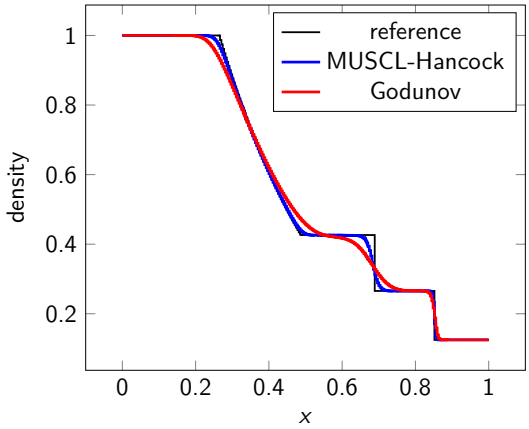


(a) Explosion — Godunov Density

(b) Explosion — MUSCL-Hancock Density



(c) Sod – MUSCL-Hancock Density



(d) Sod — Density Profiles

Fig. 2.4: (a) – (b): The density variable of a gas is computed on a 2D mesh with two different FVM methods: (a) the first-order Godunov method, (b) the second-order MUSCL-Hancock method. The circular gas explosion simulated in the test is modelled by the compressible Euler equations and the density variable is compared at time $t = 0.1$. The density variable is warped and linear interpolation is used to connect the individual volumes. Both methods use a mesh (not shown) with 270×270 volumes (a subgrid has 10×10 volumes). (c) – (d): The Sod shock tube test allows us to compare solutions obtained with the Godunov and MUSCL-Hancock method against an analytical reference solution. We compare at time $t = 0.2$ and use a mesh with 270×90 volumes. In both tests, the numerical diffusion added by the Godunov method becomes apparent.

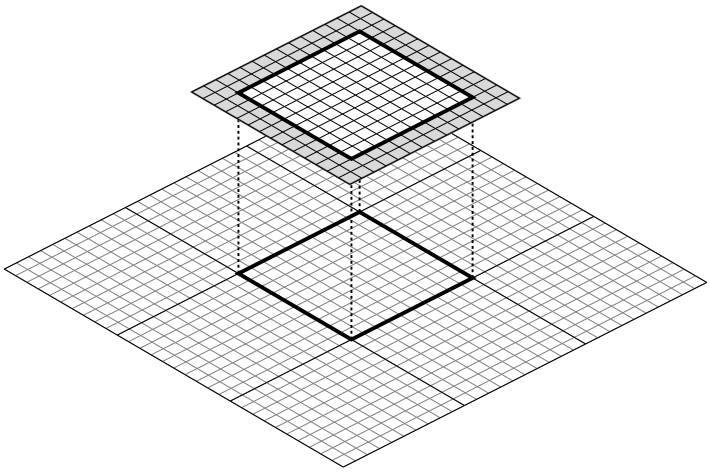


Fig. 2.5: A 2D mesh consists of 3×3 subgrids where each is divided into 10×10 volumes. The regular blocks of volume averages corresponding to the subgrids keep storage for two additional layers of volume averages. These ghost layers are used for copying the volume averages of neighbouring subgrids. They can be interpreted as a number of additional layers of virtual cells (gray fill) around each subgrid.

in the second substep [95]. The method uses ghost layers of width one, which are filled in every substep. Therefore, neighbouring patches need to exchange ghost layers during every substep–that is twice per time step.

ExaHyPE's modified MUSCL-Hancock scheme exchanges two layers of boundary cells at once in a single communication step. This comes at the expense of storing two ghost layers per neighbour and performing certain slope computations redundantly. This modification can be understood as a trade of communication latency versus communication bandwidth. With it, MUSCL-Hancock can be written in the form of Algorithm 2.1, too. The following algorithm shows a straightforward realisation of both of ExaHyPE's FVM methods. They realise a one-step scheme (cf. Algorithm 2.1) with two algorithmic phases: one loop over the interfaces that copies the boundary layers to the ghost layers and a second loop that updates all subgrids; see Algorithm 2.2.

Algorithm 2.2 (Patch-Based Finite Volumes Methods)**.** *Algorithmic phases of* ExaHyPE*'s Godunov and* MUSCL*-Hancock scheme. In an implementation,* $q_h(\cdot, T + \Delta T)$ *and* $q_h(\cdot, T)$ *can be stored in the same vector as* $q_h(\cdot, T)$ *is not required any longer after the update.*

```
1  T ← 0                                                    # set initial simulation time
2  initialiseFV( )
3
4  while T < T_final do                                     # run time steps until final time is reached
5    for face connected patches K_a, K_b ∈ 𝒯 do             # communicate with face neighbours
6      copyBoundaryLayers(q_h(·,T)|_{K_a}, q_h(·,T)|_{K_b})   # copy to neighbour's ghost layers
7    end for
8    for patch K ∈ 𝒯 do                                     # evolve (current) simulation state
9      q_h(·,T+ΔT)|_K ← update(q_h(·,T)|_K, ΔT)             # Godunov or MUSCL−Hancock update
10   end for
11   T ← T + ΔT                                            # increment (current) simulation time
12 end while
13
14 function initialiseFV( )
15   for patch K ∈ 𝒯 do
16     q_h(·,0)|_K ← ∀V ⊂ K: average q(·,0)|_V
17   end for
18 end function
```

2.3 The ADER-DG Method

The ADER-DG method is a generic recipe to construct high-order approximations of smooth solutions. Higher-order ADER-DG variants assume that the solution is sufficiently regular within each mesh cell. If this not the case, e.g. if the solutions contains jumps, these methods generate non-physical oscillations and might even become unstable. Due to this regularity assumption, I start from the strong formulation (2.1) to derive ADER-DG. Again, I consider a simplified variant of (2.1) where only the flux is non-trivial. My discrete ansatz $\hat{q}_h$ approximates the space-time solution q as cell-wise polynomial function times an interval-wise polynomial function that approximates the time evolution, i.e. I prescribe that the solution is smooth within every cell. Spatial and temporal component of the discrete ansatz use the same polynomial order p.

I then multiply (2.1) by a cell-wise polynomial test function v_h drawn from the function space

that the spatial component of the discrete ansatz $\hat{q}_h$ belongs to. Integrating over a mesh cell K and a time interval $(T, T + \Delta T)$ then results in a cell-wise spatial semi-discretisation of (2.1):

$$\int_K \int_T^{T+\Delta T} \left(\frac{\partial}{\partial t} \hat{q}_h + \nabla \cdot F(\hat{q}_h) \right) v_h \, \mathrm{d}x \, \mathrm{d}t = 0.$$

In the next step, I apply the product rule to the second term, which yields

$$\begin{aligned} \int_T^{T+\Delta T} \int_K \frac{\partial}{\partial t} \hat{q}_h \, v_h \, \mathrm{d}x \, \mathrm{d}t - \int_T^{T+\Delta T} \int_K F(\hat{q}_h) : \nabla v_h \, \mathrm{d}x \, \mathrm{d}t \\ + \int_T^{T+\Delta T} \int_K \nabla \cdot (F(\hat{q}_h) \, v_h) \, \mathrm{d}x \, \mathrm{d}t = 0, \end{aligned} \tag{2.5}$$

for any v_h as above and every $K \in \mathcal{T}$. Here, $F(\hat{q}_h) : \nabla v_h$ denotes the inner product of the two tensors. I then integrate the first term of (2.5) by parts in time, where I use that v_h does not depend on time. Furthermore, I apply Green's theorem to the third term. This yields the variational problem:

Find $\hat{q}_h$ such that

$$\begin{aligned} \int_K (\hat{q}_h(\cdot, T + \Delta T) - \hat{q}_h(\cdot, T)) \, v_h \, \mathrm{d}x = \int_T^{T+\Delta T} \int_K F(\hat{q}_h) : \nabla v_h \, \mathrm{d}x \, \mathrm{d}t \\ - \int_T^{T+\Delta T} \int_{\partial K} F(\hat{q}) : (v_h \otimes n) \, \mathrm{d}s(x) \, \mathrm{d}t, \end{aligned} \tag{2.6}$$

for all v_h as above and every $K \in \mathcal{T}$. Here, $v_h \otimes n$ is the outer product of the two vectors that yields a tensor of the same structure as $F(\hat{q})$.

The values of $\hat{q}_h$ within the time interval $(T, T + \Delta T)$ are unknown. In the ADER-DG method, those intermediate values are thus replaced by an estimate q_h^*–the *space-time predictor*. It is chosen from the same function space as the discrete ansatz $\hat{q}_h$. (Procedures to obtain q_h^* are discussed in the next section.) The space-time predictor q_h^* is inserted into F and the traces of its cell-wise restrictions are inserted into a numerical normal flux $G(q_h^{*,-}, q_h^{*,+}) \, n$, which replaces the original boundary flux. The inputs $q_h^{*,-}$ and $q_h^{*,+}$ denote the traces of the q_h^* from the cells in negative and positive direction to the normal of the shared interface, respectively.

The above modifications transform (2.6) into the `corrector` step of the ADER-DG method:

Find $q_h(\cdot, T+\Delta T)$ such that

$$\int_K (q_h(\cdot, T+\Delta T) - q_h(\cdot, T))\, v_h \,\mathrm{d}x = \underbrace{\int_T^{T+\Delta T} \int_K F(q_h^*) : \nabla v_h \,\mathrm{d}x \,\mathrm{d}t}_{\text{volume integral}} - \underbrace{\int_T^{T+\Delta T} \int_{\partial K} G(q_h^{*,-}, q_h^{*,+}) : (v_h \otimes n) \,\mathrm{d}s(x) \,\mathrm{d}t}_{\text{surface integral}} \tag{2.7}$$

for any v_h as above and every $K \in \mathcal{T}$. The first term on the right-hand side is called the volume integral and the second term is called the surface integral. The surface integral can be decomposed into single face integrals. Only the face integrals couple a cell with its neighbours in the ADER-DG method.

In (2.7), the discrete ansatz $\hat{q}_h(\cdot, \tau)$ is replaced by the discrete solution $q_h(\cdot, \tau)$, $\tau \in \{T, T+\Delta T\}$, respectively. Equation (2.7) describes ADER-DG's fully-discrete, explicit one-step update procedure advancing the solution from state $q_h(\cdot, T)$ to Note that the solution snapshots $q_h(\cdot, \tau)$, $\tau \in \{T, T+\Delta T\}$, computed with the ADER-DG method only dependent on space but not on time, i.e. ADER-DG does not compute a space-time solution. As in the FVM methods, the numerical flux $G(q_h^{*,-}, q_h^{*,+})n$ is typically obtained using an exact or approximate Riemann solver. ExaHyPE's ADER-DG uses a Rusanov solver by default. For a stable algorithm, the time step size ΔT must adhere to a CFL condition [92]:

$$\Delta T < \frac{1}{d} \frac{\mathrm{C}_{\mathrm{ADER}}(p)}{2\,p+1} \frac{\Delta x_K}{|\lambda_{\max,K}|}, \tag{2.8}$$

where d is the space dimension, p is the approximation order, Δx_K is the diameter of the mesh cell, and $\lambda_{\max,K}$ is the maximum absolute eigenvalue of the (linearised) flux tensor. Note that compared to a p-th order RK-DG (Runge-Kutta Discontinuous Galerkin) method, the ADER-DG method's stability region is smaller, which is expressed by the p-dependent factor $\mathrm{C}_{\mathrm{ADER}}(p) < 1$ in (2.8); see Chapter 5 and [48] for more details.

The Prediction Step

The original arbitrary derivative finite volume and discontinuous Galerkin algorithms introduce the space-time predictor q_h^* as a Taylor expansion in space and time around the space-time point (x_K, t) in every cell $K \in \mathcal{T}$, where x_K denotes the cell center. They then apply a Cauchy-Kowalewsky (CK) procedure to compute time derivatives in terms of the spatial derivatives. The CK (Cauchy-Kowalewsky) approach is regarded as the most efficient approach for linear

hyperbolic systems; see [59] for a detailed discussion of this approach and a comparison to other ADER approaches utilised in the literature. It can be implemented in an explicit and generic way for linear problems; however, it becomes rather cumbersome for nonlinear problems [54].

In the following, I discuss the *local space-time discontinuous Galerkin predictor* [50]. While the CK procedure is regarded as the fastest predictor implementation for linear problems [59], the local space-time DG predictor a very general and robust method that is applicable to both linear and nonlinear PDE systems. ExaHyPE provides optimised implementations of both variants.

The derivation of the variational problem for obtaining the local space-time discontinuous Galerkin predictor starts again from equation (2.1). Again, I use a discrete space-time ansatz q_h^* for the solution. However this time, I also use space-time test functions ϕ_h. Integration over the space-time slab $K \times (T, T + \Delta T)$ yields the following problem:

Find q_h^* such that

$$\int_K \int_T^{T+\Delta T} \frac{\partial}{\partial t} q_h^* \, \phi_h \, \mathrm{d}x \, \mathrm{d}t + \int_K \int_T^{T+\Delta T} \nabla \cdot F(q_h^*) \, \phi_h \, \mathrm{d}x \, \mathrm{d}t = 0,$$

for any ϕ_h as above and every $K \in \mathcal{T}$. Integrating the first term in time by parts, rephrases the problem as:

Find q_h^* such that

$$\begin{aligned} \int_K q_h^*(\cdot, T + \Delta T) \, \phi_h(\cdot, T + \Delta T) \, \mathrm{d}x - \int_K \int_T^{T+\Delta T} q_h^* \frac{\partial}{\partial t} \phi_h \, \mathrm{d}x \, \mathrm{d}t \\ = \int_K q_h^*(\cdot, T) \, \phi_h(\cdot, T) \mathrm{d}x - \int_K \int_T^{T+\Delta T} \nabla \cdot F(q_h^*) \, \phi_h \, \mathrm{d}x \, \mathrm{d}t, \end{aligned} \tag{2.9}$$

for all ϕ_h as above and every $K \in \mathcal{T}$.

Since no Green's theorem was applied, (2.9) can be decomposed into uncoupled implicit cell-wise subproblems. Using Picard iterations to solve the fixed-point problems quickly converges to a solution for many problems [54]. A straightforward choice of the initial guess is the previous solution [54]. More sophisticated choices are discussed in [51]. The latter paper also details the treatment of non-conservative terms of (2.1) with the ADER-DG method.

Convergence

To the best of my knowledge, no a priori error estimates for the ADER-DG method for particular hyperbolic conservation laws in multiple dimensions have been published to this date. The high-order approximation ability of the method is typically verified via numerical experiments. For hyperbolic PDE systems that admit a sufficiently regular solution, it has been demonstrated that the discretisation errors behave as:

$$\|q(\cdot,T) - q_h(\cdot,T)\|_{L^k(\Omega)} \leq C\,\Delta x_K^{p+1}, \tag{2.10}$$

with a constant C independent of the cell size Δx_K, the time step size ΔT, and the approximation order p. The symbol $\|\cdot\|_{L^k(\Omega)}$, $k \in \{1,2,\infty\}$, denotes the respective Lebesgue norm. See [59] for a verification of the convergence rate for the linear advection equation, and for the nonlinear, compressible Euler equations; see [52] for a verification for elastic waves. See [51] for a verification for the compressible Euler equations and the equations of ideal general relativistic magnetohydrodynamics (GRMHD).

Algorithm

The ADER-DG method can be written the following way, which decomposes each time step of Algorithm 2.1 into three algorithmic phases:

Algorithm 2.3 (The ADER-DG Method)**.** *A straightforward ADER-DG implementation has three algorithmic phases per time step. In the first phase, we loop over all mesh cells and perform the predictor step (blue). In the second phase, we loop over all mesh faces and perform the Riemann solve and the subsequent face integral (green). In the third and last phase, we loop over all cells again and perform the corrector step (red). In an implementation,* $q_h(\cdot,T+\Delta T)$ *and* $q_h(\cdot,T)$ *can be stored in the same vector as* $q_h(\cdot,T)$ *is not required any longer after the predictor computation. The symbols* n_a *and* n_b *denote the outward-directed normal vectors to the respective cell hull. They are used to determine which cell is located in direction of the positively signed normal vector* n *of the face when performing the Riemann solve.*

```
1 T ← 0                                              # set initial simulation time
2 INITIALISEADERDG( )
3
4 while T < T_final do
```

```
5   for cell K ∈ 𝒯 do                                                    # predictor step
6     ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K, ΔT)               # with CK or local space−time DG
7     q_h(·,T+ΔT)|_K ← q_h(·,T)|_K                                       # copy (or reuse) solution vector
8   end for
9   for face−connected cells K_a, K_b ∈ 𝒯 do                             # Riemann solves
10    ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
11    ( q*_h|_{K_b}|_{∂K_b∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_b∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
12    G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
13                                q*_h|_{K_b}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
14    q_h(·,T+ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
15    q_h(·,T+ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
16  end for
17  for cell K ∈ 𝒯 do                                                    # finish corrector step
18    q_h(·,T+ΔT)|_K += volumeIntegral(F(q*_h)|_K, ΔT)                   # complete the corrector step
19  end for
20  T ← T + ΔT
21 end while
22
23 function initialiseADERDG( )
24   for cell K ∈ 𝒯 do
25     q_h(·,0)|_K ← represent q(·,0)|_K as polynomial
26   end for
27 end function
```

The ADER-DG algorithm requires one step, the cell-wise predictor, prior to the neighbour communication. Aside from this, Algorithm 2.3 and Algorithm 2.2 have an identical program flow, the loop over the faces is followed by a loop over the cells.

2.4 A Posteriori Subcell Limiting for ADER-DG

Solving hyperbolic problems subject to discontinuous initial conditions with a plain higher order method causes numerical solutions to yield non-physical oscillations around the discontinuities. The polynomial solution representation used within the cells is only able to model functions that are sufficiently differentiable (Fig. 2.6 (a)). When solving nonlinear hyperbolic problems, discontinuities can develop over time even from smooth initial conditions ("shock formation"). In [64], the authors summarise three problematic effects of discontinuities on the solution computed with a higher order method:

- The loss of pointwise convergence at the point of discontinuity

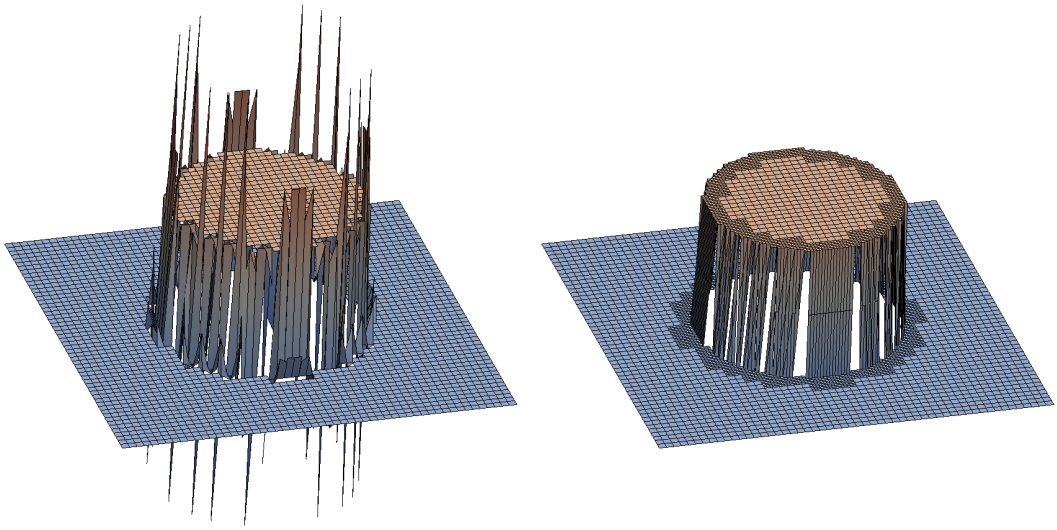
- The reduction to first order pointwise accuracy away from the point of discontinuity.
- The introduction of artificial and persistent oscillations around the point of discontinuity.

Whilst the first two effects can be tackled by AMR around the discontinuities, the last effect might lead to non-physical solution. For example, physically positive quantities such as the mass density might attain negative values. The positivity of these quantities need to be preserved for physically meaningful results of a computation.

According to the Godunov theorem [61], only first-order accurate numerical methods are capable of evolving discontinuities without introducing non-physical oscillations. While classic Godunov FVM is robust in the presence of shocks, higher-order FVM schemes require additional nonlinear stabilisation techniques. The construction of higher order FVM methods which are robust in the presence of discontinuities has been accomplished with the aid of slope (or, flux) limiters. An overview of slope limiters for the second order MUSCL-Hancock scheme [98] is given in [91].

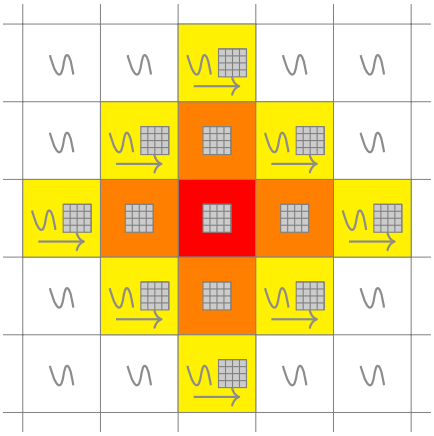
For higher-order variants of the DG and ADER-DG methods, a number of limiting approaches have been presented too; see [54] for an overview. ExaHyPE's limiting ADER-DG solver is based on the a posteriori subcell limiting algorithm, which couples the ADER-DG method with a robust FVM solver [54]. Below, I discuss the original method from [54]. ExaHyPE's variant is discussed in Chapter 7.

At the begin of every time step, the a posteriori subcell limiting ADER-DG method runs a plain ADER-DG time step. After completion of the ADER-DG time step, the ADER-DG solution is checked for non-physical values and oscillations. Cells where such defects are present in the solution are called *troubled*. If at least one cell is marked as troubled, the scheme performs a rollback to the previous valid solution in all troubled cells and their direct and second-degree neighbours. The previous valid solution might be either a previous valid ADER-DG solution or stem itself from an FVM recomputation. As some cells might require an FVM recomputation over the span of multiple time steps, an FVM patch might be available from the previous step. In all three cell types that are involved in the rollback, the scheme then projects the pre-update DG polynomial onto an FVM patch or uses the already available FVM patch from the last time step. Finally, the scheme recomputes the solution in troubled cells and their direct neighbours using a robust FVM method (Fig. 2.6 (c) – (d)).

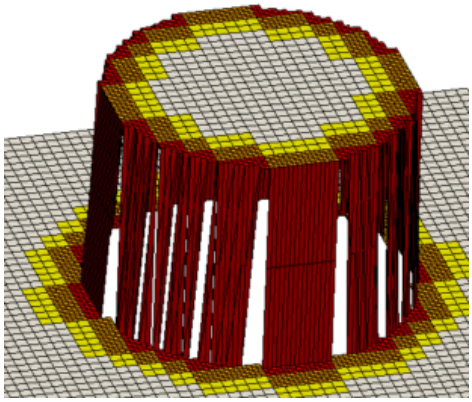


(a) Unlimited High-Order Method

(b) Limited High-Order Method



(c) Limiter Status Stencil ($L_{\max} = 3$)



(d) Limiter Status Along a Discontinuity

Fig. 2.6: (a) The unlimited ADER-DG method generates non-physical oscillations along discontinuities and strong gradients. (b) The oscillations can be cured by coupling the ADER-DG method with a robust FVM method. (c) The limiter status stencil of size $L_{\max} = 3$. White cells are considered as well-behaved and keep the DG (discontinuous Galerkin) solution. Red cells are considered as troubled and are recomputed with FVM during the recomputation step. Orange cells are recomputed with FVM, too. Yellow cells are considered well-behaved too; however, they need to project the pre-update DG solution onto a FVM patch in order to provide boundary conditions for adjacent orange cells. (d) The limiter status distribution along a discontinuity where the DG polynomials show non-physical oscillations.

The second-degree neighbours are simply rolled forward in time to their original ADER-DG solution. During the recomputation, their purpose is to provide FVM boundary data to the direct neighbours of troubled cells (Fig. 2.6 (c)).

Detection Criteria for Non-Physical Oscillations

An a posteriori subcell limiting algorithm considers two criteria to detect troubled cells after the ADER-DG time step: an application-specific PAD (physical admissibility detection) and a relaxed DMP (discrete maximum principle). The PAD is used to ensure that the state variables do not assume non-physical values, e.g. the mass density must remain positive. The PAD depends on the modelled physics or other application-specific criteria. It is provided by the user. The DMP tries to detect oscillations caused by Runge's phenomenon. It is independent of the application but the user must specify two tuning parameters. The DMP compares the extreme values of post-update DG polynomial in cell K with the minimum and maximum values of the pre-update numerical solution in a neighbourhood around cell K:

$$\min_{\substack{x\in K' \\ K'\in\mathcal{V}(K)}} q_{h,i}(x,T) - \varepsilon \leq q_{h,i}(x,T+\Delta T)|_K \leq \max_{\substack{x\in K' \\ K'\in\mathcal{V}(K)}} q_{h,i}(x,T) + \varepsilon, \tag{2.11}$$

where the set $\mathcal{V}(K)$ contains the face-connected neighbours of cell K. The DMP is evaluated per state variable i of the ADER-DG solution and uses a relaxation parameter ε which must be tuned with respect to the ADER-DG approximation order and the considered PDE. The DMP states that a significant change of the minimum and maximum values of the solution within a cell must come from its closest neighbours. Otherwise, non-physical oscillations are assumed to be the cause of the change.

Algorithmic Building Blocks

Next, I translate the textual description of the a posteriori limiting ADER-DG method into an algorithm as I have done with the other numerical methods. To this end, I first introduce the method's individual algorithmic building blocks. I start with the substeps that are required to perform the recomputation step. Here, the layers of first and second degree neighbours (Fig. 2.6 (c)) must be identified around the troubled cells (Fig. 2.6 (d)) before the FVM recomputation step can be run. This is done by diffusing an integer flag that is called *limiter status*. Two loops over the mesh are required to identify direct and second-degree neighbours of troubled cells. These loops only perform communication between neighbouring cells:

Algorithm 2.4 (Limiter Status Diffusion)**.** *In every limiter status diffusion iteration, the limiter status of a cell is computed as the maximum of its own limiter status and those of its face-connected neighbours minus one. The minimum limiter status is 0 and the maximum limiter status is* $L_{\max} = 3$*, which indicates a troubled cell.*

```
1  function DIFFUSELIMITERSTATUS(L_max = 3)
2    for i = 1, 2, ..., L_max − 1 do        # for each layer to construct around a troubled cell
3      for face-connected cells K_a, K_b ∈ T do
4        limiterStatus_{K_a} ← max(limiterStatus_{K_a}, limiterStatus_{K_b} − 1)
5        limiterStatus_{K_b} ← max(limiterStatus_{K_b}, limiterStatus_{K_a} − 1)
6      end for
7    end for
8  end function
```

After the two layers are build, three additional loops are required to recompute troubled cells and their direct neighbours with a robust FVM method. The algorithm roughly follows the structure of the FVM method but some limiter status-based filtering is necessary in the neighbour exchange and update step. In addition, the recomputed solution must be represented again as ADER-DG polynomial in troubled cells and their direct neighbours:

Algorithm 2.5 (Solution Recomputation with a FV Method)**.** *The troubled cells with limiter status* $L_{\max} = 3$ *and their direct neighbours with limiter status* $L_{\max} - 1$ *are recomputed with an* FVM *method. Before, troubled cells and their first- and second-degree neighbours, i.e. all cells with at least limiter status* $L_{\max} - 2$*, have to exchange* FVM *boundary data from time level* $t = T$*. Modifications to standalone* FVM *are highlighted in green.*

```
1  function RECOMPUTEWITHFV()
2    for cell/patch K ∈ T do # allocate/deallocate previous FV solution
3      if limiterStatus_K > 0 and oldLimiterStatus_K = 0 then
4        if T = 0 then
5          q_{h,FV}(·, 0)|_K ← ∀V ⊂ K: average q(·, 0)|_V
6        else
7          q_{h,FV}(·, T)|_K ← average polynomial q_h(·, T)|_K in V, ∀V ∈ K:
8        end if
9      else if limiterStatus_K = 0 ∧ oldLimiterStatus_K > 0 then
10       deallocate q_{h,FV}(·, T)|_K
11     end if
12   end for
```

```
13
14   for face−connected cells/patches K_a, K_b ∈ T do # exchange FV boundary layers
15     if limiterStatus_{K_a} ≥ L_max − 2 and
16        limiterStatus_{K_b} ≥ L_max − 2 then
17       copyBoundaryLayers (q_{h,FV}(·,T)|_{K_a}, q_{h,FV}(·,T)|_{K_b})
18     end if
19   end for
20
21   for cell /patch K ∈ T do # recompute troubled cells and direct neighbours
22     if limiterStatus_K ≥ L_max − 1 then
23       q_{h,FV}(·,T+ΔT)|_K ← update(q_{h,FV}(·,T)|_K, ΔT)
24       q_h(·,T+ΔT)|_K ← represent q_{h,FV}(·,T)|_K as polynomial
25     end if
26   end for
27 end function
```

The overall algorithm is then constructed from the individual building blocks as follows:

Algorithm 2.6 (The A-Posteriori Subcell Limiting ADER-DG Method)**.** *The algorithmic phases of a straightforward ADER-DG implementation with a posteriori subcell limiting as proposed in [54]. Modifications to the unlimited method are highlighted in green.*

```
1  T ← 0                                                  # set initial simulation time
2  InitialiseADERDG( )
3
4  while T < T_final do                  # run ADER−DG time steps with a few modifications
5    recomputeWithFV ← false
6    for cell K ∈ T do
7      ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K, ΔT)
8      q_h(·,T+ΔT)|_K ← q_h(·,T)|_K                       # must copy solution vector!
9    end for
10   for face−connected cells K_a, K_b ∈ T do
11     ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
12     ( q*_h|_{K_b}|_{∂K_b∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_b∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
13     G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
14                                 q*_h|_{K_b}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
15     q_h(·,T+ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h) (n · n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
16     q_h(·,T+ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h) (n · n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
17     exchange min. and max. of q_h(·,0)|_{K_a} and q_h(·,0)|_{K_b}
18   end for
19   for cell K ∈ T do
```

```
20      q_h(·, T + ΔT)|_K += volumeIntegral (F(q*_h)|_K, ΔT)
21      oldLimiterStatus_K ← limiterStatus_K                    # save current limiter status
22      limiterStatus_K ← 0                                     # compute new limiter status
23      if ¬DMP(q_h(·, T + ΔT)|_K, prev. min and max in neighbourhood) or
24         ¬PAD(q_h(·, T + ΔT)|_K) then                         # evaluate the limiter criteria
25        limiterStatus_K ← 3
26        recomputeWithFV ← true
27      end if
28    end for
29    if recomputeWithFV then
30      diffuseLimiterStatus( )
31      recomputeWithFV( )
32    end if
33    T ← T + ΔT
34  end while
```

Note that to facilitate rollbacks, the limiting ADER-DG solver must store the previous ADER-DG solution $q_h(\cdot, T)$, too. Overwriting the same solution vector is not possible anymore. Furthermore, all cells that compute with the FVM method, i.e. troubled cells and their direct neighbours, need to store one FVM patch for $q_h(\cdot, T)$ and $q_h(\cdot, T + \Delta T)$ each. Project-to-FV cells need to store an FVM patch for $q_h(\cdot, T)$ during the recomputation phase.

2.5 Discussion

In this chapter, I express typical straightforward realisations of the numerical methods used in ExaHyPE as pseudocode algorithms in order to unveil their program flow and communication patterns. I compare the algorithms against two classes of time stepping algorithms: one-step schemes and single-touch schemes. One-step schemes only require a single neighbour-communication step per time step; however, they may consist of multiple algorithmic phases per time step, i.e. they may load solution data multiple times. Hence, they are optimal with respect to avoiding network latency but not optimal with respect to memory access. In contrast, single-touch schemes are one-step schemes that load solution data only once per time step. My brief analysis reveals that the ADER-DG and FVM method's are rather simple and can both be written as one-step scheme; however, the a posteriori limiting ADER-DG method requires three additional communication steps if discontinuities are present in the solution. In total, straightforward realisations of the limiting ADER-DG

scheme require 8 loops over the mesh, of which 6 access the solution or parts of it.

3

Engine Design and Toolset

ExaHyPE's feature list has grown continuously during the duration of my PhD and user interfaces had often to be redesigned due to new requirements. To ensure code stability while keeping developer productivity at a high level, ExaHyPE separates user aspects from the core aspects of the engine via glue code. Users do not to write this glue code themselves. Instead, ExaHyPE provides a source-to-source compiler that generates it based on the profile of the user's application and numerical method of choice. This has allowed ExaHyPE's developers to keep up with the growing list of feature requests while keeping the application focus of the application scientists on the physics.

This chapter gives a brief overview of the software architecture of ExaHyPE's two main components, ExaHyPE core and ExaHyPE toolkit. The ExaHyPE core provides procedures for mesh adaptation, time stepping and plotting. It further implements the solver base classes that ExaHyPE users base their application upon. Fig. 3.1 shows the software stack for an ExaHyPE user solver. ExaHyPE's core is written in object-oriented C/C++. The ExaHyPE toolkit generates the implementation files that users start writing their application with. The toolkit wraps a glue-code generator and a code generator for optimised compute kernels. The glue-code generator is used to extend the solver base classes of the ExaHyPE core with project-specific compile-time information such as the number of variables or the

approximation order. Furthermore, it creates a template for the user solver that users tailor to their particular application. The toolkit and its components are written in `Python 3`. They are steered via the ExaHyPE specification file, where users configure compile-time and runtime parameters. The generation of optimised compute kernels is controlled from the specification file, too.

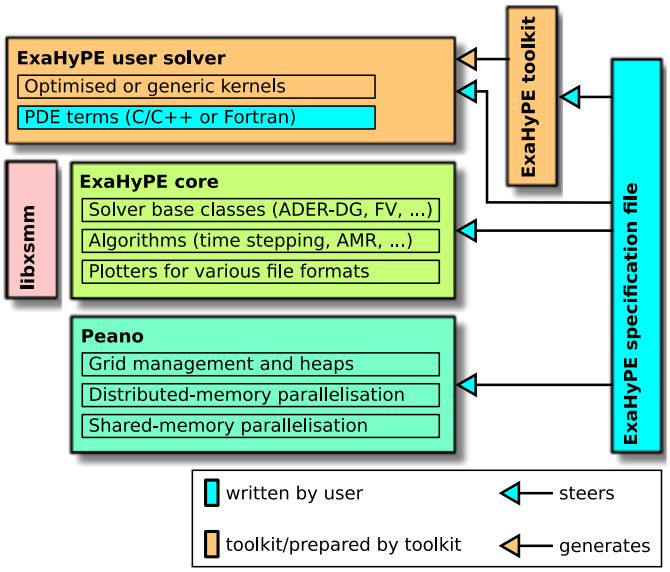


Fig. 3.1: ExaHyPE applications rely on the ExaHyPE core and the ExaHyPE code generators (ExaHyPE toolkit and optimised kernel generator). They are steered by the specification file. The ExaHyPE core is based upon Peano. The generated optimised kernels rely on `libxsmm`.

The ExaHyPE core provides three solver abstract classes that users base their implementation upon: `ADERDGSolver`, `FiniteVolumesSolver`, and `LimitingADERDGSolver`. Most of ExaHyPE's features and optimisations are available to all three solvers.

The UML (Unified Modelling Language) diagram 3.2 shows the inheritance diagram for user solver's based on ExaHyPE's ADER-DG and FVM methods. The base classes implement all high-level functionality including mesh refinement and communication routines. Their high-level routines call cell- and face-wise solver kernels, which are in turn implemented by the base class of the user solver. The toolkit generates application-specific compile-time information such as the polynomial order and number of variables into this class. Data field sizes and kernel-loop bounds can thus be derived from this information at compile time. Every

toolkit run overwrites the user solver base class.

A user solver's base class does not have knowledge of the application's fluxes, source terms, and refinement criteria. Implementing these application-specific details is left to the users. They implement the user solver class. Users can rely here on the flexibility of general-purpose programming languages such as C/C++ or Fortran.

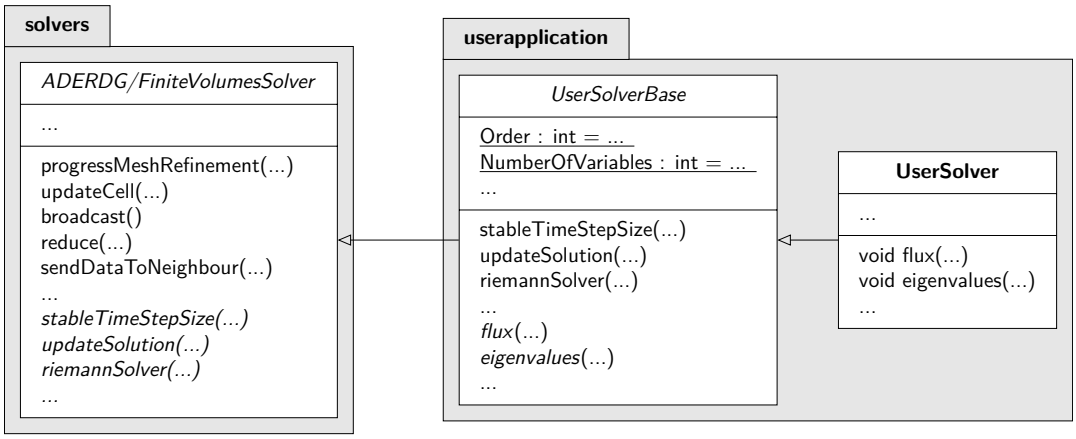


Fig. 3.2: ExaHyPE solver base class, user solver base class, and user solver. The user implements only functionality of the latter. The abstract user solver base class is regenerated every time the toolkit is called. The abstract ExaHyPE solver implements mesh refinement and communication procedures.

3.1 The ExaHyPE Core

The ExaHyPE core is realised upon the third version of the PDE framework Peano. Peano creates and stores adaptive Cartesian meshes and realises the traversal of these meshes, too. Just as ExaHyPE applications are plugged into ExaHyPE via automatically generated glue code, ExaHyPE's algorithms are plugged into Peano's mesh traversal automaton. In fact, ExaHyPE's glue code generation is inspired by Peano's glue code generation.

Peano decomposes a simulation into individual steps/phases. Per phase, certain operations are performed per cell or face read. Roughly, such a phase corresponds to a traversal of the mesh that executes multiple substeps. Peano calls the simulations steps *adapters* and the latter substeps *mappings*. Mappings are the classes that are written by the Peano application developer. Before instructing Peano to run through the mesh, an adapter must be selected first. During the mesh traversal, Peano's traversal automaton then invokes callbacks on this

adapter, which in turn invoke the same callback on all of the adapter's mappings. ExaHyPE's mappings eventually invoke PDE-specific functions of ExaHyPE user solvers. UML diagram 3.3 models the interplay between ExaHyPE's mappings, solvers, and the runner class that steers the whole simulation.

ExaHyPE can simultaneously run multiple simulations on the same mesh. These simulations may populate different mesh levels and may have different adaptive mesh refinement patterns. It is ongoing research how to efficiently couple these simulations for multi-physics simulations and UQ (uncertainty quantification) applications. However, it is also possible to integrate ExaHyPE as forward solver into classic multi-physics and UQ pipelines [89]. Here, the ExaHyPE core's plotters and infrastructure for reducing global metrics are useful tools to extract quantities of interest from a simulation. Plotters can be quickly added to a simulation via the toolkit (see Chapter 4). After the initial generation, they can be customised freely by users.

3.2 Toolkit, Optimised Kernels, and Debugging/Benchmarking Ecosystem

ExaHyPE's toolkit is written in `Python 3`. Project-specific glue-code generation is realised via the jinja2 template engine [12]. JSON Schema is used to validate ExaHyPE specification files [10]. A JSON (JavaScript Object Notation) schema defines validation information for files in the JSON format or objects in JSON representation. It is typically written in JSON itself. It allows to specify valid ranges for integers, options for string enums, optional and mandatory parameters, and dependencies.

ExaHyPE specification files must not necessarily be written in JSON format [11]. A pre-parsing step converts other compatible specification file formats such as the original ExaHyPE format into a JSON representation. JSON encodes data objects as simple objects that either contain other data objects, primitive data types (integers, strings, ...), or arrays of the first two.

Optimised Kernels

The optimised kernel generator is a `Python 3` subprogram executed by the toolkit [58]. It generates aggressively vectorised ADER-DG kernels targeted to Intel CPU architectures. To this end, the generator relies on LIBXSMM [68], which generates hardware-tailored inline

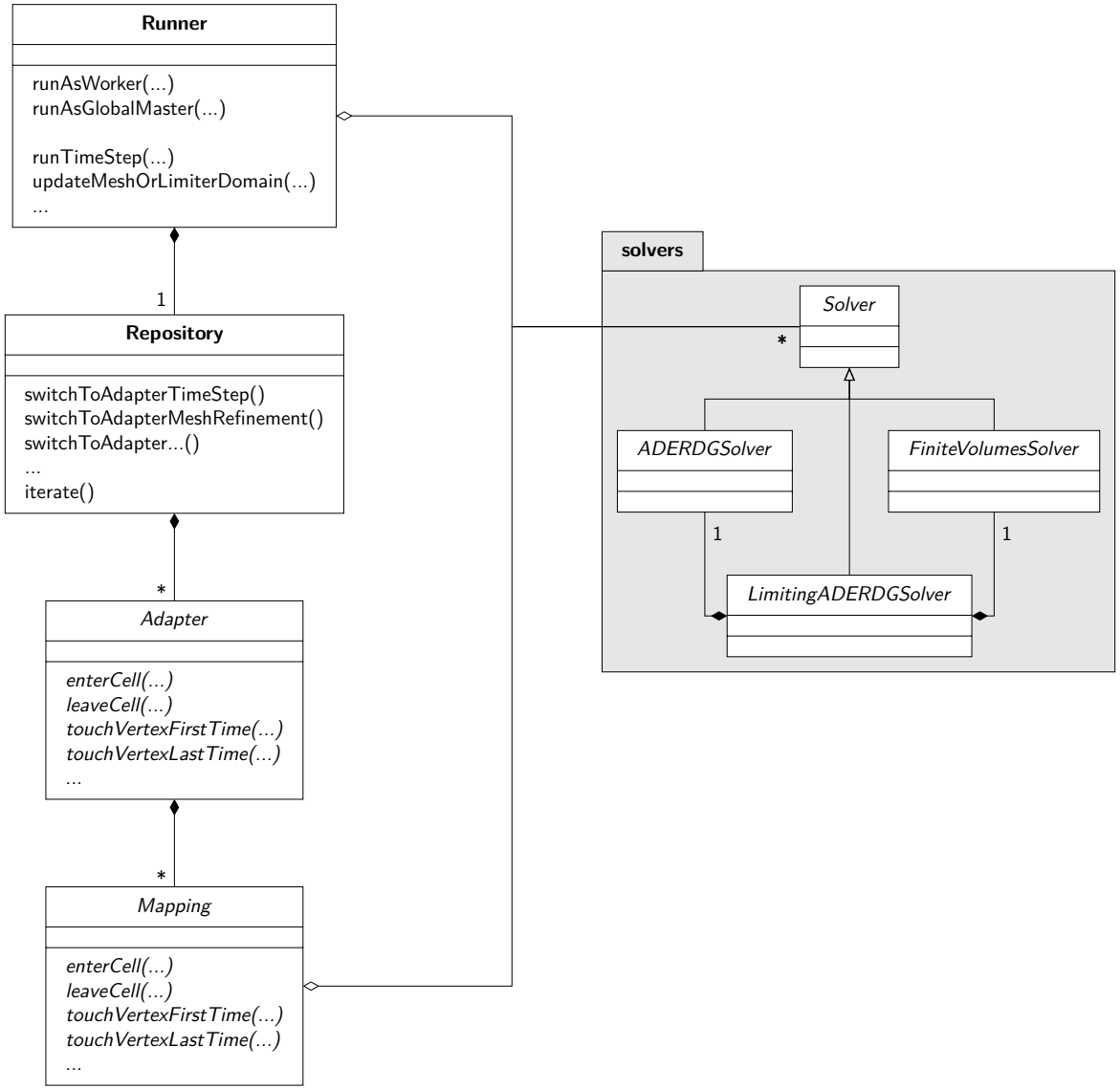


Fig. 3.3: Conceptual class diagram showing ExaHyPE's abstract solver base classes, their relationships, and their integration into ExaHyPE's remaining infrastructure. The diagram omits a further abstraction layer between repository and adapters. ExaHyPE's runner, repository, adapters, and mappings have been initially generated with Peano's code generation tool. Peano does not realise adapters and mappings as specialisations of an abstract class. Instead, Peano's code generation tool directly generates specialisations and the glue-code between repository, adapters, and mappings.

assembly code for small matrix-matrix multiplications as they are common in ADER-DG kernels.

Benchmarking, Profiling, and Debugging

Reproducibility of results is steadily becoming a very important topic in high-performance computing and scientific computing in general. ExaHyPE provides a `Python 3` tool named `sweep.py` to conduct reproducible benchmarks and parameter studies, which can be easily ported from one supercomputer to the other. Furthermore, the tool's input scripts help developers to understand what simulations the users have performed and to determine if issues that users experience stem from the hardware, ExaHyPE, or their application. ExaHyPE relies on Peano's infrastructure for debug output and assertions. Both codes use plenty of both. Log output can be filtered individually per application. In addition, both codes are populated with user-defined regions for tracing with ITAC [112] and Score-P [72]. These are essential for understanding asynchronous communication patterns and the behaviour of ExaHyPE's multi-threading implementation.

4

Example Usage: Solving the Euler Equations

When choosing a software package to solve science and engineering problems, it is important to quickly get hands-on experience with a running code. In this chapter, I showcase a typical ExaHyPE user story. I realise a dynamically adaptive, multi-threaded simulation of a nonlinear hyperbolic PDE plus a plotter to visualise the numerical solution at different time stages. All this requires less than 100 lines of code.

Tip: This chapter how cases how an application scientist would use ExaHyPE, and does not reveal anything about how ExaHyPE's algorithmic building blocks work. It is not necessary to read this chapter to understand the other chapters of this thesis. Feel free to skip it now and to read it later.

Problem Formulation

I want to solve the compressible Euler equations in 2D,

$$\frac{\partial q}{\partial t} + \nabla \cdot F(q) = 0, \tag{4.1}$$

with state variables $q = (\rho, \rho v, \rho E)^T$. Here, ρ, $v = (v_x, v_y)^{\mathrm{T}}$, and E are mass density, velocity, and energy density, respectively. The product ρv forms the moment density.

The flux F and the pressure p are given as:

$$F = \begin{pmatrix} \rho v \\ \rho\, v \otimes v + pI \\ v(\rho E + p) \end{pmatrix}, \qquad p = (\gamma - 1)\,(\rho E - \frac{1}{2}\,\rho v \cdot v),$$

where $(v \otimes v)_{ij} = v_i\, v_j$, $i, j = 1, 2$. The pressure is defined according to the EOS of a perfect gas with adiabatic index γ. While this is a small PDE with only four unknowns, it is difficult to solve as the flux is nonlinear in q.

I want to solve the compressible Euler equations with a third order ADER-DG method in 2D. Being a high-order method, the ADER-DG method represents smooth solutions very precisely using only a small number of cells.

The problem that I consider is an entropy wave with the smooth analytical solution:

$$\begin{pmatrix} p \\ \rho v_x \\ \rho v_y \\ \rho E \end{pmatrix} = \begin{pmatrix} \rho_0(x,t) \\ \rho_0(x,t) \cdot v_{0,x} \\ 0 \\ \frac{p_0}{\gamma - 1} + \rho_0(x,t) \cdot (v_0 \cdot v_0) \end{pmatrix}, \tag{4.2}$$

where $p_0 = 1$ and $v_0 = (0.5, 0)^{\mathrm{T}}$. Here, $\rho_0(x, y, t)$ describes a moving Gaussian-like pulse that is initially centered at $x_0 = (0.5, 0.5)^{\mathrm{T}}$,

$$\rho_0(x, y, t) = 0.5 + 1 \cdot \exp\left(\frac{-||(x - x_0) - v_0\, t||_2}{0.3^2}\right),$$

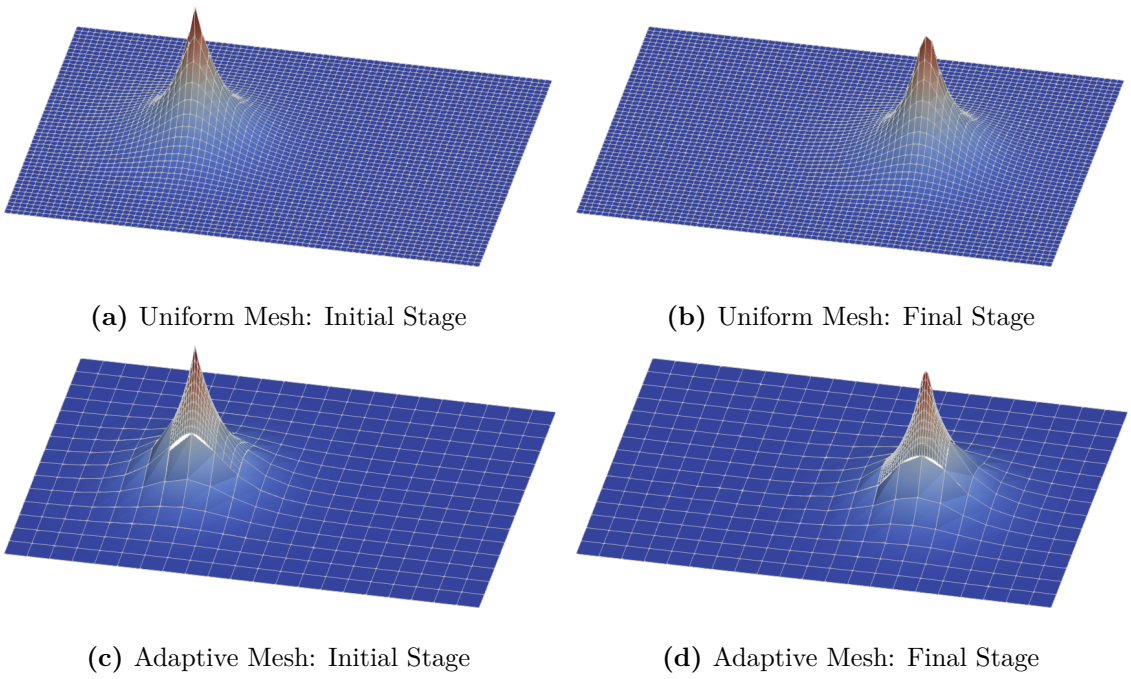
where $|| \cdot ||_2$ denotes the Euclidean norm.

Generating the Solver

The development of any ExaHyPE application always starts with the *specification file*. The minimal specification file to generate my ADER-DG solver look as follows:

```
exahype-project Euler
  output-directory const  = ./ApplicationExamples/ThesisCharrier/UserWorkflow
```

(continues on next page)



(a) Uniform Mesh: Initial Stage

(b) Uniform Mesh: Final Stage

(c) Adaptive Mesh: Initial Stage

(d) Adaptive Mesh: Final Stage

Fig. 4.1: (a) – (b): Snapshots of the advected pulse on a uniform mesh. (c) – (d): Snapshots of the propagation on an adaptive mesh. All plots show the warped density variable of the compressible Euler equations. The shown grid does not show the actual mesh cells. To visualise the high order polynomial approximation of the ADER-DG method in each cell, the cell-wise solution is sampled on an equidistant subgrid with four grid points per coordinate direction.

(continued from previous page)

```
  computational-domain
    dimension const            = 2
    offset                     = 0.0, 0.0, 0.0
    width                      = 1.5, 1.0, 1.0
    end-time                   = 1.0
  end computational-domain

  solver ADER-DG EulerSolver_ADERDG
    variables const     = rho:1,j:2,E:1
    order const         = 3
    maximum-mesh-size   = 0.12
    time-stepping const = globalfixed
    type const          = nonlinear
    terms const         = flux
    optimisation const  = generic
  end solver
end exahype-project
```

I use an `ADER-DG` solver named `EulerSolver_ADERDG`, where I specify the type of the PDE and its variables (`nonlinear`, `rho:1,j:2,E:1`), and the PDE terms that I want to use from (1.1). For the compressible Euler equations, I specify only that I want to use a `flux`. Furthermore, I set the approximation order to `3` and specify that I use a fixed time step size throughout the whole simulation (`globalfixed`). Additionally, I prescribe that I want to discretise the computational domain with a uniform mesh with mesh spacing not larger than `0.12`.

Next, I call ExaHyPE's toolkit to generate glue code into the output directory (`./ApplicationExamples/ThesisCharrier/UserWorkflow`):

```
Toolkit/toolkit.sh Euler.exahype
```

This generates C/C++ glue code to register my solver `EulerSolver_ADERDG` in ExaHyPE's solver registry, and further glue code to implement my solver's details. The glue code for the solver consists of two classes:

The abstract base class (`AbstractEulerSolver_ADERDG`) gathers all project-specific constants

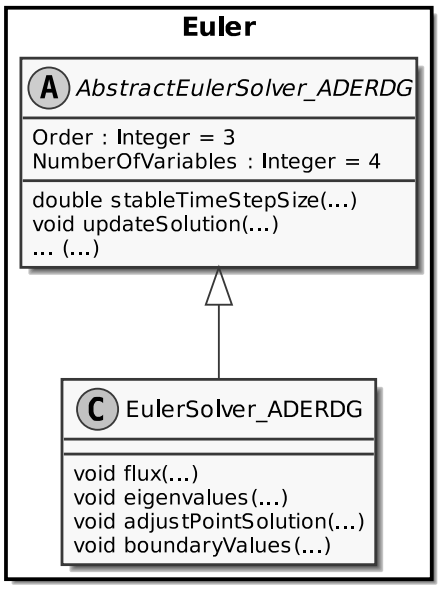


Fig. 4.2: Abstract base class and user solver. The user implements only functionality of the latter. The abstract base class is regenerated every time the toolkit is called.

(order, number of variables, ...) and implements the generic parts of the ADER-DG method's substeps. These ADER-DG substeps, in turn, call PDE terms that are defined in the user solver class (`EulerSolver_ADERDG`). While the abstract base class is regenerated every time the toolkit is called, the user solver class is only generated once.

In the user solver, I need to implement the PDE terms for the compressible Euler equations.

Specifying Flux and Eigenvalues

I start with specifying the flux term in the source file `EulerSolver_ADERDG.cpp`. I notice that the toolkit has already generated a stub for the flux routine as I have put `flux` as PDE term into the solver specification. I complete the routine as follows[1]:

```
void Euler::EulerSolver_ADERDG::flux(double* Q, double** F) {
  const double j2 = Q[1]*Q[1]+Q[2]*Q[2];
  const double p  = (gamma-1) * (Q[3] - 0.5*irho*j2);
  // col 1
  F[0][0] = Q[1];
```

(continues on next page)

[1] Note that the first element of an array has index 0 in C/C++, and that the const modifiers can be omitted in the source file signatures, but not in the ones in the header file.

(continued from previous page)

```
  F[0][1] = irho*Q[1]*Q[1] + p;
  F[0][2] = irho*Q[2]*Q[1];
  F[0][3] = irho*(Q[3]+p)*Q[1];
  // col 2
  F[1][0] = Q[2];
  F[1][1] = irho*Q[1]*Q[2];
  F[1][2] = irho*Q[2]*Q[2] + p;
  F[1][3] = irho*(Q[3]+p)*Q[2];
}
```

Next, I specify the eigenvalues of the linearised flux tensor:

```
void Euler::EulerSolver_ADERDG::eigenvalues(double* Q,int direction,double* lambda)
↪{
  constexpr double gamma = 1.4;
  const double irho      = 1./Q[0];
  const double j2        = Q[1]*Q[1]+Q[2]*Q[2];
  const double p         = (gamma-1) * (Q[3] - 0.5*irho*j2);

  const double u_n = Q[direction + 1] * irho;
  const double c   = std::sqrt(gamma * p * irho);

  lambda[0] = u_n - c;
  lambda[1] = u_n;
  lambda[2] = u_n;
  lambda[3] = u_n + c;
}
```

Specifying Boundary and Initial Conditions

Having the physics in place, I still need to specify boundary and initial conditions.

I impose the values of the analytical solution (4.2) at time $t = 0$ as initial conditions. By default, ExaHyPE assumes that I want to impose initial conditions per support point of the ADER-DG method. I complete the pre-generated stub as follows:

```
void referenceSolution(const double* const x,const double t,double* const Q) {
  constexpr double gamma = 1.4;
  constexpr double p     = 1.0;
  constexpr double v0    = 0.5;
  constexpr double width = 0.3;

  const double distX    = x[0] - 0.5 - v0 * t;
  const double distY    = x[1] - 0.5;
  const double distance = std::sqrt(distX*distX + distY*distY);

  Q[0] = 0.5 + 1.0 * std::exp(-distance / std::pow(width, DIMENSIONS));
  Q[1] = Q[0] * v0;
  Q[2] = 0;
  // total energy = internal energy + kinetic energy
  Q[3] = p / (gamma-1)  +  0.5*Q[0] * (v0*v0);
} // must come before adjustPointSolution and boundaryValues (see below)

void Euler::EulerSolver_ADERDG::adjustPointSolution(const double* const x,const␣
↪double t,const double dt, double* const Q) {
  if (tarch::la::equals(t, 0.0)) {
    referenceSolution(x,t,Q);
  }
}
```

As I have the analytical solution at hand, I prescribe Dirichlet boundary conditions:

```
void Euler::EulerSolver_ADERDG::boundaryValues(...,int direction,double* fluxIn,
↪double* stateIn,double* gradStateIn,double* fluxOut,double* stateOut){
  referenceSolution(x,t+0.5*dt,stateOut);

  double _F[3][NumberOfVariables]={0.0};
  double* F[3] = {_F[0], _F[1], _F[2]};
  flux(stateOut,F);
  for (int i=0; i<NumberOfVariables; i++) {
    fluxOut[i] = F[direction][i];
```

(continues on next page)

(continued from previous page)

```
  }
}
```

The centre of the Gaussian pulse stays far away from the boundary throughout the simulation. Hence, I simply evaluate the analytical solution in the middle of the time interval $[t, t + \Delta t]$.

Adding a Plotter

To understand what is happening during the simulation, I want to plot the solution. To this end, I insert a plotter definition into my solver's specification and let the toolkit generate the plotter class files:

```
exahype-project Euler
  [...]

  solver ADER-DG EulerSolver_ADERDG
    [...]

    plot vtu::Cartesian::vertices::ascii Plotter
      variables const = 4
      time            = 0.0
      repeat          = 1e-2
      output          = ./solution
    end plot
  end solver
end exahype-project
```

The plotter will create the first plot at simulation time `0.0` and additional plots every `0.01` simulation time units. All plot files are prefixed with `solution`.

Rerunning the toolkit generates a class `Plotter` that writes VTU files. As I want to plot all four variables of the compressible Euler equations, I extend the pre-generated stub in the class as follows:

```
void Euler::Plotter::mapQuantities(
```

(continues on next page)

(continued from previous page)

```
    ...,
    double* Q,
    double* outputQuantities,
    double timeStamp
) {
   outputQuantities[0] = Q[0];
   outputQuantities[1] = Q[1];
   outputQuantities[2] = Q[2];
   outputQuantities[3] = Q[3];
}
```

Building and Running the Simulation

The next step is to build the code. For now, I specify that I want to build in release mode with the GNU compiler, and turn off the distributed- and shared-memory parallelisation. Afterwards, I run make on the toolkit-generated make file in the project repository. The required steps to perform in the command line are given below:

```
export COMPILER=GNU
export MODE=Release
export DISTRIBUTEDMEM=None
export SHAREDMEM=None

make
```

At this stage it is convenient to copy the specification file into the project folder. From the project folder, I run the simulation via:

```
./ExaHyPE-Euler Euler.exahype
```

The output of the simulation can be visualised as shown in Fig. 4.1 (a) – (b).

4.1 Adaptivity and Parallelisation

In the first part of this chapter, I implemented a solver for the compressible Euler equations. It required me to write around 50 lines of code myself. So far, this implementation is running

in serial and uses a uniformly spaced mesh. With ExaHyPE, advanced techniques such as adaptive mesh refinement and multi-threading can be realised with minimal user effort on top of such a first implementation.

Dynamic Adaptive Mesh Refinement

To use adaptive mesh refinement in ExaHyPE, two modifications are necessary. First, I need to specify the maximum depth of the adaptive mesh on top of the uniform base mesh. Here, I specify two levels of AMR[2]:

```
exahype-project Euler
  [...]

  solver ADER-DG EulerSolver_ADERDG
    variables const     = rho:1,j:2,E:1
    order const         = 3
    maximum-mesh-size   = 0.4
    maximum-mesh-depth  = 2
    [...]
  end solver
end exahype-project
```

Second, I need to specify which mesh cells to refine and which to erase in the user solver implementation. Let ρ_K denote the maximum density in cell K. Below, I refine cells in vicinity of the density maximum ($0.75 < \rho_K/1.5 \leq 1.0$), and erase cells further away ($\rho_K/1.5 < 0.6$). I keep all other cells at their current mesh level. I extend the pre-generated `refinementCriterion` stub as follows:

```
exahype::solvers::Solver::RefinementControl
Euler::EulerSolver_ADERDG::refinementCriterion(double* luh, ...) {
  double largestRho  = -std::numeric_limits<double>::infinity();
  for (int i=0; i<(Order+1)*(Order+1); i++) { // loop over all nodes
    const double* Q = luh + i*NumberOfVariables;
    largestRho  = std::max (largestRho,   Q[0]);
  }
```

(continues on next page)

[2] From hereon, I use a slightly larger uniform mesh spacing than before for visualisation reasons.

(continued from previous page)

```
  if ( largestRho/1.5 > 0.75 ) { // maximum is 1.5
    return RefinementControl::Refine;
  } else if ( largestRho/1.5 < 0.6 ) {
    return RefinementControl::Erase;
  }
}
```

A snapshot of the adaptively refined solution is shown in Fig. 4.1 (c) – (d).

Multi-Threading

I want to use Intel's TBB to run my solver in parallel on a multi-core CPU. I thus recompile it with the following settings:

```
export COMPILER=GNU
export MODE=Release
export DISTRIBUTEDMEM=None
export SHAREDMEM=TBB

make
```

Furthermore, I add a `shared-memory` parallelisation block to the specification file. Here, I specify the number of threads that I want to use via the `cores` parameter[3]:

```
exahype-project Euler
  [...]

  shared-memory
    cores = 2
  end shared-memory

  solver ADER-DG EulerSolver_ADERDG
    [...]
```

(continues on next page)

[3] The parameter name "cores" is misleading. These are actually the threads that ExaHyPE uses.

(continued from previous page)

```
    plot vtk::Cartesian::vertices::ascii Plotter
      [...]
    end plot
  end solver
end exahype-project
```

After the built, I run the simulation the usual way.

More Features

Other features such as the distributed-memory parallelisation, the usage of optimised ADER-DG kernels, and the reduction of global observables can be activated and tailored via the toolkit, too. No further code must be written. For a comprehensive overview and documentation of all these features, I refer to the ExaHyPE guidebook [6].

5
Related Work

The development process of scientific simulation codes can be roughly decomposed into four main steps (Fig. 5.1): First, the continuous model of the problem is derived that is a PDE in strong form in the context of this thesis. Second, numerical methods are applied to discretise the continuous problem. The resulting discrete problem is an approximation to the continuous problem that can be solved on a computer. The discretisation reveals details on substeps, data fields, and data dependencies. This information is required to derive algorithms in the third step of the process. In the fourth and final step, the algorithms are translated into code.

Application scientists have expert knowledge on the continuous model. They aim to explain physical processes with the aid of computational methods. Furthermore, they are interested in refining the continuous model and to find its limitations by comparing simulations against measurements that they have obtained from experiments. Applied mathematicians have expert knowledge on numerical methods and the mathematical aspects of algorithms. The latter is an expertise they share with computer scientists, who are experts in translating algorithms into efficient software.

For application scientists, it is important to know where to start with the software development process to minimise the time to create a running simulation tailored to their problem. They

will ask the following questions:

1. Is there already an in-house, community, or commercial code that solves my problem?
2. Is there a PDE engine such as `PyClaw` [71], `FEniCS` [56], `Firedrake` [57], or ExaHyPE, which only requires me to express the physics of my problem and the rest of the solver construction is automated?
3. Is there a general-purpose numerics framework/toolbox, such as `Peano` [103] or `SUNDIALS` [66], where I find the discretisation, meshes, and algorithms that I want to use? In comparison to the PDE engines, I have to interface individual solver components myself when using such frameworks.

If all of the above does not apply, application scientists start developing from a meshing framework or in case they have very particular requirements, from a general-purpose programming language like texttt{C/C++}, texttt{Fortran} or texttt{Python}. A general rule is: The lower the abstraction level, the more flexibility in the implementation, but the more time-demanding the software development process will become.

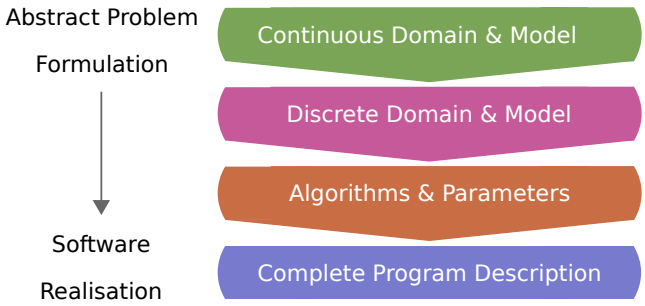


Fig. 5.1: The development of scientific software is performed in four steps. Application scientists understand the continuous model of the problem. Mathematicians develop numerical methods to discretise the continuous problem. Discretisations reveal details on substeps and data dependencies, which are necessary to derive algorithms. The algorithms are translated into efficient software by computer scientists. *Adapted from* [69].

Developers of software frameworks and engines face the same questions but from the other perspective when deciding how generic or specialised their product should be. ExaHyPE aims to solve a wide range of hyperbolic PDEs. Its underlying numerical methods must therefore be generic, accurate, and robust. To allow simulating large-scale problems on supercomputers, these methods must exhibit high arithmetic intensity and low communication cost.

Structure

The first section of this chapter compares different numerical methods for solving hyperbolic differential equations. The second section groups the scientific computing software stack into four categories. I show where ExaHyPE fits. Following this review, I list the requirements on ExaHyPE and discuss why ExaHyPE is built upon the PDE framework Peano.

5.1 Numerical Methods for Hyperbolic PDEs

A computer simulation's accuracy depends on the approximation quality of its numerical methods. Scientific simulations make errors in the spatial discretisation, in the time evolution, and in the representation of material and geometry. In order to obtain a scientifically meaningful result, these approximation errors must be controlled and reduced to an acceptable level. The efficiency of numerical methods can be measured by the computational work they require to keep the approximation errors below a certain threshold. This work can be often significantly reduced with adaptive procedures where more computational work is only invested where local approximation errors dominate the global approximation errors. ExaHyPE combines accurate and robust numerical methods with adaptive mesh refinement. All of ExaHyPE's building blocks have a long research history. This section aims to give a high-level overview.

Spatial Discretisation

ExaHyPE follows the Eulerian approach for solving hyperbolic PDE systems; it describes the evolution of the numerical solution to a hyperbolic PDE with respect to a spatial computational mesh. The solution is then decomposed into a spatially-varying and temporally-varying component. For both components an ansatz is made. The ansatz for each components makes assumptions on the solution. These assumptions might or might not be justified for the modelled wave phenomena. In the following, I discuss different approaches for constructing an ansatz for the spatially-varying component.

Hyperbolic partial differential equations can be roughly grouped into two main groups: conservation laws and non-conservative equations. Finite volumes methods (FVM) and DG methods are considered a natural choice for discretising conservation laws [64] in space yet are also applicable to hyperbolic PDE systems that contain non-conservative terms [36][49][81]. In the FVM, the computational domain is decomposed into many small cells (*volumes*), which

together form the computational mesh. Each cell holds volume averages of the state variables. Godunov type finite volume methods compute the numerical flux at the interface of the cells as the solution of a Riemann problem [61]. The volume averages stored in both cells that share the interface are used as left and right initial values of the Riemann problem. In practice, the Riemann problem is typically not solved exactly but an approximate Riemann solver is employed instead. Approximate Riemann solvers differ in the number of wave characteristics they model; see [95] for a comprehensive overview of exact and approximate Riemann solvers. After the Riemann solve, high-order finite volume methods, so called high-resolution schemes, reconstruct polynomials from the volume averages of cells and a number of their neighbours [98][35][62]. Discontinuities in the solution pose a difficulty for high-order variants of the FVM. Discontinuous initial solutions introduce non-physical oscillations into the numerical solution that evolve in time. For nonlinear problems, such discontinuities can form over time even from smooth initial data; see e.g. [64]. Due to non-physical oscillations, physical quantities can assume non-physical values during a simulation, too. Therefore, artificial oscillations must be prevented or removed from the solution or at least damped. Such a procedure is called *limiting*. A comparison of limiters for high resolution schemes can be found in [91].

The FDM (finite difference method) is a straightforward recipe to derive discretisations for solving hyperbolic problems that admit smooth solutions. It is a grid-based method that approximates the derivatives in the strong formulation of the PDE by finite differences. Constructing difference stencils is simple for linear PDE systems; however, it becomes cumbersome for nonlinear problems. Moreover, if material parameters are strongly heterogeneous or the geometry is complex, difference-stencil-based approximations to strong derivatives are not well-defined or difficult to construct. Lastly, high-order FDM stencils enforce that the numerical solution is more differentiable. This is not appropriate if the real solution contains discontinuities.

Similar to the FVM, the continuous Galerkin FEM (finite element method) is derived from an integral or *weak* formulation of the hyperbolic PDE system. The FEM weak formulation is a variational formulation of the strong formulation that relaxes the differentiability assumptions on the (numerical) solution by "moving" spatial derivatives from the solution (ansatz) to a smooth test function. Like the FVM, FEM decomposes the computational domain into a finite number of cells, which are named *elements* in the FEM context. Being based on a weak

formulation, FEM imposes less strict requirements on the differentiability of the solution than FDM. However, FEM still enforces continuity at the interface between cells. If we apply them naively, FDM and FEM introduce spurious oscillations into the numerical solution in the vicinity of discontinuities due to their in-built assumptions on the continuity of the solution. This is known as Runge's phenomenon or Gibbs phenomenon. Both methods can be applied to simulate such phenomena if artificial viscosity is introduced to the original hyperbolic PDE system [100]. Evolving the numerical solution with an explicit time stepping scheme requires to compute the inverse of the FEM mass matrix. Here, a second difficulty arises: The FEM mass matrix has multiple off-diagonals, i.e. its inverse is dense [64]. Therefore, inverting the mass matrix is not feasible if the number of elements is large. In practice, this issue is mitigated via mass lumping techniques that approximate the mass matrix or its inverse so that the inverse is sparse. However, this requires care to not violate physically motivated constraints such as the conservation of mass.

The DG method can be read as hybrid between FEM and FVM. It represents the solution as polynomial in every cell similar to FEM, and it couples the solution polynomials of neighbouring cells by means of numerical fluxes similar to the FVM. In comparison to FEM, DG does not enforce continuity at cell interfaces. Therefore, h-adaptive DG methods can be realised with less effort compared to h-adaptive finite volume or finite element methods [64]. Another advantage of DG methods is that the mass matrices of DG discretisations are block diagonal (non-orthogonal shape functions) or fully diagonal (simplices and quadrilaterals/hexahedrals) as DG employs ansatz functions with compact support. This enables efficient explicit time marching schemes [64]. Moreover, compared to FEM and FDM, DG has a compact communication stencil as neighbour cells only communicate via the faces and not via edges and corners.

The first DG method was introduced in 1973. It was used to solve the hyperbolic neutron transport equation [82]. Since then, DG methods have been applied successfully to many different hyperbolic conservation laws. Well-known is the papers series on RK-DG methods for conservation laws by Cockburn and Shu [42][41][43][40][44], which provides a theoretical foundation for the method. While DG methods satisfy a local entropy inequality and are thus nonlinearly stable in the $L^2(\Omega)$ norm [54], the numerical solution produced with these methods also exhibit artificial oscillations in the vicinity of strong gradients. This can be

attributed to the fact that the method's cell-wisely polynomial ansatz is subject to Runge's phenomenon, too. Consequently, slope limiting approaches for the RK-DG method have been presented. The first in [42][41]. An extensive overview of limiting approaches for DG is given in [54].

ExaHyPE uses DG and FVM due to their nonlinear stability and robustness with respect to discontinuities and shocks.

Temporal Discretisation

A spatial semi-discretisation must be paired up with an equally accurate temporal discretisation to obtain optimal convergence of the approximation error with respect to the mesh resolution. Therefore, an accurate discretisation of the temporal derivative is of high importance for the numerical solution of time-dependent hyperbolic PDEs.

RK (Runge-Kutta) methods are a popular choice for the temporal discretisation. They approximate the first elements of a Taylor series expansion in time. This yields a multi-stage algorithm; see e.g. [32]. Up to the fourth-order method, the number of RK stages is proportional to the order of the GTE (global truncation error) of the methods. Above this *Butcher barrier*, the methods become more difficult to construct as more and more conditions must be satisfied, and the number of RK stages grows faster than the order of the GTE. Sixth order schemes, e.g., require seven stages [32][33].

The ADER (Arbitrary DERivative in time) temporal discretisation technique is a completely different approach to obtain high-order accuracy in time [93][94]. This time discretisation is tailored to FVM and DG methods that use a Riemann solver. The original ADER method uses a Taylor expansion in space-time and the subsequent application of a CK procedure for computing temporal derivatives from the spatial derivatives. The temporal derivatives are used as initial values for the generalised Riemann problem that is solved at the interface between cells.

Recently, an ADER approach was presented that uses a cell-local space-time DG method to compute input data for the generalised Riemann problem [50]. The method is more easily adapted to nonlinear problems than the original ADER method and has been successfully applied to the FVM and DG; see e.g. [59][54]. Similar to the original ADER method, which uses the CK procedure, the local space-time ADER-FVM and ADER-DG methods

are one-step schemes. They perform only a single communication step per time step. See [59] for a comparison of different ADER variants applied to the FVM and DG spatial semi-discretisations.

ExaHyPE favours the ADER-DG method over the RK-DG method due to its in-built high arithmetic intensity and genericity. High-order ADER-DG methods for linear and nonlinear are constructed according to generic recipes and are not more complex than low order variants. While the number of RK stages grows with increasing temporal approximation order [33], the algorithmic complexity of ADER-DG does not change with increasing order. Moreover, there exist generic recipes for constructing LTS (local time stepping) algorithms for the ADER-DG method for both linear and nonlinear problems [53][86]. Such a recipe does not exist for RK-DG methods.

A disadvantage of the ADER-DG methods is that they require to use smaller time step sizes than equivalent RK-DG methods; see [44][48]. An 8-th order ADER-DG method, e.g., must use a roughly 3× smaller time step size than its RK-DG equivalent (see Table 5.1). However, ADER-DG is efficient in exploiting modern hardware. Despite using a smaller time step size, ADER-DG can yield a shorter time to solution due to the method's higher arithmetic intensity [85]. ADER-DG requires no more than three loops over the mesh independently of the approximation order as opposed to RK-DG schemes, where the number of RK stages grows with increasing temporal approximation order [32]. Being one-step schemes, ADER-DG schemes are better suited for strong-scaling than RK-DG methods [51], where one communication step is required per RK stage.

Table 5.1: Stable time step size of the ADER-DG method in comparison to the RK-DG method's time step size. *Data up to and including order 4 stems from* [Dumbser:08:Unified]. *Data for higher order were found experimentally; they are taken from the ExaHyPE repository* [ExaHyPE:19:Download].

Order	0	1	2	3	4	5	6	7	8
$\frac{\Delta T_{\text{ADER-DG}}}{\Delta T_{\text{RK-DG}}}$	100.0 %	99.0 %	85.0 %	70.0 %	62.1 %	49.5 %	49.4 %	45.0 %	34.0 %

Adaptive Mesh Refinement

Hyperbolic PDE systems are often characterised by multitudes of scales in space and time. Small and large scale features of the conserved quantities evolve, appear, and disappear in time. Therefore, an efficient numerical method requires a computational mesh that is dynamically adapted to these features. Numerical methods working with uniformly refined meshes might perform many unnecessary computations and approach memory limits in order to resolve localised small scale features in the conserved quantities. Additionally, local time stepping is required since different parts of the mesh will evolve in time with different time step sizes based on the wave propagation speed in these cells and the multiple spatial scales that are introduced by the adaptive mesh refinement to resolve small scale features in the conserved quantities. Dynamic adaptive mesh refinement plus local time stepping for hyperbolic PDEs introduced in the seminal papers [25][26]. Nowadays, very sophisticated adaptive mesh refinement codes are available. State-of-the-art codes support distributed memory parallel programming [31][103] as well as shared-memory parallel programming [103]. They are used to realise scalable simulation codes that utilise thousands of CPU cores.

There is a large number of open-source AMR packages available [111]. They can be categorised into block-structured and tree-structured AMR codes. In the hyperbolic PDE context, block-structured AMR codes were employed first [26]. They work with a collection of overlapping blocks of cells. Blocks with finer mesh spacing are introduced in areas where high resolution is required [47]. They hold ghost layers that are filled from coarser blocks before all cells on the fine grid block are updated. After the fine grid update, fine grid information is represented on the coarser blocks to enable the next cell updates there. Block-structured AMR is typically employed for the FVM and the FDM, which have a low computational cost per single cell or grid node.

FEM and DG schemes, which have a larger memory footprint per cell, are typically implemented on top of tree-structured AMR meshes. Here, the mesh is refined by splitting mesh cells into k^d child cells, where d denotes the spatial dimension and k the refinement factor. If this procedure is run for multiple iterations, a mesh is created that exhibits a tree structure, i.e. every mesh cell is subject to a unique parent-child relationship. Many tree-structured AMR codes do not store refined parent cells in the adaptive mesh but just the leaf cell [103]. The tree-structured AMR code Peano is an exception in this regard.

A merger of tree-structured and block-structured AMR codes are patch-based tree-structured AMR codes. They use a tree-structured adaptive mesh as organisational structure for storing regular blocks of cells. Tree-structured AMR codes are thus a well-suited data structure for realising AMR codes for the FVM, too [97][30].

5.2 ExaHyPE in the Scientific Computing Landscape

With ExaHyPE, application scientists only require knowledge about the continuous model to build a full simulation code as ExaHyPE prescribes the numerical scheme and in turn hides its internals. ExaHyPE itself is realised upon the PDE framework Peano [103][105][102][101]. Framework-upon-framework approaches that decompose software into different abstraction layers is a successful concept in scientific computing. Similar examples are `PeanoClaw` [16][97][96] and `ForestClaw` [30][34], patch-based FVM codes for solving general hyperbolic PDE systems, which extend the FVM package `Clawpack` with parallel, dynamic AMR. `PeanoClaw` relies on Peano to create and traverse adaptive meshes, `ForestClaw` on `p4est` [31].

An emerging trend is to tailor mature general-purpose frameworks to particular application areas in order to build community codes. This is the approach of `ASPECT` [76][3] , which builds a code for simulating convection processes in the Earth's mantle and crust based on `deal.II`, and of `GeoClaw` [71][17], which builds a code for simulating geophysical flows like tsunami waves based on `Clawpack` and `PETSc` (via `PyClaw`). Other examples include `ExWave` [113][87], a code for photo-acoustics that is based on deal.II, `ExaSeis`, which builds seismic wave propagation codes upon ExaHyPE [4], and `Dune-composite`, which is built upon `DUNE` [23][5]. Many community codes are able to outperform problem-tailored commercial software; see e.g. [84]. This performance advantage is often attributed to the fact that open-source software benefits directly from all advances in numerical methods, algorithms, and kernel optimisation of their individual open-source components. A comparison of a wide range of open-source FEM frameworks and closed-source commercial tools can be found at [1]. I categorise scientific software into four categories (Fig. 5.2):

1. Community codes, in-house codes, and commercial tools,
2. PDE engines,

3. generic, general purpose numerics frameworks plus scientific computing toolboxes, and
4. pure algorithm frameworks, e.g. standalone parallel mesh refinement and iterative solver libraries.

The lines between different categories are often blurred. For example, a number of numerics frameworks implement their own mesh refinement algorithms and iterative solvers. `DUNE`, e.g., implements its own collection of parallel iterative solvers [27][28]. Furthermore, specialised software must not necessarily rely on components from the next more generic stack. Many codes of the first category are written from scratch with a general-purpose programming language such as C/C++ or Fortran.

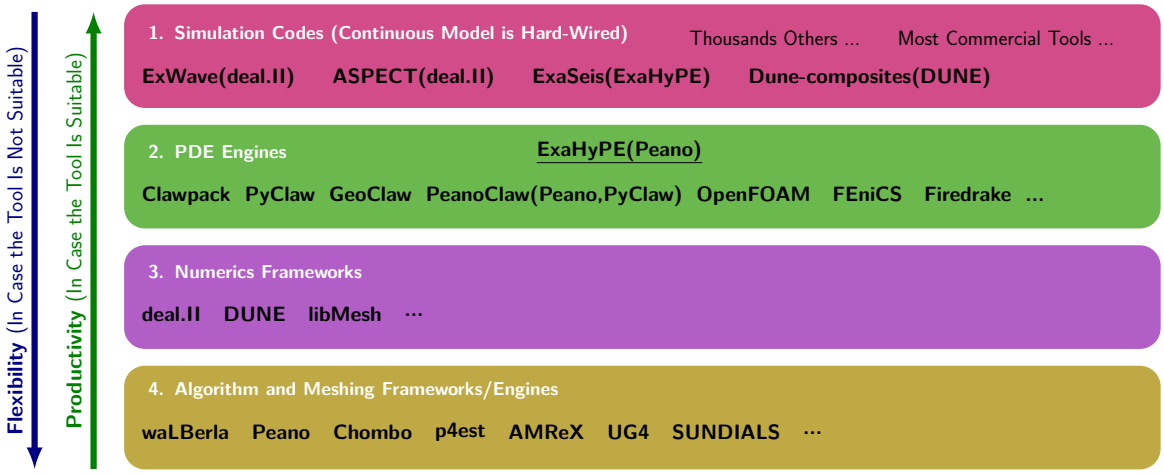


Fig. 5.2: An attempt to categorise the scientific computing landscape into four categories: simulation codes, PDE engines, numerics frameworks, and algorithm frameworks. The overview is not comprehensive at all. Application scientists and framework developers have more flexibility if they start code development based on an algorithm or numerics framework. However, they will be required to write more code to realise their application.

ExaHyPE, being a PDE engine, is highly specialised in comparison to general-purpose numerics and algorithm frameworks. The question arises if ExaHyPE itself can be built upon a more generic software package. Three requirements in the ExaHyPE agenda constrain the search for a suitable framework [8]:

- ExaHyPE must be able to solve nonlinear PDE systems. Per cell, the ADER-DG method for nonlinear PDE systems solves a locally-implicit equation system, which

requires a number of Picard iterations that varies from cell to cell.

- To realise a novel multi-level approach to UQ, ExaHyPE must be able to solve multiple wave propagation problems simultaneously on the same mesh. These simulations may exhibit different refinement patterns, and individual simulations may be coupled in areas where they use different mesh resolutions.

The local coupling of DG and FVM discretisations cannot be realised out-of-the-box with high-level FEM engines and frameworks such as `deal.II`, `FEniCS` and `Firedrake`, or FVM frameworks such as `OpenFOAM` [107][14]. To the best of my knowledge, it is not possible in general-purpose FEM frameworks to apply nonlinear operators to a cell's solution in a straightforward way. Consequently, ExaHyPE is built on top of one of the more generic AMR frameworks. To enable the multi-solver functionality required for the novel UQ approach, ExaHyPE prefers the AMR framework Peano over other parallel meshing frameworks. Peano keeps refined parent cells in the mesh and they are traversed by Peano's mesh traversal. Therefore, these cells can be relabelled as the leaf cells of a solver that operates on a coarser grid. In addition, Peano provides callbacks for realising data exchange between cells that reside on different mesh levels.

Use and Feel

Similar to the other PDE engines, ExaHyPE is more specialised than general-purpose numerics frameworks and less specialised than codes of the first category. Compared to the other engines, ExaHyPE offers a more guided approach to the development of simulation codes. ExaHyPE inherits a programming paradigm named *The Hollywood Principle* from Peano [103]: "Don't call us! We'll call you!". Applications built upon ExaHyPE do not have control over the order mesh cells are traversed, e.g. Instead, application scientists write a single configuration file and implement pre-generated function stubs. They are guided via assertions to the code sections that they need to implement. Configuration file verification code helps them to specify all compile- and run-time parameters that are required to implement and run their application. Consequently, application scientists do not need to have any knowledge about fundamental data structures of ExaHyPE.

6

Adaptive Mesh Refinement

In this chapter, I present ExaHyPE's adaptive mesh refinement capabilities. Adaptive mesh refinement is important for many of ExaHyPE's application areas. While all fluid dynamic problems require higher resolution around shock fronts and solution discontinuities, diffuse interface methods, for example, additionally require high resolution in vicinity of the approximated geometry. In this chapter, I detail ExaHyPE's mesh refinement data structure, its parallel decomposition, and the algorithms to build it. ExaHyPE uses Peano's spacetree as meta data structure and embeds arbitrarily spaced regular computational meshes into the tripartitioned spacetree. On top of a given regular base mesh, ExaHyPE performs adaptive mesh refinement according to a user-defined refinement criterion. ExaHyPE's implementation of the ADER-DG method is able to deal with arbitrary mesh resolution jumps. I realise this in ExaHyPE via a regularisation of the spacetree where I place virtual subcells into the spacetree. Moreover, I propose a procedure to eliminate local master-worker communication for the prolongation and restriction of face data in ExaHyPE's adaptive meshes. Finally, I present two pre-refinement techniques that can tackle unpredictable and predictable mesh adaptations. Their implementation is straightforward on top of the presented mesh data structure.

Contributions

I present a mesh data structure, the corresponding mesh adaptation algorithms and the required kernels to implement unconstrained AMR with the DG method. ExaHyPE's mesh refinement algorithms prevent loss of fine grid information where possible– up to the user's refinement criterion. Based on this data structure, I propose a technique to eliminate local master-worker communication in ExaHyPE's distributed meshes that are built upon Peano's spacetree. I propose two novel mesh pre-refinement flavours, *halo refinement* and *a posteriori refinement*. Halo refinement aims to reduce numerical issues and loss of information at the interface between coarse and fine grid cells. It traces interesting features and ensures that the mesh is pre-refined around them. A posteriori refinement allows to pre-refine the mesh where new features emerge, e.g. where shocks form or where a wave travels into the domain from the domain boundary. The fundamental assumption is here that a single step of an explicit time stepping scheme is cheap. We can thus do a (wrong) step and rollback to the previous solution if the mesh was not refined properly.

Related Work

In [24], the author emphasises the importance of pre-refinement when solving hyperbolic PDE systems on adaptive meshes. He proposes to always keep the previous solution and to roll back to it whenever the adaptive mesh did not suffice too keep the solution quality at the desired level. In this case, the mesh should be further refined first. The a posteriori refinement approach presented in this chapter realises such a scheme. I present detailed pseudocode for all steps of the scheme. In particular, the coarsening was not covered in [24].

Structure

Section 6.1 gives an overview of Peano's spacetree meshes. Furthermore, it addresses how the tree is decomposed if a Peano application is run in parallel. Section 6.2 presents ExaHyPE's mesh data structure and detail the algorithms to build it. Section 6.3 details how ExaHyPE's mesh data structure is used to identify and eliminate local synchronisation points that can arise in parallel simulations due to the spacetree partitioning. Section 6.4 discusses the two pre-refinement techniques, a posteriori refinement and halo refinement. In Section 6.5, I introduce inter-grid transfer operators that allows unconstrained jumps in the refinement levels of neighbouring cells. ExaHyPE's mesh data structure grid data structure is flexible enough to realise this. Lastly, Section 6.6 discusses the presented techniques and links to

subsequent chapters and future work.

6.1 Parallel Adaptive Spacetree Meshes

ExaHyPE plugs into Peano's hierarchical Cartesian meshes to create and adapt the computational mesh of a solver. For a spatial dimension $d \in \{2, 3\}$, Peano forges these meshes from a k^d-spacetree data structure [22]:

Definition (k^d-Spacetree).

A tree is called a k^d -spacetree if it has the following properties:

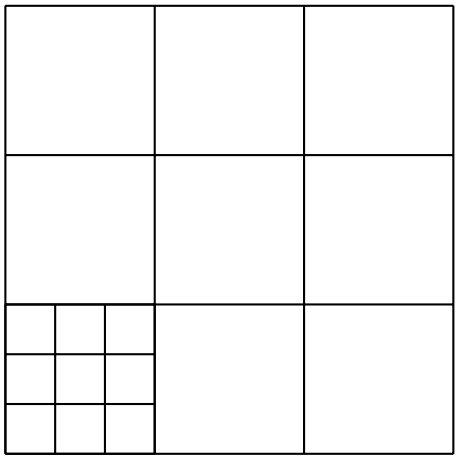
- *Each node of the tree is either a leaf or has exactly k^d children. Each child node is again a k^d spacetree.*
- *Each node represents a d-dimensional hypercube.*
- *Children hypercubes have edge lengths that are by a factor of k^{-1} smaller than edges of their parent.*

The process of splitting tree cells into k^d smaller cells is called (mesh) *refinement*. The level that a hypercube is associated with is called *refinement level*. The coarsest cell has level 0, its children level 1 and so forth. If all unrefined cells, i.e. cells without children, are on the same refinement level, the spacetree is called *regularly refined*. Otherwise, it is called an *adaptive* spacetree [22]. The decision what cells are refined depends on the particular application. The broad definition of k^d-spacetrees encompasses the well-known quadtree and octree data structures, which are $(k = 2)^d$-spacetrees. Peano realises a $(k = 3)^d$-spacetree (Fig. 6.1). A k^d-spacetree $\mathcal{T}$ can be decomposed into a mesh hierarchy:

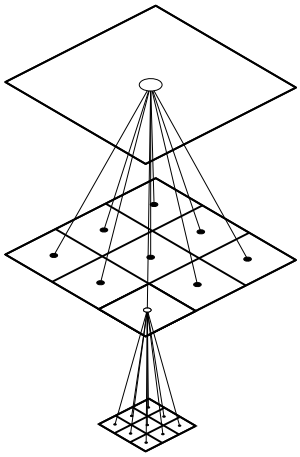
$$\mathcal{T} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \ldots \cup \mathcal{T}_{l_{\max}}$$

where $l_{\max}$ is the maximum refinement level. The level-wise meshes $\mathcal{T}_i$, $i = 0, \ldots, l_{\max}$, must not necessarily be connected, i.e. they may consist of isolated islands. Cells in mesh $\mathcal{T}_{i+1}$ are children of cells in mesh $\mathcal{T}_i$ $i = 0, \ldots, l_{\max} - 1$.

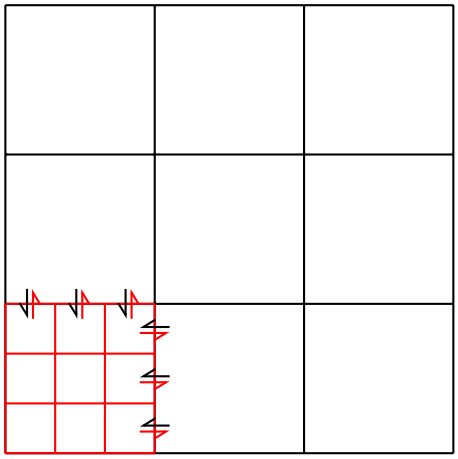
In terms of storage, $(k = 3)^d$-spacetrees can be efficiently encoded using a Peano SFC (space filling curve) [102][22]. Storing the $(k = 3)^d$-spacetree requires only a single bit per cell that encodes if the cell is refined or not. Peano realises a second aspect of PDE solvers aside from the spacetree adaptation: the spacetree traversal. The spacetree traversal strictly follows the



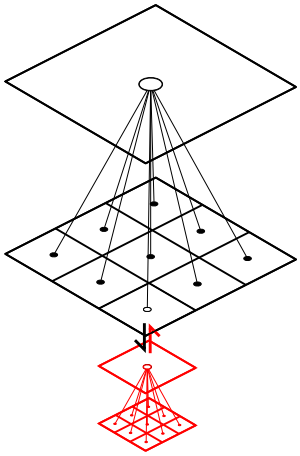
(a) Adaptive Mesh



(b) Grid Hierarchy



(c) Neighbour Communication



(d) Vertical Communication

Fig. 6.1: (a) An adaptive Cartesian mesh. (b) Its representation as hierarchy of regular Cartesian meshes. The adaptive mesh exhibits a tree structure. In particular, it exhibits the structure of a k^d-spacetree with $k = 3$ and $d = 2$. (c) – (d) PEANO partitions a spacetree by cutting off a subtree (red). (c) At the interface between two partitions, neighbour communication is required; only communication between leaf cells is shown. (d) Processors may be required to exchange mesh refinement flags and simulation data at the root of a subtree.

Peano SFC. User codes have no say on the order in which mesh cells and vertices are processed in. They must subscribe to events triggered by Peano's spacetree traversal automaton. Events align with transitions of the traversal from one spacetree entity (cell or vertex) into another. The start of a traversal (`beginIteration`) is an event, moving from one cell into another is an event (`enterCell` or `descend` within the tree), loading a vertex for the first time is an event (`touchVertexFirstTime`), and so forth.

In Peano, all modifications to the spacetree are triggered via the vertices: A cell is removed from the spacetree if all adjacent vertices are flagged for erasing (logical and). A spacetree cell is refined if at least one of its adjacent vertices is marked for refinement (logical or). In the following sections, I hide this aspect as it unnecessarily complicates the algorithms. I assume that cells can be refined or erased directly. Different to most other spacetree codes, Peano stores and traverses interior nodes, i.e. refined cells, of the spacetree. All events triggered for spacetree leaf cells are triggered for refined spacetree cells, too.

Spacetree Partitioning

To parallelise a simulation, computational work must be distributed among multiple processors. In this chapter, I do not distinguish between the terms *processor*, *process*, or *MPI rank*. As the majority of computational work in a mesh-based PDE simulation can typically be linked to operations performed on the mesh cells or vertices, it is reasonable to use these entities as measure for computational work. Therefore, mesh refinement and mesh partitioning are often closely interlinked in spacetree codes. Having constructed a space-filling curve, it can be used to distribute load among multiple processors by cutting the curve into roughly equally sized sections. Each processor gets assigned mesh cells and vertices corresponding to one section.

Peano follows a different domain decomposition strategy. It cuts off subtrees from the spacetree partition of a processor and deploys them to processors that do not have work yet (Fig. 6.1). The processors that receive the subtrees are called *workers* of the original processor. The latter is called *master* in this relationship. Peano's tree partitioning introduces a tree topology on the MPI ranks. As a processor's partition ends at the boundary to another processor's partition, both neighbours have to exchange data at the interface of the partitions. Peano realises a multi-scale non-overlapping domain decomposition. The projection property of the Peano SFC ensures that neighbour communication is conducted in a deterministic order [102]. Peano applications often transfer data between different levels of the tree. Depending

on where the spacetree is cut, these applications may need to synchronise data between master and worker at the root of a worker's subtree, too. This *vertical* master-worker communication comes on top of the *horizontal* neighbour communication at the interface between partitions (Fig. 6.1). If Peano is asked to partition the spacetree, it will trigger additional events during the spacetree traversal. User applications can subscribe to them in order to transfer data between a master and its workers and between neighbours.

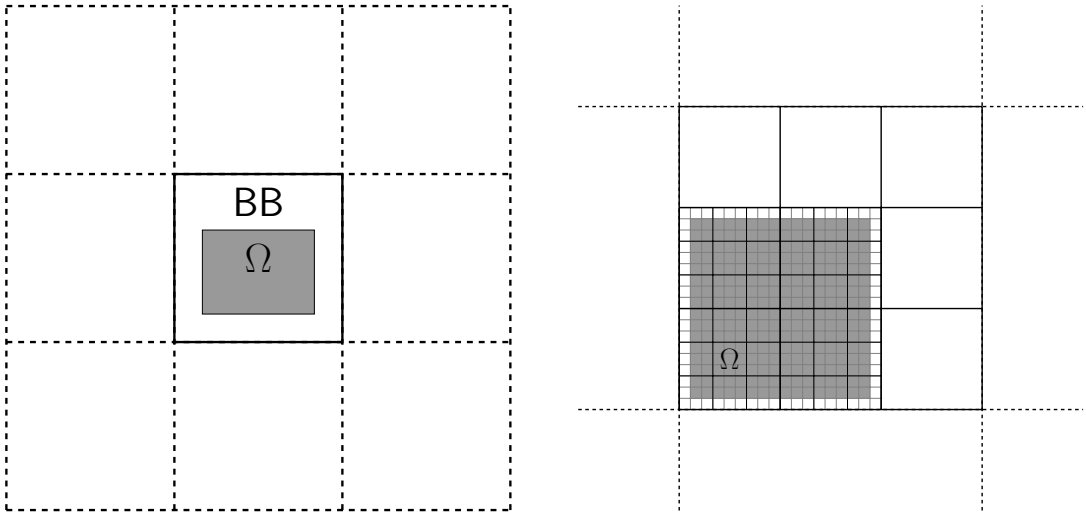
6.2 The Spacetree as Meta Data Structure

ExaHyPE relies on Peano's spacetree as a meta data structure. There are two aspects to this: ExaHyPE inherits Peano's marker-based approach to resolve geometries. It works only with cells that are at least partially in the interior of the computational domain; see Section 6.2.1. In addition, I augment the grid with helper cells to facilitate the inter-grid transfer operations of ExaHyPE's ADER-DG method and to detect where master-worker communication can be eliminated; see Section 6.2.2 until Section 6.2.8.

6.2.1 Embedding the Geometry

Peano starts building the spacetree from a single root cell and its 3^d children. The central child is called the *bounding box* as it embeds the computational domain (Fig. 6.2). Peano allows to position and scale the bounding box arbitrarily. ExaHyPE's computational domain is a simple box-shaped geometry and its numerical methods build an adaptive mesh from a regular base mesh that partitions the computational domain. I employ the bounding box scaling to control the mesh spacing of this regular base mesh. As a consequence, ExaHyPE can construct arbitrarily spaced regular base meshes even though Peano performs tripartitioning. For ExaHyPE applications, this is important to enable a comparison to results in the literature, where typically bipartitioned meshes are employed. ExaHyPE can handle other characteristic subdivision schemes besides bipartitioning. However, Peano's spacetree partitioning still adheres to the tripartitioning rule. A scaling of the bounding box may lead to an unfair distribution of work from the geometric perspective. Sometimes, the bounding box can be shifted by a number of cells to obtain fairer partitions.

The bounding box must be refined up to a level L to accommodate regular base mesh and bounding box shift. Given the desired number of mesh cells to resolve the computational domain (N_{inside}) and a number of bounding box cells ($N_{\text{outside,left}}$) to shift the domain, it is



(a) Bounding Box embeds Domain **(b)** 16×16 Mesh in Bounding Box

Fig. 6.2: (a) PEANO starts building a spacetree mesh from a tree with a single root cell and its 3^d children. The central child, the bounding box ("BB"), embeds the computational domain. (b) Embedding of a 16 times 16 mesh in the bounding box. The siblings of the bounding box are not shown.

computed as follows:

$$3^L \geq N_{\text{inside}} + N_{\text{outside,left}} \qquad \Rightarrow \qquad L = \lceil \log_3(N_{\text{inside}} + N_{\text{outside,left}}) \rceil.$$

For bounding box scaling and shift, this yields

$$\text{scaling} = \frac{3^L}{N_{\text{inside}}}, \qquad \text{shift} = -w \cdot \frac{N_{\text{outside,left}}}{N_{\text{inside}}}, \tag{6.1}$$

where w is the length of the longest edge of the computational domain.

In Fig. 6.2 (b), the bounding box is shifted by the equivalent of one bounding box cell to obtain a fair distribution of interior cells among 2^2 ranks. Note that PEANO will only employ workers in areas where there is an overlap with the computational domain.

Meshes for Parallel Computations

With (6.1), weak scaling of a regular computational mesh with 3^l cells per dimension can be accomplished by choosing $N_{\text{inside}} = k \cdot 3^l$, $k \in \mathbb{N}^+$. PEANO will then embed the resulting mesh into the spacetree at the bounding box mesh level that hosts the required number of cells. For example, a 9^d mesh fits into the second bounding box mesh level, which has 3^2 cells per dimension, while 18^d and 27^d meshes fit into the third bounding box mesh level.

In parallel simulations, Peano starts building the spacetree with a single process, the global master. After creating the root cell and the first 3^d cells, it distributes the central element to the first worker process. The latter continues with the mesh refinement and introduces more and more workers where the computational load is high. Fig. 6.3 (a) shows an example with six processes. Peano's spacetree partitioning strategy implies that regular meshes are only distributed fairly if 3^d, 9^d, 27^d, ... worker ranks are employed. Equation (6.1) allows to construct meshes where the first levels can be distributed by 2 and 4 processes per dimension instead of the 3 and 9, respectively, which are expected from the tripartitioning rule. For example, the 2D mesh in Fig. 6.2 (b) can be distributed among 4 workers with Peano.

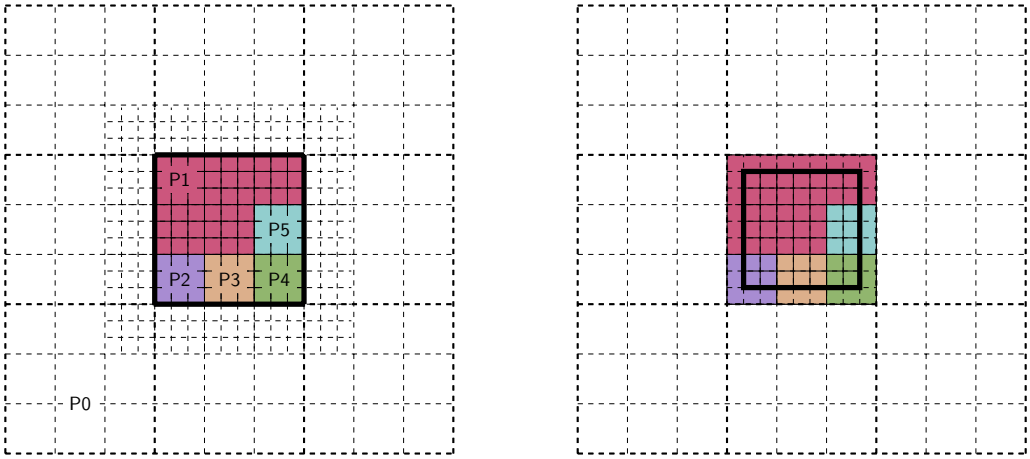
Refining cells in Peano is done via refining the vertices. Due to implementational details, this introduces additional outside cells on the global master if the boundary of the bounding box aligns with the boundary of the computational domain [15]; see Fig. 6.3 (a). The global master process runs through roughly half of its outside cells before communicating with the first worker. Therefore, the global master's traversal can become a bottleneck if the worker partitions are small compared to the outside cells of the global master. As Peano removes outside cells that are away from the domain boundary, this scalability bottleneck can be reduced by moving the computational domain into the interior by exactly one bounding box cell; see Fig. 6.3 (b). This can be accomplished via the bounding box scaling mechanism. For example, if the mesh should have roughly 9^d cells, then the choice $N_{\text{inside}} = 7$ and $N_{\text{outside,left}} = 1$ realises this. A disadvantage of this approach is that the load balancing for regular meshes is not optimal. This effect is less severe the larger the partitions are; see Table 6.1. In general, we found it advantageous to ensure that mesh partition boundaries do not align with domain boundaries to simplify the implementation and the load balancing.

6.2.2 Unconstrained Adaptivity via Virtual Cells

ExaHyPE puts an additional `type` marker into every spacetree cell that is inside of the computational domain. It distinguishes between `Empty`, `Leaf`, `Parent`, and `Virtual` cells. Together, they form the host mesh (Fig. 6.4 (b)), which ExaHyPE directly maps to Peano's spacetree. `Leaf` cells are the actual ADER-DG or FVM cells that allocate a solution vector. Per solver, they form the computational mesh (Fig. 6.4 (a)). `Parent` cells do not allocate any persistent memory at all. They indicate which cells have been adaptively refined and further where to place `Virtual` cells. `Empty` cells are spacetree cells that are not used (yet)

Table 6.1: The global master administers 3^d workers on a regular spacetree. Parallel efficiency if the bounding box is scaled such that 2 unrefined cells are placed outside of the domain per coordinate direction.

Workers	Bounding Box Cells	Domain Cells	Theo. Parallel Efficiency 3D (2D)
3^d	9^d	7^d	47.04% (60.44%)
3^d	27^d	25^d	79.37% (85.78%)
3^d	81^d	79^d	92.78% (95.11%)
3^d	243^d	241^d	97.56% (98.33%)
3^d	729^d	727^d	99.19% (99.44%)



(a) Domain Aligned With Bounding Box **(b)** Domain Inside Bounding Box

Fig. 6.3: (a) A mesh is distributed among six processes. The computational domain (thick black square) is aligned with the bounding box. (b) The computational domain is moved into the interior by one bounding box cell. The mesh resolution is reduced accordingly. One layer of outside cells on the global master (P0) has been removed.

by ExaHyPE. They might be introduced temporarily during ExaHyPE's mesh refinement operations, or persistently due to pre-refinement of the spacetree by Peano or ExaHyPE. Only if a solver performs adaptive mesh refinement, ExaHyPE introduces `Virtual` cells to the spacetree. They are placed around `Leaf` cells during the mesh adaptation phase. In ExaHyPE, `Virtual` cells are used to facilitate the ADER-DG face integral between `Leaf` cells that are located on different levels of the spacetree. Each `Virtual` cell has an attribute `ancestor` that refers to their ancestor of type `Leaf`. We present details on the exact handling of resolution transitions later.

Below, I modify the original ADER-DG algorithm (Algorithm 2.3) to distinguish between `Leaf` and `Virtual` cell's boundary DOFs (degrees of freedom) to the fine grid. A boundary extrapolation of space-time predictor and volume flux is necessary beforehand. After the Riemann solve, the traversal collects the result and performs the face integral on the interface between coarse and fine grid `Leaf` cells. The resulting update vector is added to the coarse grid `Leaf` cell's solution vector. With these modifications, the ADER-DG algorithm is written as:

Algorithm 6.1 (Global ADER-DG Time Stepping With Virtual Subcells)**.** *ADER-DG time stepping after introducing* `Virtual` *subcells to the grid. The algorithm distinguishes now between* `Leaf` *cells and* `Virtual` *subcells (blue). Solving Riemann problems that involves the latter requires a look up of a coarse grid* `Leaf` *cell parent (green). Global time stepping is employed, i.e. the same fixed time step size ΔT is used on all mesh levels.*

```
1  T ← 0
2  ΔT ← ...
3  INITIALISEADERDG()                          # only leaf cells impose initial conditions
4
5  while T < T_final do
6    for level l = 0, 1, ..., l_max do
7      for K ∈ T_l do
8        if type_K ∈ {Leaf} do                 # only leaf cells perform prediction
9          ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K)
10         q_h(·, T+ΔT)|_K ← q_h(·,T)|_K
11       end if
12     end for
13   end for
14
15   for level l = 0, 1, ..., l_max do
```

```
16    for face−connected cells K_a, K_b ∈ T_l do                                  # Riemann solves
17      if type_{K_a} ∈ {Leaf} and type_{K_b} ∈ {Leaf} then
18        ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
19        ( q*_h|_{K_b}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_b}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
20        G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
21                                    q*_h|_{K_b}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
22        q_h(·, T + ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
23        q_h(·, T + ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
24      else if type_{K_a} ∈ {Leaf} and type_{K_b} ∈ {Virtual} then
25        ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
26        K_c ← ancestor_{K_b}                    # look up leaf parent and interpolate boundary data
27        ( q*_h|_{K_b}|_{∂K_a∩∂K_c}, F(q*_h)|_{K_b}|_{∂K_a∩∂K_c} ) ← extrapolate_{∂K_a∩∂K_c}(q*_h|_{K_c}, F(q*_h)|_{K_c})
28        G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
29                                    q*_h|_{K_b}|_{∂K_a∩∂K_b}, F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
30        q_h(·, T + ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
31        q_h(·, T + ΔT)|_{K_c} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
32      else if type_{K_a} ∈ {Virtual} and type_{K_b} ∈ {Leaf} then
33        # as above with reversed roles
34        # ...
35      else # do nothing
36      end if
37    end for
38  end for
39
40  for level l = 0, 1, ..., l_max do
41    for K ∈ T_l do
42      if type_K ∈ {Leaf} do                          # only leaf cells perform volume integral
43        q_h(·, T + ΔT)|_K+ = volumeIntegral(F(q*_h)|_K)
44      end if
45    end for
46  end for
47  T ← T + ΔT
48 end while
```

6.2.3 Building the Computational Mesh

ExaHyPE creates a regular base mesh as specified in the description of the user solver and adapts it according to the user solvers' mesh refinement criterion. Initially, this base mesh is completely populated with `Leaf` cells. A user solver can implement a refinement criterion specifying where the mesh should be further adapted. It is only evaluated in `Leaf` cells and gives one of three answers:

- `Coarsen`: The solution can be represented on a coarser mesh level.
- `Keep`: The solution is represented on the optimal mesh level.
- `Refine`: The solution must be represented on a finer mesh level.

While the time stepping works with just two states (`Leaf` and `Virtual`), the state space grows significantly when performing mesh refinement operations. These operations consist of multiple substeps that require a communication protocol between cells and their children. In the algorithms presented in this section, I realise such protocols via the `type` attribute instead of using multiple single attributes. This linearises the state space and, therefore, simplifies logic depending on the state of a cell.

Notation. *While the algorithms work with various states that the cell types* `Virtual`, `Leaf`, and `Parent` *can attain, I typically omit these states in the textual description of the algorithms in text body and algorithm captions.*

6.2.4 Refinement Marking

Peano runs through the adaptive spacetree in a level-wise depth-first order and then tracks back to the root cell [103]. The traversal automaton always pops 3^d (children) cells at once from the cell stack. All traversal events are issued for these cells before the automaton picks the next refined cell to descend into. The procedure is repeated after loading the 3^d children of this cell. A very rough approximation of this traversal order is a traversal that traverses all cells first in top-down and then in bottom-up direction. This approximation suffices to describe all of ExaHyPE's mesh refinement algorithms.

Algorithm 6.2 shows one iteration of the *marking* procedure in ExaHyPE. Typically, multiple of such iterations are run in a row. Algorithm 6.2 requires one top-down and one bottom-up traversal of the spacetree. Therefore, it can be mapped to a single Peano spacetree traversal. ExaHyPE regards the evaluation of the refinement criterion as expensive, it is thus only evaluated once and not every time the cell is accessed during the mesh adaptation phase. Furthermore, the algorithm prevents multi-level coarsening as the refinement criterion might indicate that an intermediate level is optimal to represent the solution. Therefore, coarsening is performed level by level. The refinement procedure, in contrast, can add multiple levels in one rush.

Refining a `Leaf` cell introduces further `Leaf` cells to the mesh while the original `Leaf` cell is transformed to a `Parent` cell. ExaHyPE uses a veto mechanism to determine which `Parent` cells can be coarsened again: During a top-down mesh traversal, the coarse grid `Parent` cell changes its state (`ParentCoarseningRequested`) indicating that it wants to coarsen all its children. The children are traversed next and reset this state if their refinement criterion does not return a coarsening request. In the following bottom-up traversal, the coarse grid `Parent` cell collects the result. If its request was accepted, it changes its state accordingly (`ParentCoarsening`). Otherwise, it changes its state such that it will not trigger the coarsening procedure again in the coming marking iterations (`ParentKeep`).

Algorithm 6.2 (Marking)**.** *One iteration of the refinement marking procedure. Child cells veto erasing events of their* ***`Parent`*** *if the refinement criterion returns a keep or refine request. After the iteration, information is available if a* ***`Leaf`*** *cell needs to be refined or a* ***`Parent`*** *cell coarsened. All subroutine calls with prefix* ***`Spacetime::`*** *refer to the mesh back end, i.e.* Peano*. A posteriori refinement adapts the mesh and then performs a rollback. Hence, the previous solution must be checked, too (blue).*

```
1  function MARKING(aposterioriRefinement)
2    # 1. Top down traversal (mark children)
3    for level l = 0, 1, ..., l_max do
4      for cell K ∈ T_l do
5        if type_K ∈ {Leaf} then
6          marker ← refinementCriterion(q_h(·, T + ΔT)|_K)
7          if aposterioriRefinement = true and marker_K ∈ {Coarsen} then
8            marker ← refinementCriterion(q_h(·, T)|_K)
9          end if
10         # refine / keep / coarsen
11         type_K ← LeafCoarsen
12         if marker ∈ {Refine} and l < l_max then
13           type_K ← LeafRefineInitiated
14           Spacetree::refineIfUnrefined(K)      # introduces Empty cells to the mesh
15         else if marker ∈ {Keep} then
16           type_K ← LeafKeep                    # do not evaluate refinement criterion again
17         end if
18         # support/veto coarsening
19         if l > 0 and type_K ∉ {LeafCoarsen} then
20           K_c ← Spacetree::getParent(K)
21           type_K_c ← ParentKeep
```

```
22            end if
23          else if type_K ∈ {Parent} then
24            type_K ← ParentCoarseningRequested
25            K_c ← Spacetree::getParent(K)
26            if type_{K_c} ∈ {ParentCoarseningRequested} then
27              type_{K_c} ← Parent                    #veto multi−level coarsening
28            end if
29          else if type_K ∈ {ParentKeep} then
30            K_c ← Spacetree::getParent(K)
31            if type_{K_c} ∈ {ParentCoarseningRequested} then
32              type_{K_c} ← ParentKeep                #veto coarsening
33            end if
34          end if
35        end for
36      end for
37      # 2. Bottom up traversal (revisit parents)
38      for level l = l_max, l_max − 1, ..., 0 do
39        for cell K ∈ T_l do
40          if type_K ∈ {ParentCoarseningRequested} then
41              type_K ← ParentCoarsening
42          end if
43        end for
44      end for
45    end function
```

6.2.5 Refining

The marking iterations (see Algorithm 6.2) request the creation of new `Empty` cells from Peano if a cell that is flagged for refinement has no `Virtual` children. ExaHyPE employs an iteration of Algorithm 6.3 to finalise refinement operations. In the top-down traversal, ExaHyPE interpolates the ADER-DG solution from the coarse grid `Leaf` cell to the new fine grid `Leaf` cells. In the bottom-up traversal, the original coarse grid `Leaf` cell changes its type to `Parent`. Marking a cell and refining it takes two mesh traversals in total.

Notation. *I explicitly write "spacetree refinement" whenever I discuss the refinement of the underlying spacetree. If I write "refinement" or "mesh refinement", I refer to Alg.* 6.3 *and the preceding refinement marking in Alg.* 6.2*, although mesh refinement often triggers spacetree refinement.*

Algorithm 6.3 (Refinement)**.** *Refinement procedure. If a cell was marked for refinement,*

new `Leaf` *cells are created and data is interpolated from the coarse grid cell to the fine grid children. A posteriori refinement adapts the mesh and then performs a rollback. Hence, the previous solution must be interpolated, too (blue).*

```
1  function REFINE(aposterioriRefinement)
2    # 1. Top down traversal (interpolate from coarse grid)
3    for level l = 0, 1, ..., l_max do
4      for cell K ∈ T_l do
5        if type_K ∈ {Empty, Virtual} then
6          assert: l > 0
7          K_c ← Spacetree::getParent(K)
8          if type_{K_c} ∈ {LeafRefineInitiated, LeafRefine} then
9            type_{K_c} ← LeafRefine
10           type_K ← Leaf
11           q_h(·, T + ΔT)|_K ← interpolate_{K⊂K_c}(q_h(·, T + ΔT)|_{K_c})
12           if aposterioriRefinement = true then
13             q_h(·, T)|_K ← interpolate_{K⊂K_c}(q_h(·, T)|_{K_c})
14           end if
15         end if
16     end for
17   end for
18   # 2. Bottom up traversal (deallocate data on refined cells )
19   for level l = l_max − 1, ..., 0 do
20     for cell K ∈ T_l do
21       if type_K ∈ {LeafRefine} then
22         type_K ← Parent
23       end if
24     end for
25   end for
26 end while
```

6.2.6 Coarsening

AMR adapts the mesh resolution around interesting solution features, e.g. shock waves. If a solution feature travels out of a certain area, the solution can be represented on a coarser mesh in this area. Such a coarse grid representation requires less DOFs and thus less memory. Moreover, less computations need to be performed. Therefore, dynamic AMR simulations coarsen refined cells–i.e. the `Parent` cells in ExaHyPE's host mesh–whereever possible. Coarsening has to be performed with care due to the following observations; e.g. [96]:

- Coarsening of cells may result in a loss of information.
- While adapting the mesh, the refinement criterion might flag a cell for refinement directly after the same cell was coarsened.

The first point is especially a problem when goal-oriented refinement criteria are considered. Here, it might not be possible to recover the lost information after the coarsening. The second observed behaviour is sometimes caused by gradient-based refinement criteria where a cell is refined because the gradient is measured too small in the fine grid cells and too large in the coarse grid cell. Without countermeasures, the cell is then refined and coarsened over and over again; e.g. [96].

ExaHyPE's coarsening procedure (Algorithm 6.4) anticipates both issues and, therefore, requires two mesh traversals. In the first traversal's top-down part, `Leaf` cells whose `Parent` has been marked for coarsening project their ADER-DG solution onto the `Parent`'s cell. They do not deallocate the solution yet! In the traversal's bottom-up part, the `Parent` evaluates the refinement criterion on this projected solution. If the criterion returns a refinement request, the coarsening procedure is stopped. Otherwise, the `Parent` cell becomes a `Leaf` cell in the next traversal's top-down part. Furthermore, the fine grid `Leaf` cells become `Empty` cells and deallocate their solution. `Empty` cells are cleaned up by a later mesh refinement iteration.

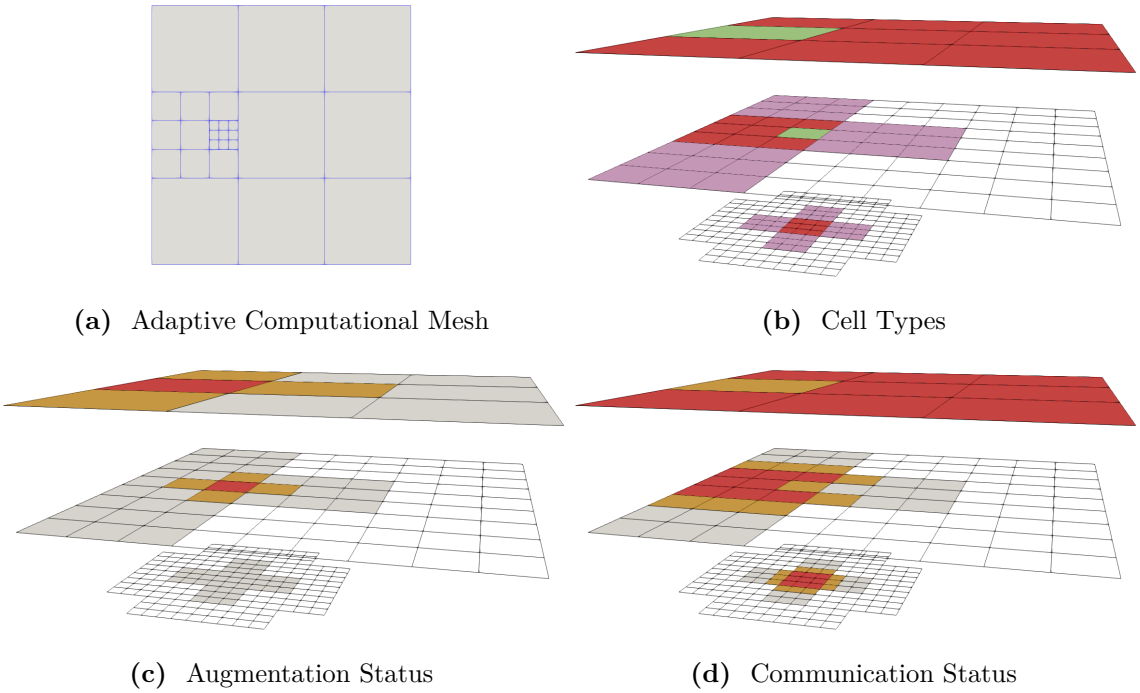
Algorithm 6.4 (Safe Coarsening)**.** *Coarsening procedure. If none of the fine grid children veto the coarsening request of their* `Parent`*, they restrict their solution up to the* `Parent` *in the next traversal. To not lose fine grid information and to prevent refine-coarsen oscillations, the refinement criterion is evaluated again on the restricted solution. Only if it does not indicate refining, the coarsening procedure is finished. All subroutine calls with prefix* `Spacetime::` *refer to the mesh back end, i.e.* Peano*. A posteriori refinement adapts the mesh and then performs a rollback. Hence, the previous solution must be checked, too (blue).*

```
1  function coarsenSafely(aposterioriRefinement)
2    # 1. Top down traversal (project on coarse grid)
3    for level l = 0, 1, ..., l_max do
4      for cell K ∈ T_l do
5        if l > 0 and type_K ∈ {LeafCoarsen} then
6          K_c ← Spacetree::getParent(K)
7          if type_{K_c} ∈ {ParentCoarsening} then
8            q_h(·, T + ΔT)|_{K_c} ← project_{K_c}(q_h(·, T + ΔT)|_K)
```

```
9             if aposterioriRefinement = true then
10              q_h(·,T)|_{K_c} ← project_{K_c}(q_h(·,T)|_K)
11            end if
12          end if
13        end for
14      end for
15      # 2. Bottom up traversal (revisit parent cells)
16      for level l = l_max − 1, ..., 0 do
17        for cell K ∈ T_l do
18          if type_K ∈ {ParentCoarsening} then
19            tempMarker ← refinementCriterion(q_h(·,T + ΔT)_K)
20            if aposterioriRefinement = true and tempMarker ≠ Refine then
21              tempMarker ← refinementCriterion(q_h(·,T)_K)
22            end if
23            if tempMarker ∈ {Refine} then
24              type_K ← ParentKeep
25            else
26              type_K ← Leaf
27            end if
28        end for
29      end for
30      # 3. Top down traversal (erase fine grid cells)
31      for level l = 0, 1, ..., l_max do
32        for cell K ∈ T_l do
33          if type_K ∈ {Leaf} and l > 0 then
34            K_c ← Spacetree::getParent(K)
35            if type_{K_c} ∈ {Leaf} then
36              type_K ← Empty
37            end if
38        end for
39      end for
40    end functions
```

6.2.7 Building the Virtual Mesh

In this subsection, I present an algorithm to build up layers of `Virtual` cells around the adaptively refined cells, i.e. cells of type `Parent`. Inspired by the limiting ADER-DG method's limiter status diffusion, I propose to use a similar mechanism to flag the `Leaf` children need to be introduced. I further add a second integer status to determine on which `Virtual` cell to interpolate boundary data from a coarse grid `Leaf` cell. I call the first "augmentation status",



(a) Adaptive Computational Mesh

(b) Cell Types

(c) Augmentation Status

(d) Communication Status

Fig. 6.4: (a) An arbitrarily adaptive computational mesh. (b) The host mesh contains `Leaf` cells (red), `Parent` cells (green), and `Virtual` cells (magenta). Transparent cells are `Empty` spacetree cells due to pre-refinement by Peano. (c) The augmentation status spreads from the `Parent` cells. (d) The communication status spreads from the `Leaf` cells.

and the second "communication status". The propagation of the integer flags is inspired by heat diffusion. To diffuse the augmentation status, I use `Parent` cells as heat source (Fig. 6.4 (c)). For the communication status, I use `Leaf` cells as heat source (Fig. 6.4 (d)). Furthermore, I store augmentation and communication status of a cell's face neighbours. The information in which direction a `Leaf` cell borders a `Virtual` cell allows to only interpolate boundary data for specific interfaces.

The condition

$$\texttt{type}_{K_a} \in \{\texttt{Leaf}\} \quad \textbf{and} \quad \texttt{type}_{K_b} \in \{\texttt{Virtual}\}$$

in algorithm Algorithm 6.1 can then be brought into the following form:

$$\begin{aligned} \texttt{communicationStatus}_{K_a} &= \texttt{MaxCommunicationStatus} \quad \textbf{and} \\ \texttt{neighbourAugmentationStatus}_{K_a}[\partial K_a \cap \partial K_b] &< \texttt{MaxAugmentationStatus} \quad \textbf{and} \\ \texttt{neighbourCommunicationStatus}_{K_a}[\partial K_a \cap \partial K_b] &\geq \texttt{MaxCommunicationStatus} - 1. \end{aligned}$$

Modelled as integers, the maximum communication and augmentation status can be tuned. For example, increasing the maximum augmentation status introduces more `Virtual` cells into the mesh. The spacetree can be pre-refined this way.

Algorithm 6.5 (Virtual Refinement)**.** *Diffusion of augmentation and communication status and* `Virtual` *refinement. Any state of a* `Leaf` *cell serves as heat source for the communication status diffusion. Any state of a* `Parent` *cell serves as heat source for the augmentation status. Leaf cells may only introduce* `Virtual` *children if they are in the* `LeafKeep` *or* `LeafCoarsen` *state. All subroutine calls with prefix* `Spacetime::` *refer to the mesh back end, i.e.* Peano.

```
1  function virtualRefinement()
2    # Top−down traversal
3    for level l = 0, 1, ..., l_max do
4      for face−connected cells K_a, K_b ∈ T_l: type_{K_a}, type_{K_b} ∉ {Empty} do # loop over faces
5        neighbourAugmentationStatus_{K_a}[∂K_a ∩ ∂K_b] ← augmentationStatus_{K_b}
6        neighbourAugmentationStatus_{K_b}[∂K_a ∩ ∂K_b] ← augmentationStatus_{K_a}
7        neighbourCommunicationStatus_{K_a}[∂K_a ∩ ∂K_b] ← communicationStatus_{K_b}
8        neighbourCommunicationStatus_{K_b}[∂K_a ∩ ∂K_b] ← communicationStatus_{K_a}
9      end for
10     for cell K ∈ T_l do
11       # merge communication status
12       if type_K ∈ {Leaf, LeafKeep, LeafCoarsen, ...} then
13         communicationStatus_K ← MaxCommunicationStatus
14       else if type_K ∉ {Empty} then
15         communicationStatus_K ← max(neighbourCommunicationStatus) − 1
16       end if
17       # merge augmentation status
18       if type_K ∈ {Parent, ParentKeep, ...} then
19         augmentationStatus_K ← MaxAugmentationStatus
20       else if type_K ∉ {Empty} then
21         augmentationStatus_K ← max(neighbourAugmentationStatus) − 1
22       end if
23        # create virtual cells & link to leaf cell / erase virtual cells
24       if l > 0 and type_K ∈ {Empty, Virtual} then
25         K_c ← Spacetree::getParent(K)
26         if type_{K_c} ∈ {Virtual, LeafKeep, LeafCoarsen} and
27            augmentationStatus_{K_c} > 0 then
28           type_K ← Virtual
29           # link to leaf cell
30           if type_{K_c} ∈ {LeafKeep, LeafCoarsen} then
31             ancestor_K ← K_c
```

```
32            else if
33              ancestor_K ← ancestor_{K_c}
34            end if
35          end if
36          # erase virtual cells
37          else if type_{K_c} ∈ {Empty} or augmentationStatus_{K_c} = 0 then
38            type_K ← Empty
39          end if
40        end if
41        # trigger spacetree refinement
42        if type_K ∈ {Virtual, LeafKeep, LeafCoarsen} and augmentationStatus_K > 0 then
43          Spacetree::refineIfUnrefined(K)
44        end if
45      end for
46    end for
47  end while
```

6.2.8 Refining and Coarsening at Master-Worker Boundaries

Peano ensures that it's tree decomposition always preserves a logical master-worker tree topology on the ranks (see Fig. 6.3) and adopts its mesh such that user (i.e. ExaHyPE's) mesh requirements are met at least. It might decide to use finer meshes than required. ExaHyPE can in general not steer when and where Peano may cut off a subtree from a spacetree partition that is deployed to another worker rank. Therefore, ExaHyPE's mesh refinement operations have to handle multiple scenarios that arise due to the spacetree partitioning:

- Refinement operations may not have completed at the time a subtree is deployed to a worker rank. They need to be continued afterwards.
- Peano might deploy subtrees that consist only of `Virtual` or `Empty` cells. ExaHyPE's mesh refinement procedure might later on decide to replace them with `Parent` and `Leaf` cells.
- `Leaf` and `Parent` cell might get coarsened at a master-worker boundary.

To handle such situations, ExaHyPE introduces master-worker communication into its refinement and coarsening protocols. Peano's spacetree traversal triggers communication events that ExaHyPE subscribes to. The master's fine grid spacetree cell is placed at the master-worker boundary, its coarse grid spacetree cell is placed one level above; see Fig. 6.1

(d). In the top-down part of every mesh refinement iteration, ExaHyPE plugs into one of Peano's master-to-worker exchange callbacks to send the master's coarse and fine grid spacetree cell state to the worker. During ExaHyPE's mesh refinement iterations, workers are thus aware of the states of the master's coarse grid and fine grid spacetree cell.

The master takes over control during mesh refinement operations. If the master introduces a new fine grid `Leaf` cell, it sends the cell's state and solution to the worker. The worker, which is aware of the master's fine grid cell's state, replaces its `Empty` or `Virtual` cell with the received `Leaf` cell and allocates memory to store the interpolated solution.

If the master's coarse grid `Parent` cell attempts to coarsen its children, it obtains the worker's `Leaf` cell's state back in the bottom-up part of a mesh traversal. The state `LeafCoarsen` indicates that the worker's `Leaf` cell does not veto the coarsening procedure. If the worker's `Leaf` cell, or any of the other `Leaf` cells on the master, do not veto the coarsening request, the coarsening procedure is started for the master's coarse grid `Parent` cell. In the top-down part of the next mesh traversal, the worker receives the updated state of the master's coarse grid cell. In the bottom-up part of the same traversal, the worker represents the `Leaf` cell's solution on the geometry of the master's coarse grid cell and sends the result of this projection up to the master.

6.3 Eliminating Local Master-Worker Communication

Coarser `Leaf` cells communicate with finer `Leaf` cells at mesh resolution transitions. Peano's spacetree partitioning might introduce a master-worker boundary right at a resolution transition. ExaHyPE uses the `Virtual` cells/markers in its mesh meta data structure to identify where such a partitioning has occurred. It realises the communication between adjacent coarse and fine grid `Leaf` cells via `Virtual` cells. `Virtual` cells that neighbour a fine grid `Leaf` cell interpolate boundary data to the fine grid and send face integral contributions to the coarse grid. If the spacetree partitioning cuts the spacetree vertically above such a `Virtual` cell, this data flow is interrupted. It must then be reestablished via local master-worker communication!

In ExaHyPE, I circumvent such local master-worker communication by artificial mesh refinement, i.e. ExaHyPE's mesh refinement introduces `Leaf` cells wherever local master-worker communication is required. This is done via the augmentation status: The master process checks if the worker's cell is a `Virtual` cell and if its augmentation status is one less

than the maximum. In this case, the `Virtual` cell is next to a `Leaf` cell. Then, the master looks up the next `Leaf` cell on coarser level of its spacetree partition and marks it for refining. The mesh refinement algorithm picks this refinement request up and refines the marked `Leaf` cell.

Note that the top-most cell of the master's spacetree could be a `Virtual` cell, too. However, due to the construction procedure of the `Virtual` mesh, this cell has also an augmentation status one less than the maximum. Therefore, the above procedure becomes active here, too.

`Virtual` cells do not only reveal information where to refine but also where not to erase cells: Assume the worker cell is a `Leaf` cell and its `Parent` cell belongs to the master. Furthermore, assume the worker cell has an augmentation status one less than the maximum. Then, this implies that some of its descendants are adjacent to a `Leaf` cell. In this case, the worker's `Leaf` cell must not be erased i.e. coarsening attempts initiated by the `Parent` cell that resides on the master process must be vetoed. The grid coarsening is constrained by the communication pattern. If in doubt, ExaHyPE works with a too fine mesh rather than a too adaptive mesh.

6.4 Pre-Refinement Techniques

This section presents two novel mesh refinement techniques I introduced into ExaHyPE. I originally implemented them to reuse the limiter indicators of the a posteriori subcell limiting ADER-DG method as mesh refinement indicators. If an indicator becomes active in a cell, i.e. the cell becomes troubled, a rollback and FVM recomputation is required as the ADER-DG solution is troubled in this case. Therefore, refinement according to the limiter indicators is not straightforward and requires a rollback, too. I call the resulting mesh adaption procedure *a posteriori refinement*. Furthermore, ExaHyPE's implementation of AMR for the a posteriori limiting ADER-DG method requires that a number of neighbour cells around the troubled cells is refined, too. For this purpose, I introduce a technique I name *halo refinement*. In this section, I generalise both refinement techniques and apply them to classic dynamic AMR for the sole ADER-DG method.

A Posteriori Refinement

If a cell on a coarser tree level requests mesh refinement, it is often already too late: Fine grid information may have been destroyed or spurious oscillations have been introduced because the solution representation has been chosen too coarse. Inspired by the recomputation procedure

of the a posteriori limiting ADER-DG method, I propose to use a similar rollback mechanism for classic dynamic AMR. The idea is to adapt the mesh according to the current solution, and then rollback to the previous solution state. Afterwards, the time step is rerun with a mesh that is refined where the solution will be.

In the previously presented mesh adaptation algorithms, the blue highlighted lines indicate where the algorithms must be changed to support a posteriori refinement. The marking procedure (Algorithm 6.2) must ensure that `Parent` cells are not coarsened where their fine grid children's previous solution will be required after the rollback. The coarsening procedure (Algorithm 6.4), must check if a cell that gets coarsened in the current time step would have also get coarsened in the previous time step.

Halo Refinement

Halo refinement allows to introduce additional `Leaf` cells around `Leaf` cells that have been refined to resolve interesting solution features, e.g. to prepare the mesh to host a propagating wave. ExaHyPE realises halo refinement by diffusing another integer status, the halo status. Whenever the refinement criterion is evaluated on a `Leaf` cell and returns `Keep`, this implies that the `Leaf` cell resolves an interesting feature on the current mesh level. The same is the case if the criterion returns `Refine` on the maximum user-prescribed level. ExaHyPE assigns a `Leaf` cell in both cases the maximum halo status. The halo status is then diffused just like the limiter, augmentation, and communication status. The maximum halo status can be tuned to add an arbitrary number of halo cells around interesting solution features.

Implementing halo refinement on top of ExaHyPE's mesh meta data structure is straightforward: If a `Virtual` cell carries a halo status greater than zero, it notifies its `Leaf` ancestor to refine (Fig. 6.5 (top left)). This is done repeatedly until the `Virtual` cell is replaced by a `Leaf` cell (Fig. 6.5 (top right)). The halo refinement procedure terminates if no `Virtual` cell carries a halo status greater zero anymore. I want to emphasise that halo refinement can also be realised without `Virtual` cells. The key ingredient is that a coarse grid `Leaf` cell knows if its fine grid `Leaf` cell neighbours want to keep their solution representation.

Combining Halo and A Posteriori Refinement

Halo refinement refines the mesh around interesting features. As a tracked solution feature will likely propagate into one of the adjacent cells, halo refinement can be classified and used

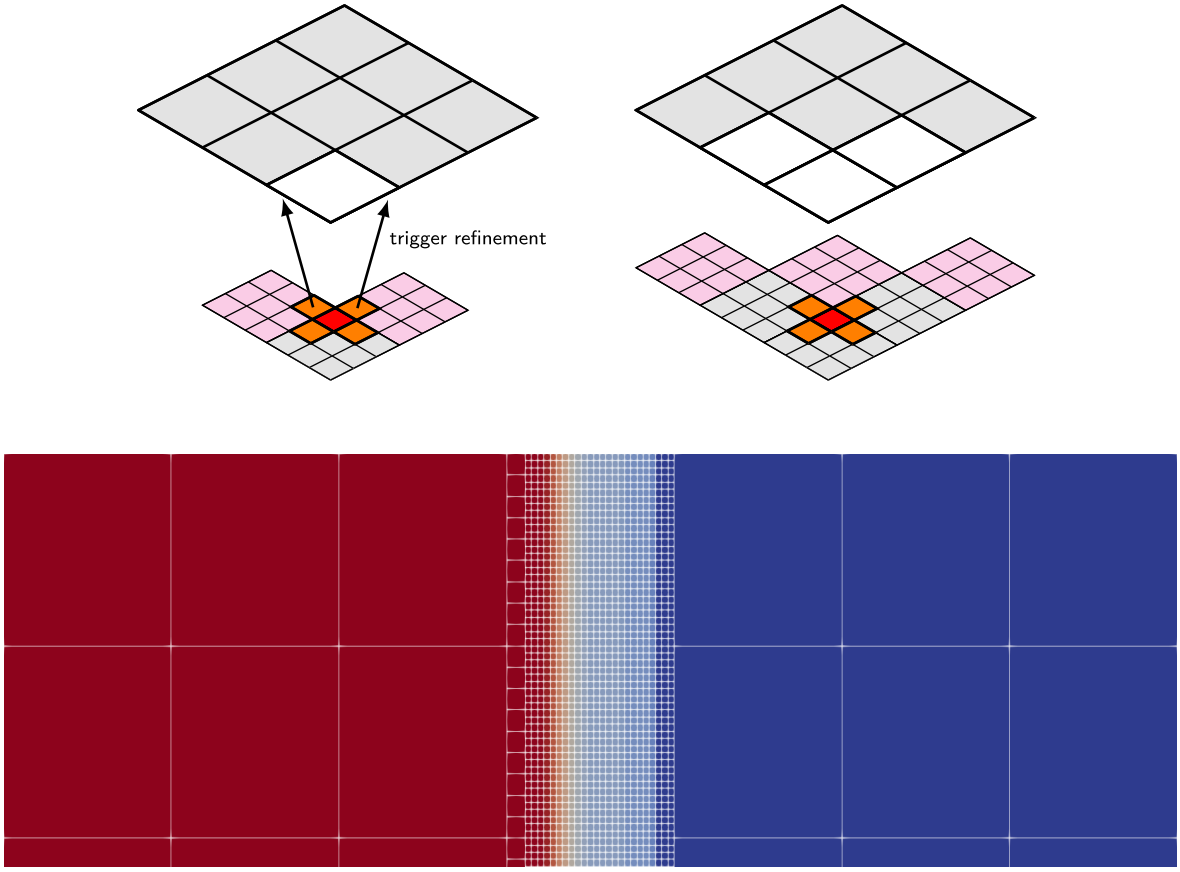


Fig. 6.5: Top left: A virtual cell (magenta) carries a halo status greater zero. This triggers refinement of its `Leaf` cell ancestor. `Leaf` cells are filled grey and `Virtual` cells are filled white. Top right: The mesh is refined such that no virtual cell carries a halo status greater zero. Bottom: Aggressively refined mesh that employs halo refinement along a shock front.

as a very crude pre-refinement procedure. A posteriori refinement, on the other hand, is well-suited to deal with situations where an interesting feature suddenly emerges somewhere in the mesh. The classical example is wave steepening and shock formation as they are observed with nonlinear hyperbolic PDE systems. If both techniques are used together, it is important to keep a record of the old halo status from before the mesh adaptation. Coarsening a `Parent` cell must only be allowed if its `Leaf` cell children are not flagged for halo refinement in the current and previous time step.

6.5 Inter-Grid Transfer Operators

Inter-grid transfer operators are necessary to compute the DG face integral at mesh resolution transitions. In the context of dynamic AMR, inter-grid transfer operators are required to represent the solution on finer or coarser mesh levels if a cell is refined or an ensemble of 3^d cells is coarsened into one cell, respectively. In this section, I derive all required operators for arbitrary mesh resolution jumps and demonstrate that they can be constructed from 1D single-level operators that can be pre-computed for a given polynomial order p.

6.5.1 Preliminaries

ExaHyPE's adaptive meshes consist of cubes (or squares in 2D). Each cell K can be represented as an affine mapping of the unit cube. The Jacobian determinant of the cells is space independent and equals their volume. A mapping for each face of a cell can be derived from the cell mapping by fixing a single one of the mapping parameters to 0 or 1. ExaHyPE's ADER-DG solver employs a nodal DG spatial discretisation [64]. The DG solution is discontinuous at the cell interfaces but smooth within each cell, where the solution is approximated as polynomial of order p. Per cell, basis functions of trial and test space are constructed as tensor products of $p+1$ Lagrange basis functions whose support points coincide with the quadrature nodes of a Gauss Legendre quadrature. Basis functions with support in cell K are defined such that they evaluate to the same value at point $x \in K$ as their reference cell equivalent at the reference coordinate $\hat{x} \in [0,1]^d$ that maps to $x \in K$. As the nodal basis functions are constructed from Lagrange polynomials, it holds that

$$\varphi_i(x_j) = \hat{\varphi}_i(\hat{x}_j) = \delta_{ij},$$

where φ_i and $\hat{\varphi}_i$, $0 \leq i \leq p$, denote basis and reference basis functions, respectively. Their respective support points are denoted by x_j and $\hat{x}_j$, $0 \leq j \leq p$. The Kronecker delta δ_{ij} is

defined as follows:

$$\delta_{ij} = \begin{cases} 1 & i = j, \\ 0 & \text{else.} \end{cases}$$

Let f be a polynomial of order p with support in cell K. The L^2 product between the basis function and f can be exactly integrated with a Gauss-Legendre rule with $p+1$ quadrature nodes [13][19],

$$\begin{aligned} (f, \varphi_i)_{L^2(K)} &= \int_K f\,\varphi_i \,\mathrm{d}x = |K| \sum_{j=0}^{p} w_j\,\varphi_i(x_j)\,f(x_j) = |K| \sum_{j=0}^{p} w_j\,f(x_j)\,\delta_{ij} \\ &= |K|\,w_i\,f(x_i), \qquad i \in \{0, 1, \ldots, p\}. \end{aligned} \tag{6.2}$$

If f is one of the basis functions itself, it holds that

$$(\varphi_l, \varphi_i)_{L^2(K)} = |K|\,w_i\,\delta_{il}, \qquad i, l \in \{0, 1, \ldots, p\}. \tag{6.3}$$

Remark on Gauss-Lobatto Quadrature and Support Points

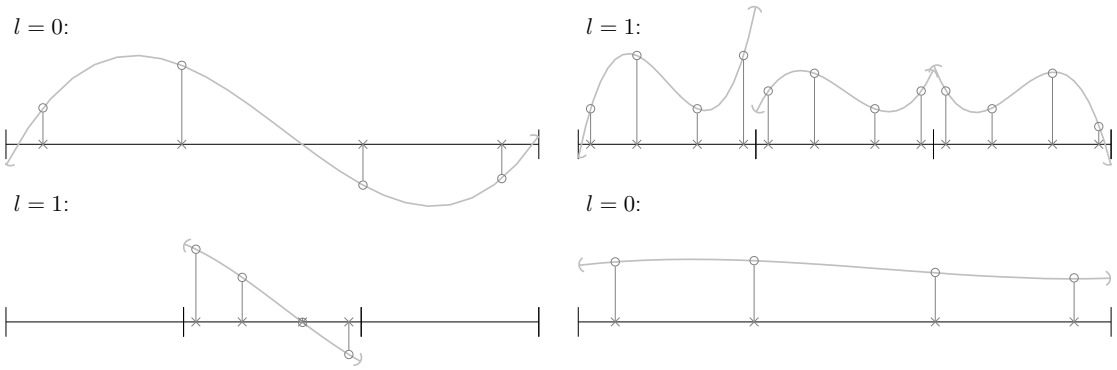
Note that the continuous and numerical integral in (6.2) and (6.3) are not identical if a Gauss-Lobatto quadrature is used as this quadrature only integrates polynomials with maximum degree $2\,(p+1) - 3 = 2\,p - 1$ exactly. The product of two basis functions has the maximum degree $2\,p$.

In the remainder of this section, I assume that Gauss-Legendre support points are used in combination with a Gauss-Legendre quadrature rule. I emphasise that all derived operators are exact in this case. Note that if Gauss-Lobatto nodes are used as support points for the basis functions in combination with a Gauss-Lobatto quadrature rule, these operators are only approximations.

6.5.2 Interpolation

Both the refinement of a cell and the representation of coarse grid DOFs on the fine grid before the face integral require that coarse grid DOFs are interpolated to a subset of the respective grid entity. In ExaHyPE, coarse grid DOFs associated with a cell volume or face can always be exactly represented on finer grids as the polynomial order p is not adapted (Fig. 6.6 (a)).

An interpolation operator represents coarse grid DOFs in terms of fine grid DOFs on a subset of the coarse grid entity, which may be a face or cell. For nodal DG, it can be derived by



(a) Coarse-to-Fine Interpolation **(b)** Fine-to-Coarse Projection

Fig. 6.6: Spatial coarse-to-fine interpolation (a) and fine-to-coarse projection (b) visualised using third order polynomials (nodal Gauss-Legendre basis). The interpolation samples the coarse grid polynomial to represent it on the fine grid (level 1). The fine-to-coarse projection operation projects multiple fine grid polynomials onto a single polynomial on the coarse grid (level 0).

evaluating the coarse grid representation at the support points of the fine grid DOFs (see also Fig. 6.6 (a)),

$$Q_i^{\text{fine}} = \sum_j Q_j^{\text{coarse}} \hat{P}_{ij}^{\Delta l}. \tag{6.4}$$

where the $i = (i_\xi)_{\xi=0,1,\ldots,\xi_{\max}}$ and $j = (j_\xi)_{\xi=0,1,\ldots,\xi_{\max}}$ index the fine and coarse grid DOFs, respectively. Their components run from 0 to p. If volume data is interpolated, $\xi_{\max} = d$. In case of boundary data, $\xi_{\max} = d - 1$. Equation (6.4) naturally introduces the interpolation operator $\hat{P}^{\Delta l}$, which depends on the level difference Δl between fine and coarse grid,

$$\hat{P}_{ij}^{\Delta l} = \varphi_j^{\text{coarse}}(x_i^{\text{fine}}) = \hat{\varphi}_j^{\text{coarse}}(\hat{x}_i^{\text{fine}}), \tag{6.5}$$

where $\varphi_j^{\text{coarse}}$ denotes a basis function associated with the coarse grid entity, and $\hat{\varphi}_j^{\text{coarse}}$ denotes its reference domain representation. Furthermore, x_i^{fine} and $\hat{x}_i^{\text{fine}}$ denote a support point on the fine grid and the corresponding reference coordinate. Note that $\hat{x}_i^{\text{fine}}$ must be expressed with respect to the coarse grid entity's reference domain.

The interpolation operator is identical to the L^2 projector that projects the coarse grid DOFs

onto the fine grid basis, i.e.,

$$\begin{aligned}\hat{P}_{ij}^{\Delta l} &= \frac{(\varphi_j^{\text{coarse}}, \varphi_i^{\text{fine}})_{L^2(E)}}{(\varphi_i^{\text{fine}}, \varphi_i^{\text{fine}})_{L^2(E)}} = \frac{|E| \sum_l w_l \, \hat{\varphi}_j^{\text{coarse}}(\hat{x}_l^{\text{fine}}) \, \hat{\varphi}_i^{\text{fine}}(\hat{x}_l^{\text{fine}})}{|E| \sum_l w_l \, \hat{\varphi}_i^{\text{fine}}(\hat{x}_l^{\text{fine}}) \, \hat{\varphi}_i^{\text{fine}}(\hat{x}_l^{\text{fine}})} \\ &= \frac{\sum_l w_l \, \hat{\varphi}_j^{\text{coarse}}(\hat{x}_l^{\text{fine}}) \, \delta_{il}}{\sum_l w_l \, \delta_{il} \, \delta_{il}} = \frac{w_i \, \hat{\varphi}_j^{\text{coarse}}(\hat{x}_i^{\text{fine}})}{w_i} \\ &= \hat{\varphi}_j^{\text{coarse}}(\hat{x}_i^{\text{fine}}) = \varphi_j^{\text{coarse}}(x_i^{\text{fine}})\end{aligned} \tag{6.6}$$

where for an interpolation of volume DOFs, $E = K_{\text{coarse}} \cap K_{\text{fine}}$, and for boundary DOFs, $E = \partial K_{\text{coarse}} \cap \partial K_{\text{fine}}$. In (6.6), I use the fact that the L^2 product can be exactly integrated with a Gauss-Legendre quadrature rule that uses $p+1$ quadrature nodes per reference coordinate direction. Due to the tensor product structure of the basis functions, interpolation operator $\hat{P}^{\Delta l}$ has tensor product structure, too. Furthermore, multi-level interpolation operators $\hat{P}^{\Delta l}$ can be constructed by recursively applying a single-level operator (Fig. 6.7).

In summary: Interpolation of coarse grid degrees of freedom to arbitrary fine grid levels can be conducted by recursively applying tensor products of 1D single-level interpolation operators. The single-level operators are independent of a cell's geometry and can be pre-computed.

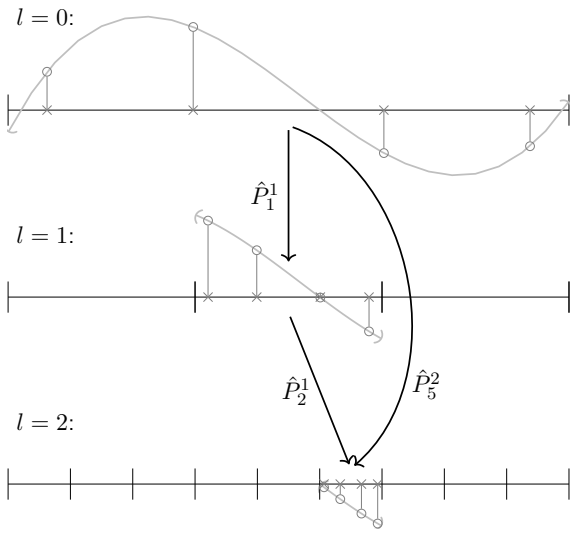


Fig. 6.7: If the level difference is greater than one, the interpolation operator can be constructed by chaining single-level operators $\hat{P}^1$. The subscript of the interpolation operators indicates the subinterval from the perspective of the respective coarse grid.

6.5.3 Face Integral at Mesh Resolution Transitions

The same 1D reference operators can be reused to add fine grid face integral contributions to a coarse grid cell's solution. Let K_a be a mesh cell, and $\mathcal{V}(K) = \{K' \in \mathcal{T} \colon \partial K' \cap \partial K' \neq \emptyset\}$

be the set of its direct neighbours. The surface integral can be decomposed into contributions from the interfaces that a cell shares with its direct neighbours:

$$\begin{aligned}&\int_T^{T+\Delta T} \int_{\partial K_a} v_h \, G(q_h^{*,-}, q_h^{*,+}) \, n \, \mathrm{d}s(x) \, \mathrm{d}t \\ &= \sum_{K_b \in \mathcal{V}(K)} \int_T^{T+\Delta T} \int_{\partial K_a \cap \partial K_b} v_h \, G(q_h^{*,-}, q_h^{*,+}) \, n \, \mathrm{d}s(x) \, \mathrm{d}t,\end{aligned} \tag{6.7}$$

where the Riemann solution is denoted by $G(q_h^{*,-}, q_h^{*,+}) \cdot n$ and the spatially varying DG test function by v_h. The latter is drawn from the same function space as the ADER-DG solution. The Riemann problem between cells is then solved as in Algorithm 6.6. Here, the `extrapolate` substep involves the coarse-to-fine interpolation discussed in the previous section. After the Riemann solve, the `faceIntegral` substep adds the result of the DG face integral to the coarse grid solution.

ExaHyPE's ADER-DG method extrapolates the space-time predictor and the normal component of the volume flux onto the faces before solving the Riemann problem. The implementation represents these boundary-extrapolated quantities using a Lagrange basis of the same order p that is used to represent the DG solution. In contrast to the DG solution, there are only $m\,(p+1)^{d-1}$ spatial DOFs per face. However, there is one such array of spatial DOFs for each of the $(p+1)$ time integration points. This makes $m\,(p+1)^d$ DOFs in total per face and per extrapolated flux and predictor. The following paragraphs focus on interpolation and projections of spatial DOFs from one mesh resolution to the other. The concepts are applicable to any DG method not only ADER-DG. Therefore, I do not discuss the time dependence of ADER-DG boundary data further in the remainder of this section.

Algorithm 6.6 (The DG Face Integral)**.** *The* DG *face integral between cells, which may be on different levels, can be decomposed into the substeps* ***`extrapolate`****,* ***`Riemann`****, and* ***`faceIntegral`****.*

```
1  for face−connected cells K_a, K_b ∈ 𝒯 do                          # Riemann solves
2    ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
3    ( q*_h|_{K_b}|_{∂K_b∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_b∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
4    G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
5                                q*_h|_{K_b}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
```

```
    q_h(·, T + ΔT)|_{K_a} += faceIntegral(G(q_h^{*,+}, q_h^{*,−}) (n · n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
    q_h(·, T + ΔT)|_{K_b} += faceIntegral(G(q_h^{*,+}, q_h^{*,−}) (n · n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
end for
```

At the interface of a coarse and fine grid cell, the Riemann solution is always expressed in terms of the fine grid DOFs. Therefore, it is straightforward to derive the contributions of the face integral to the fine grid cell's solution. However, the face integral contributions to the coarse grid cell's solution must be expressed in terms of the coarse grid DOFs. This requires applying another inter-grid transfer operator. Riemann solution and boundary-extrapolated coarse grid test function can be expanded as follows:

$$G(q_h^{*,-}, q_h^{*,+})\, n = \sum_j G_j\, \varphi_j^{\text{fine}}, \qquad v_h|_{\partial K_a \partial K_b} = \sum_{k,i} v_{k,i}\, \hat{F}_k\, \varphi_i^{\text{coarse}},$$

where I used that one of the reference coordinates remains constant on interface $\partial K_a \cap \partial K_b$. I number the respective one-dimensional basis functions with index k. The coefficients $\hat{F}_k$ are evaluations of these basis functions on the interface. Expanding equation (6.7) then leads to:

$$\begin{aligned}
&\int_{\partial K_{\text{coarse}} \cap \partial K_{\text{fine}}} G(q_h^{*,-}, q_h^{*,+}) : (v_h \otimes n)\, \mathrm{d}s(x) \\
&\qquad = (\sum_j G_j\, \varphi_j^{\text{fine}}, \sum_{k,i} v_{k,i}\, \hat{F}_k\, \varphi_i^{\text{coarse}})_{L^2(\partial K_{\text{coarse}} \cap \partial K_{\text{fine}})} \\
&\qquad = \sum_{k,i} v_{k,i}\, \hat{F}_k \sum_j G_j\, (\varphi_j^{\text{fine}},\, \varphi_i^{\text{coarse}})_{L^2(\partial K_{\text{coarse}} \cap \partial K_{\text{fine}})} \\
&\qquad = \sum_{k,i} v_{k,i}\, \hat{F}_k\, |\partial K_{\text{coarse}} \cap \partial K_{\text{fine}}| \sum_j G_j\, w_j\, \varphi_i^{\text{coarse}}(x_j^{\text{fine}}) \\
&\qquad = |\partial K_{\text{coarse}} \cap \partial K_{\text{fine}}| \sum_{k,i} v_{k,i}\, \hat{F}_k \sum_j G_j\, w_j\, \hat{P}_{ji}^{\Delta l}
\end{aligned} \tag{6.8}$$

where I have substituted the interpolation operator (6.5) in the last step. Note the transposed indices. Equation (6.8) reveals that conducting face integrals between cells at mesh resolution transitions requires to apply transposed interpolation operator to the fine grid degrees of freedom. Therefore, the DG face integral at arbitrary mesh resolution transitions can also be computed via the 1D single-level interpolation operator (6.5).

The face integral performs an (non-normalised) L^2 projection of fine grid boundary DOFs onto coarse grid boundary DOFs. This fine-to-coarse L^2 projection might remove information from the find grid representation as it smooths it out (Fig. 6.6 (b)).

6.5.4 Coarsening

In ExaHyPE, the fine-to-coarse projection of the ADER-DG solution is defined as L^2 projection. The solution of coarse grid leaf cell K_c is constructed as follows from the fine grid DOFs:

$$Q_i^{K_c} = \sum_{K_f \in \mathcal{T}:\ K_f \subset K_c} \sum_j Q_j^{K_f} \frac{(\varphi_i^{K_c}, \varphi_j^{K_f})_{L^2(K_c \cap K_f)}}{(\varphi_i^{K_c}, \varphi_i^{K_c})_{L^2(K_c)}},$$

where i and j are d-dimensional multi indices for the basis function and their support points. Furthermore,

$$\frac{(\varphi_i^{K_c}, \varphi_j^{K_f})_{L^2(K_c \cap K_f)}}{(\varphi_i^{K_c}, \varphi_i^{K_c})_{L^2(K_c)}} = \frac{|K_f|}{|K_c|} \frac{w_j\, \hat{\varphi}_i^{K_c}(\hat{x}_j^{K_f})}{w_i} = \frac{1}{3^d} P_{ji}^1.$$

Again, the physical extents of the cell do not play a role in the projection, and the projector can be constructed from the single-level 1D reference domain interpolation operators (see (6.5)).

6.6 Discussion

In this chapter, I presented ExaHyPE's dynamic AMR machinery. ExaHyPE uses the adaptive mesh that it builds from Peano's spacetree as meta data structure. This has two implications: First, Peano allows to position ExaHyPE's box-like geometries within a bounding box, where the bounding box governs the mesh resolution. ExaHyPE uses this flexibility to realise arbitrary regular base meshes for its solvers. Second, to handle the DG methods data flow between coarse and fine grid cells at arbitrary adaptivity boundaries, ExaHyPE places virtual helper cells into the spacetree.

The proposed mesh data structure can be generalised to other applications and other features can be built on top of it. Peano's spacetree partitioning introduces a master-worker relationship between processes that might require master-worker communication to realise the Riemann solve data flow at the interface between coupled coarse and fine grid cells. The proposed mesh data structure allows to identify points in the spacetree where local master-worker is necessary due to the ADER-DG method's Riemann solves. I use this information in ExaHyPE to decouple master and worker at these points via artificial mesh refinement.

To realise limiter indicator based refinement for the a posteriori subcell limiting ADER-DG method, I proposed two novel mesh pre-refinement approaches in this chapter that I generalised

to classic dynamic AMR for the sole ADER-DG scheme. Here, I conceptually decompose mesh refinement events into events that can be predicted, e.g. a wave travels from a fine grid cell into a coarse grid cell, and events that cannot be predicted, e.g. a shock or an interesting solution feature forms on a coarser grid or travels in from a boundary. To address refinement events that can be predicted, I introduced halo refinement, which ensures that there is always one (or more) additional layers of cells around interesting solution features. I have demonstrated that this feature can be integrated straightforwardly into ExaHyPE's mesh refinement procedures. To address unpredictable refinement events, I introduced a posteriori refinement. With this technique, the mesh is first refined according to a solution feature that has newly appeared, but then a rollback to the previous solution state is performed and the time step is rerun with the now prepared mesh. The presented techniques are essential for ExaHyPE's realisation of the a posteriori limiting ADER-DG method on adaptive meshes (see Chapter 7).

Impact of This Work

ExaHyPE's adaptive mesh refinement techniques have been an important ingredient to obtain competitive simulation results in the context of seismic wave propagation, cloud formation processes, and shallow water flow [92][75][83].

Outlook

In a next step, ExaHyPE could integrate halo refinement into the time stepping iterations, i.e. run mesh refinement on-the-fly while ADER-DG time steps are run. Concerning a posteriori mesh refinement for unlimited DG methods, it must be evaluated if it pays off to store the previous solution vector as this reduces the maximum problem size that can be tackled by the method.

7

Limiting Hybrid ADER-DG-FVM

In this chapter, I transform the a posteriori limiting ADER-DG method into a hybrid ADER-DG-FVM solver to reduce the number of communication phases of the method. The well-suitedness of a numerical scheme for massively parallel machines depends to a large degree upon the number of data exchanges required by the scheme per time step. The a posteriori ADER-DG method requires up to four neighbour-communication steps per time step due to the additional recomputation with a robust FVM. Two cannot be hidden behind computations and might become a bottleneck in the strong-scaling limit. In the first part of this chapter, I reformulate the a posteriori limiting ADER-DG method as a hybrid ADER-DG-FVM solver. In the best case, this reduces the number of neighbour communication steps to a single one per time step. In the worst case, the original four communication steps are required as the scheme is still as robust as the original approach. The limiter criteria detect shocks and other discontinuities. These solution features can only be resolved with a high mesh resolution. In the second part of this chapter, I present a mesh adaptation procedure that reuses the limiter criteria of the a posteriori limiting framework as refinement criterion. In some scenarios, this frees users completely from specifying a separate refinement criterion.

Contributions

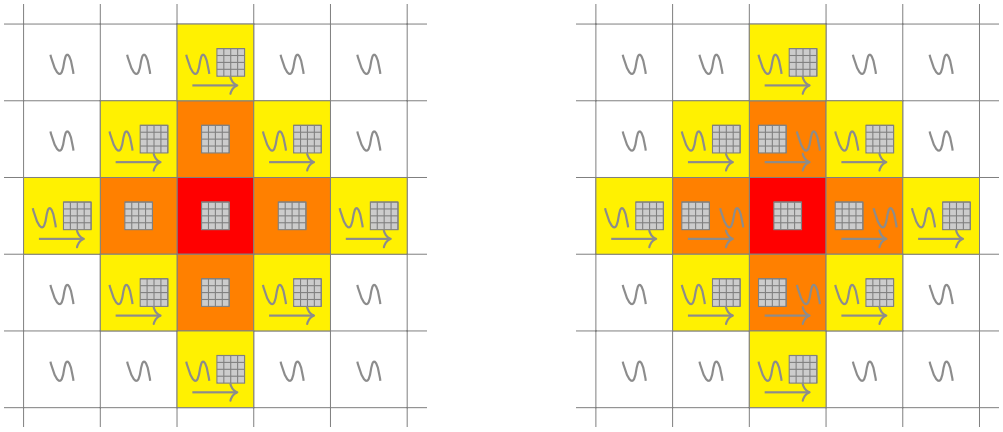
I present two communication-avoiding hybrid solver realisations of the a posteriori limiting ADER-DG method. Detailed pseudocode is presented for both algorithms. Moreover, I detail how the limiter criteria of the a posteriori limiting ADER-DG framework can be reused as refinement indicators based on the halo refinement and a posteriori refinement techniques from Chapter 6. To the best of my knowledge, this is the first time that this has been presented in such a detailed form.

Related Work

The authors of [77] present an DG-FVM hybrid solver that utilises *a priori* indicators. The method has been shown to work well for challenging problems; however, there still remains the chance that the indicators fail if shock formation cannot be predicted. Being based on [54], the hybrid method that I present in this chapter is more resilient as it uses *a posteriori* indicators that are applied to the *updated* solution. It *recomputes* the solution with robust FVM if numerical instabilities are detected after the time step. Reusing the limiter criteria as refinement indicator has been explored briefly in [51], but implementational aspects have not been discussed there.

Structure

This chapter is organised as follows: Section 7.1 introduces the hybrid ADER-DG-FVM solver, which requires only a single communication step if ADER-DG and FVM subdomains remain unchanged. This is, for example, guaranteed when solving the diffuse interface formulation of the elastic wave equations; however, it is usually not the case when solving the compressible Euler equations. Here, the FVM solver is applied on shock fronts that move through the computational domain. To also reduce communication in such scenarios, I merge additional operations into the time stepping iterations; see Section 7.2. In Section 7.3, I employ the halo and a posteriori refinement techniques developed in Chapter 6 to realise mesh refinement that is guided by the hybrid solver's shock detection criteria. Section 7.4 ends the chapter with a discussion and an outlook on a third hybrid solver variant that allows a limited growth of the FVM domain during the time stepping iterations.



(a) Recomputation Stencil **(b)** Subdomains of the Hybrid Method

Fig. 7.1: (a) The recomputation stencil of the original a posteriori subcell limiting as presented in [54]. (b) Subdomains of the hybrid method: The red cells compute with the FVM, the orange cells compute with the FVM and project the result onto a ADER-DG polynomial, the yellow cells compute with ADER-DG and project the result onto an FVM patch, and the white cells compute solely with ADER-DG.

7.1 A Hybrid ADER-DG-FVM Solver

In this section, I reformulate the a posteriori limiting ADER-DG method as hybrid solver. To highlight my modifications, I shortly reiterate principal program flow of the original method. After every ADER-DG time step, the a posteriori limiting ADER-DG method uses a robust finite volume method to recompute the solution in cells where the ADER-DG solution exhibits numerical instabilities or is not physical; see Algorithm 2.5. The assessment whether an ADER-DG solution exhibits such defects is done a posteriori, i.e. after the ADER-DG solution update; see Algorithm 2.6. The cells where such defects are present in the ADER-DG solution are marked as *troubled*. If at least one cell is troubled after the ADER-DG time step, the recomputation procedure is started. Before the FVM recomputation, a posteriori limiting ADER-DG creates FVM subdomains. This requires to diffuse a limiter status for two iterations; see Algorithm 2.4. In this algorithmic phase, troubled cells get assigned the maximum limiter status $L_{\max}$, while face-connected neighbours and second degree neighbours get assigned the limiter status $L_{\max} - 1$ and $L_{\max} - 2$, respectively. All cells with limiter status $\geq L_{\max} - 2$ roll back to the previous time step. They either project a previous stable ADER-DG solution onto an FVM patch or reuse the FVM patch from the previous time

step if the ADER-DG solution was already not stable back then. Afterwards, all cells that carry a limiter status $\geq L_{\max} - 2$ exchange patch boundary layers. Finally, troubled cells and their face-connected neighbours (limiter status $\geq L_{\max} - 1$) evolve the solution with the FVM, while second-degree neighbours simply roll forward in time again (limiter status $= L_{\max} - 2$).

Numerical instabilities in the ADER-DG solution are typically caused by discontinuities such as shock waves. As long as these are present, there is an FVM recomputation necessary in the majority of time steps. In my experiments, I observed that a posteriori recomputation is often applied to the same set of cells for multiple time steps in a row. I assume that this observation is linked to the time step size restrictions that apply to high-order ADER-DG schemes; see e.g. the CFL condition (2.8). The high subcell resolution of these schemes requires that a small time step size must be chosen. Consequently, it takes a wave a proportional number of time steps to pass through the cell.

After the FVM recomputation step, the a posteriori limiting ADER-DG method continues as follows: In all previously troubled cells and their face-connected neighbours, the ADER-DG solution polynomial is recreated as projection of the FVM solution; see Algorithm 2.5. In the next time step, all cell solutions are then evolved again with the ADER-DG method. The information if a cell was troubled in the previous step is not used. Motivated by my observation, I make the following assumption:

Assumption 1. *Once an ADER-DG cell is marked as troubled, it likely remains troubled for a number of time steps.*

Under Assumption 1, I propose to reformulate a posteriori limiting ADER-DG as hybrid solver that connects FVM and ADER-DG subdomains. The original method's troubled cells and their face-connected neighbours belong to an FVM subdomain (limiter status $\geq L_{\max} - 1$) whilst ADER-DG cells and second degree neighbours of troubled cells belong to an ADER-DG subdomain (limiter status $< L_{\max} - 1$). Following the recomputation procedure (Algorithm 2.5), the FVM and ADER-DG subdomains are interfaced by projecting the FVM solution patches onto a ADER-DG polynomial in face-connected neighbours of troubled cells and doing the reverse in second degree neighbours of troubled cells (Fig. 7.1 (b)). Where Assumption 1 holds, the cell updates performed by the hybrid ADER-DG-FVM solver are identical to the ones performed by the original approach [54]. Contrary to the latter, a single additional step of the dissipative FVM will be run in cells where the assumption does not

hold.

Notation. *In the algorithms presented in this chapter, I do not include the predictor step of the ADER-DG method as it is not relevant in this chapter. In fact, the concepts and algorithms presented in this chapter are more general than presented; other numerical methods could be substituted for ADER-DG and FVM.*

Having defined the subdomains and the operations performed by the cells in the interface layers, I present the hybrid ADER-DG-FVM scheme in Algorithm 7.1, where I omit most technical details of ADER-DG and FVM.

Algorithm 7.1 (A Posteriori Limiting Hybrid ADER-DG-FVM Method)**.** *The algorithmic phases of the a posteriori limiting hybrid ADER-DG-FVM method. The limiter status is taken into account when performing neighbour exchange and cell updates (blue). Cells with troubled status recompute the local minimum and maximum again with the* FVM *solver after evaluating limiter criteria with the* ADER-DG *values (green). If a cell becomes newly untroubled or newly troubled, a limiter status diffusion is triggered (red). In the latter case, an* FVM *cells recomputation is triggered, too.*

```
1  function staticHybridSolverTimeStep(ΔT)
2    for face−connected cells K_a, K_b ∈ 𝒯 do                    # loop over faces
3      if limiterStatus_{K_a} < troubledStatus and limiterStatus_{K_b} < troubledStatus:
4        exchange DG Riemann solver input data
5      end if
6      if limiterStatus_{K_a} > 0 and limiterStatus_{K_b} > 0:
7        copyBoundaryLayers(q_{h,FV}(·,T)|_{K_a}, q_{h,FV}(·,T)|_{K_b})
8      end if
9      neighbourMin_{K_a} ← min(neighbourMin_{K_a}, localMin_{K_b})
10     neighbourMax_{K_a} ← max(neighbourMax_{K_a}, localMax_{K_b})
11     neighbourMin_{K_b} ← min(neighbourMin_{K_b}, localMin_{K_a})
12     neighbourMax_{K_b} ← max(neighbourMax_{K_b}, localMax_{K_a})
13   end for
14
15   recomputeFVCells ← false
16   updateSolverDomains ← false
17   for cell K ∈ 𝒯 do                                           # loop over cells
18     if limiterStatus_K ≥ troubledStatus − 1 then
19       q_{h,FV}(·,T+ΔT)|_K ← updateFV(q_h(·,T)|_K)
20       q_h(·,T+ΔT)|_K ← represent q_{h,FV}(·,T+ΔT)|_K as polynomial
```

```
21      else if limiterStatus_K = troubledStatus − 2
22        q_h(·, T + ΔT)|_K ← updateDG(q_h(·, T + ΔT)|_K)
23        q_{h,FV}(·, T + ΔT)|_K ← average polynomial q_h(·, T + ΔT)|_K in V, ∀V ∈ K
24      else
25        q_h(·, T + ΔT)|_K ← updateDG(q_h(·, T + ΔT)|_K)
26      end if
27      # evaluate the limiter criteria
28      (localMin_K, localMax_K) ← evalMinAndMax(q_h(·, T + ΔT)|_K)
29      isTroubled ← ¬DMP(localMin_K, localMax_K, neighbourMin_K, neighbourMax_K) or
30                   ¬PAD(q_h(·, T + ΔT)|_K)
31      if limiterStatus_K = troubledStatus then
32        (localMin_K, localMax_K) ← evalMinAndMax(q_{h,FV}(·, T + ΔT)|_K)
33      end if
34      neighbourMin_K ← localMin_K
35      neighbourMax_K ← localMax_K
36      # update limiter status
37      oldLimiterStatus_K ← limiterStatus_K
38      if isTroubled = true and limiterStatus_K ≠ troubledStatus then
39        limiterStatus_K ← troubledStatus
40        updateSolverDomains ← true
41        recomputeFVCells ← true
42      else isTroubled = false and oldLimiterStatus_K = troubledStatus
43        limiterStatus_K ← 0
44        updateSolverDomains ← true
45      end if
46    end for
47    return ( updateSolverDomains, recomputeFVCells )
48  end function
```

The hybrid solver triggers an update of the ADER-DG and FVM domains if a previously troubled cell is not troubled anymore. To this end, Algorithm 2.4 is employed plus a routine for deallocating the FVM patches (not shown). Continuing to compute with the FVM in these cells would introduce unnecessary numerical dissipation. The algorithm also triggers an update of the domains if previously untroubled cells become troubled. Additionally, it triggers an FVM recomputation with Algorithm 2.5. Under Assumption 1, this is exactly the behaviour of the original method. Contrary to the original approach, the local solution minimum and maximum is computed from the FVM patch in troubled cells. The ADER-DG polynomials in these cells contain spurious oscillations which could deteriorate the accuracy of the DMP.

7.2 Dynamic Curing

I next extend Algorithm 7.1 such that it dynamically transforms previously troubled cells, i.e. cells that compute with finite volumes, back into ADER-DG cells. To this end, the proposed scheme builds the limiter status diffusion into the time stepping iterations. The scheme still requires that an FVM recomputation step is run when a cell becomes troubled, i.e. when this cell wasn't troubled before.

The method works as follows: If a troubled cell is free of artificial oscillations after a time step, the dynamic curing procedure first transforms this cell into an ADER-DG-to-FVM projection cell (limiter status = $L_{\max} - 2$). During the neighbour exchange at the begin of the next time step, the cells now also exchange their limiter status. If the previously cured cell, i.e the new ADER-DG-to-FVM projection cell is informed that one face-connected neighbour is troubled, it raises its limiter status to $L_{\max} - 1$, i.e. it becomes an FVM-to-ADER-DG projection cell. If all surrounding cells are FVM-to-ADER-DG projection cells or pure ADER-DG cells, then the new ADER-DG-to-FVM projection cell decreases its limiter status again and becomes a pure ADER-DG cell. In the last remaining case, i.e. no neighbour is troubled and one neighbour is an FVM-to-ADER-DG cell (limiter status = $L_{\max} - 1$), the new ADER-DG-to-FVM projection cell keeps it limiter status of $L_{\max} - 2$.

Algorithm 7.2 (Dynamically Curing Hybrid ADER-DG-FVM Method)**.** *Algorithmic phases of the time step of the dynamically curing hybrid method. No limiter status diffusion step is necessary when a cell is cured, i.e. is not troubled anymore (blue). The method exchanges the limiter status between neighbours at the begin of every time step and uses this information to decide what update to perform (green).*

```
1  function curingHybridSolverTimeStep(ΔT)
2    for face−connected cells K_a, K_b ∈ T do                    # loop over faces
3      if limiterStatus_{K_a} < L_max and limiterStatus_{K_b} < L_max:
4        exchange DG Riemann solver input data
5      end if
6      if limiterStatus_{K_a} > 0 and limiterStatus_{K_b} > 0:
7        copyBoundaryLayers(q_{h,FV}(·,T)|_{K_a}, q_{h,FV}(·,T)|_{K_b})
8      end if
9      neighbourMin_{K_a} ← min(neighbourMin_{K_a}, localMin_{K_b})
10     neighbourMax_{K_a} ← max(neighbourMax_{K_a}, localMax_{K_b})
```

```
    neighbourMin_{K_b} ← min(neighbourMin_{K_b}, localMin_{K_a})
    neighbourMax_{K_b} ← max(neighbourMax_{K_b}, localMax_{K_a})
    # exchange limiter status
    mergedLimiterStatus_{K_a} ← max(mergedLimiterStatus_{K_a}, limiterStatus_{K_b})
    mergedLimiterStatus_{K_b} ← max(mergedLimiterStatus_{K_b}, limiterStatus_{K_a})
  end for

  recomputeFVCells ← false
  updateSolverDomains ← false
  for cell K ∈ 𝒯 do                                              # loop over cells
    if mergedLimiterStatus_K ≥ L_max − 1 then
      q_{h,FV}(·, T + ΔT)|_K ← updateFV(q_h(·, T)|_K)
      q_h(·, T + ΔT)|_K ← represent q_{h,FV}(·, T + ΔT)|_K as polynomial
    else if mergedLimiterStatus_K = L_max − 2
      q_h(·, T + ΔT)|_K ← updateDG(q_h(·, T + ΔT)|_K)
      q_{h,FV}(·, T + ΔT)|_K ← average polynomial q_h(·, T + ΔT)|_K in V, ∀V ∈ K
    else
      deallocate any FV patch                                    # here it is safe
      q_h(·, T + ΔT)|_K ← updateDG(q_h(·, T + ΔT)|_K)
    end if
    # evaluate the limiter criteria
    (localMin_K, localMax_K) ← evalMinAndMax(q_h(·, T + ΔT)|_K)
    isTroubled ← ¬DMP(localMin_K, localMax_K, neighbourMin_K, neighbourMax_K) or
                 ¬PAD(q_h(·, T + ΔT)|_K)
    if limiterStatus_K = L_max then
      (localMin_K, localMax_K) ← evalMinAndMax(q_{h,FV}(·, T + ΔT)|_K)
    end if
    neighbourMin_K ← localMin_K
    neighbourMax_K ← localMax_K
    # update limiter status
    oldLimiterStatus_K ← limiterStatus_K
    if isTroubled = true and limiterStatus_K ≠ L_max then
      limiterStatus_K ← L_max
      updateSolverDomains ← true
      recomputeFVCells ← true
    else isTroubled = false and oldLimiterStatus_K = L_max
      limiterStatus_K ← limiterStatus_K − 2
    end if
    mergedLimiterStatus_K ← limiterStatus_K
  end for
  return ( updateSolverDomains, recomputeFVCells )
end function
```

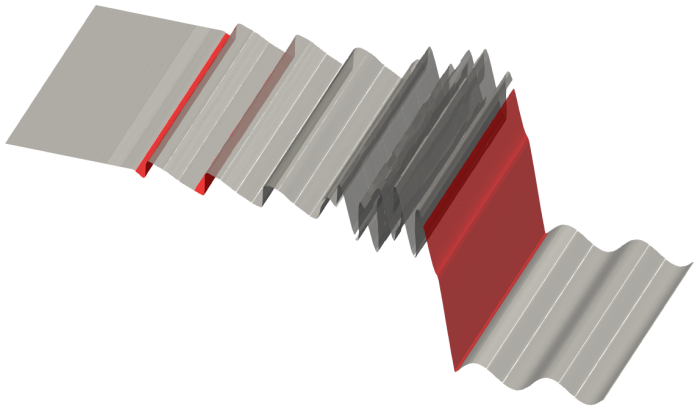
Experimental Evidence

First, I validate that the implementation of the curing hybrid scheme is computing correctly and does not add too much dissipation. To this end, I consider the Shu-Osher 1D benchmark for compressible Euler codes [88]. Here, a domain with bounds $[-5, 5]$ is separated at $x = -4$ into two regions: In the left region, density and velocity are initially chosen $\rho_{\mathrm{L}} = 3.8571$ and $v_{\mathrm{L}} = 2.6294$, respectively. The pressure is chosen as $p_{\mathrm{L}} = 10.333$. In the right region, the initial values are $\rho_{\mathrm{R}} = 1 + 0.2 \cdot \sin(5\,x)$, $v_{\mathrm{R}} = 0$, and $p_{\mathrm{R}} = 1$. The EOS is chosen as the one of an ideal gas. The simulation is run until non-dimensional time $T_{\mathrm{final}} = 1.8$.

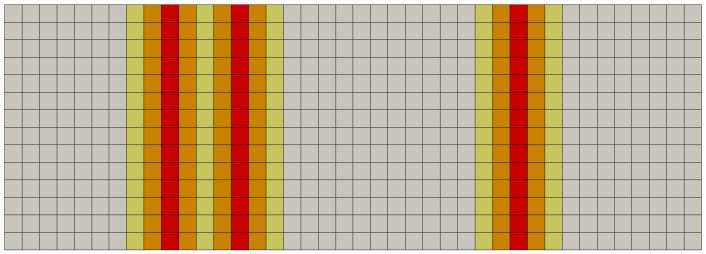
Shu-Osher is considered a difficult benchmark as the interaction of the harmonic density perturbation with the shock generates high-frequent oscillations behind the main shock. If a scheme applies too much numerical dissipation to capture the shocks, it will damp out these oscillations. If a scheme does not use enough numerical dissipation, spurious oscillations develop around the main shock and around secondary shocks that form at the end of the simulation. To treat the shock in the Shu-Osher benchmark, I use ExaHyPE's MUSCL-Hancock method in combination with a Koren slope limiter [74]. The reference solution for this test was computed with a third-order ADER-WENO ("Weighted Essentially Non-Oscillatory") scheme on a very fine mesh. The reference data was kindly provided by Franceso Fambri. In order to enable a comparison with [54], I employ 40 mesh cells along the x direction via the bounding box scaling mechanism presented in Ch. 6.

The solution computed with ExaHyPE's 9^{th}-order hybrid ADER-DG-FVM solver matches both reference solutions well even though ExaHyPE uses a coarse mesh with only 40 cells, a second-order FVM limiter, and a simple Rusanov flux for ADER-DG and FVM (Fig. 7.2). It seems not to be necessary to use a higher-order WENO (Weighted Essentially Non-Oscillatory) scheme as limiter and the more expensive generalised Osher flux as Riemann solver that the authors of [54] used. However, I found that it is essential to use a good slope limiter such as the Koren limiter. The classic minmod limiter leads to significantly more numerical dissipation in this test.

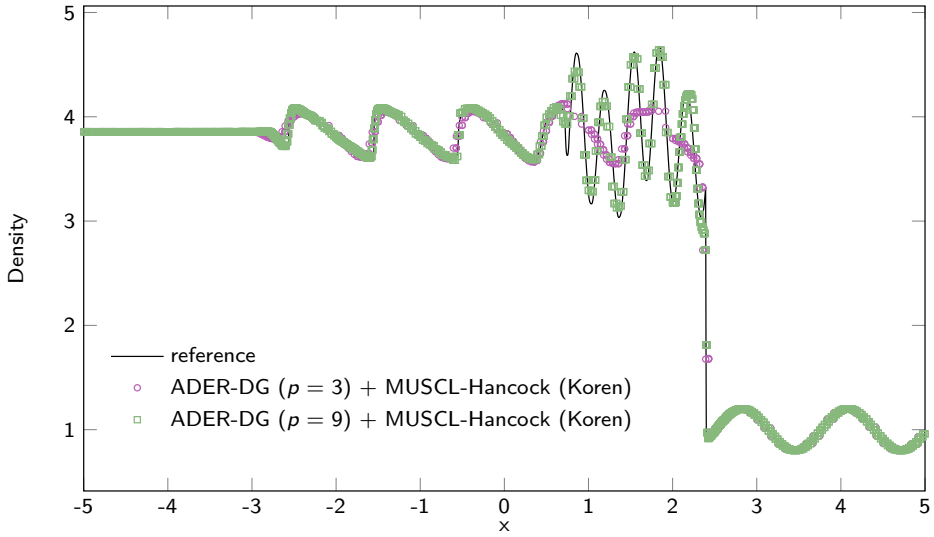
Figure 7.3 shows the number of communication steps when the curing limiting scheme is used instead of the original implementation. Three standard benchmark scenarios are considered: The first two, the Sod shock tube and the Shu-Osher problem, are 1D problems while the third



(a)



(b)



(c)

Fig. 7.2: In the Shu-Osher test, a shock and a smooth harmonic wave interact. (a) Troubled cells (red) at time $T = 1.8$ superimposed on the warped density profile. (b) Troubled FVM cells (red) and interface cells (orange and yellow) surrounded by ADER-DG cells (white) at time $T = 1.8$. (c) Comparison of results obtained with ExaHyPE's hybrid ADER-DG-FVM solver against a third-order ADER-WENO result that was computed on a very fine mesh.

models circular two-dimensional explosion. In all experiments, the limiter is active in every time step; see [54] for a detailed description of the explosion problem and the Sod shock tube. Therefore, the original approach requires four communication steps in every time step, i.e. one ADER-DG communication step, 2 limiter status diffusion iterations, and one exchange of FVM boundary layers. The curing scheme works well for the 1D Riemann problems; however, it is less successful if the 2D explosion problem is considered. As expected, the curing scheme is more successful if a high-order ADER-DG method is employed. It seems less effective if the spatial resolution of the 2D problem increases while no such trend can be spotted for the 1D problems.

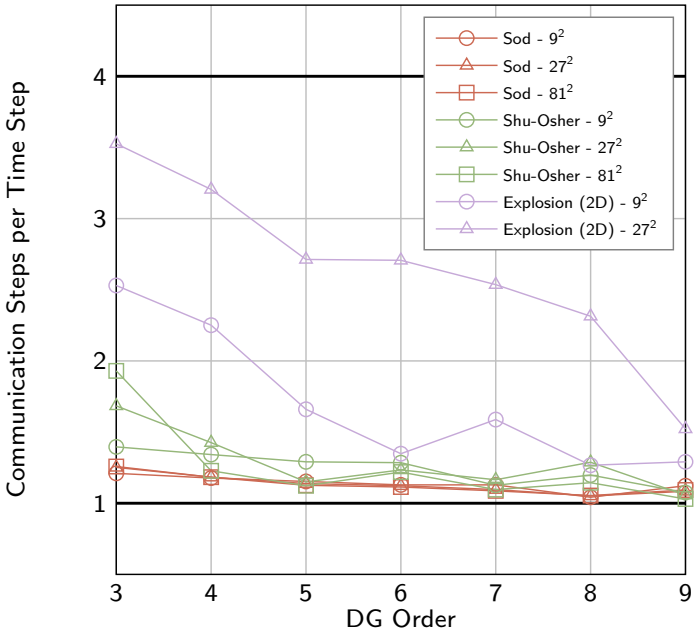
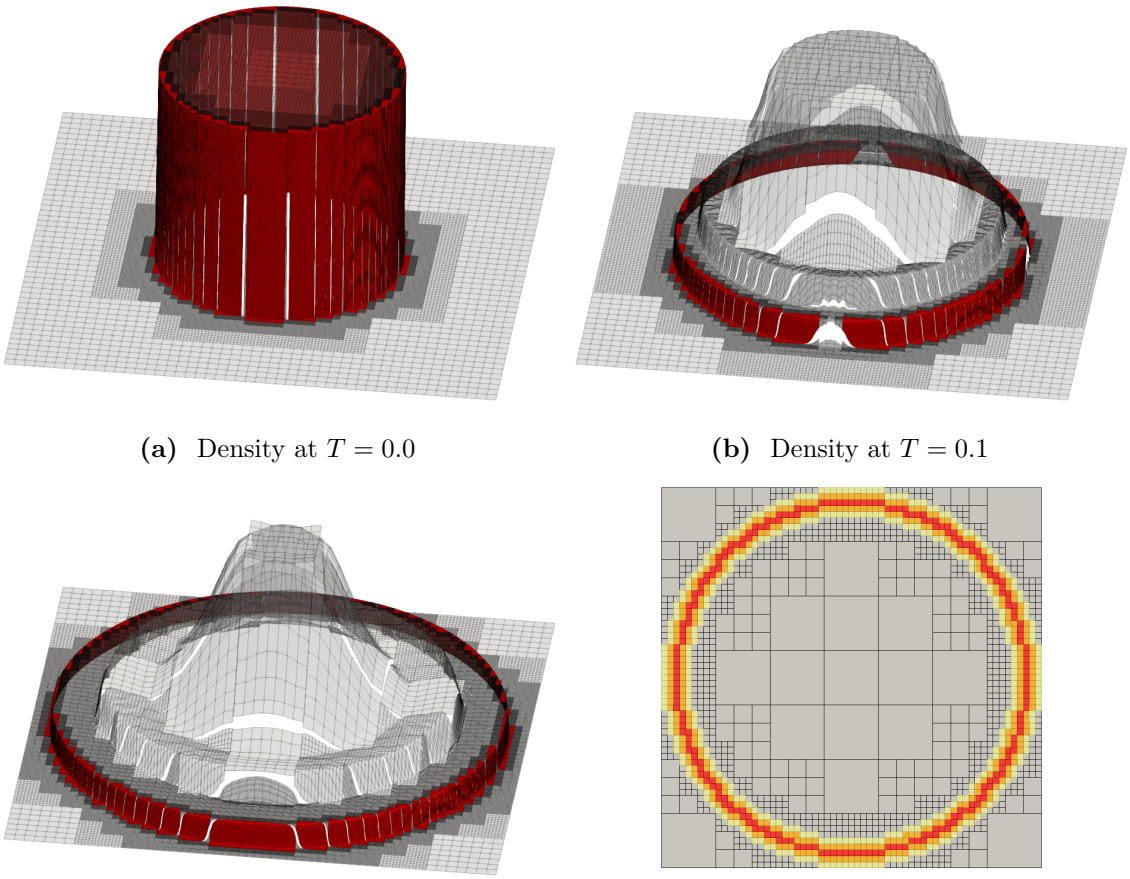


Fig. 7.3: Averaged number of neighbour communication steps per time step of the curing hybrid ADER-DG-FVM solver. The original method requires 4 steps for these problems where limiting is necessary in every time step. The minimum possible number of communication steps is the number of communication steps of the unlimited ADER-DG method, which is 1.

7.3 Limiter-Criteria-Guided Refinement

ExaHyPE realises limiter-criteria-guided refinement via a combination of the halo and a posteriori refinement techniques proposed in Chapter 6. High order polynomial approximations can resolve most smooth features on a rather coarse resolution. A fine mesh resolution is typically only required in the vicinity of strong gradients, discontinuities, and complicated



(a) Density at $T = 0.0$

(b) Density at $T = 0.1$

(c) Density at $T = 0.2$

(d) Mesh and Subdomains at $T = 0.2$

Fig. 7.4: Simulation of a circular explosion in an ideal gas. A 7^{th}-order a posteriori limiting ADER-DG was used and the computational mesh was adapted according to the limiter criterion that detects oscillations in density and pressure. To visualise the high-order polynomials in plot (a) – (c), the density solution in the ADER-DG cells is interpolated onto an uniform 8×8 subgrid. The actual computational grid corresponding to plot (c) is shown in plot (d) along with the hybrid solver subdomains. At $T = 2.0$, the simulation used 1572 cells that computed primarily with ADER-DG and 236 that computed primarily with FVM.

geometries. As the admissibility checks of the a posteriori limiting ADER-DG method identifies shock waves, it makes sense to (re-)use them as refinement criterion. However, if a shock has been detected in a cell via the limiter criteria, this cell does contain a troubled ADER-DG solution. We must go back in time. This links to a posteriori refinement. The authors of [109] observe spurious oscillations in the neighbours of cells that resolve a shock. To prevent these oscillations, they suggest to artificially refine a number of additional cells in a radius around every shock. This links to halo refinement.

ExaHyPE's limiter-criteria-guided refinement is realised as follows: If troubled cells are detected on a coarser grid, the respective cells are refined to the finest adaptive mesh level of the grid (Fig. 7.4). Then, halo refinement is employed on top to bring face-connected and second-degree neighbours onto the finest mesh level. To accomplish this, the maximum halo status is chosen equal to the maximum limiter status $L_{\max}$. On the one hand, this halo refinement prevents pre- and post-shock oscillations. On the other hand, it simplifies the implementation of the FVM limiter as no inter-grid transfer operators are required. In the last step, a global rollback is performed, i.e. all cell solutions, if troubled or not, are rolled back to their state of the previous time step. The whole time step is then rerun with the new effectively pre-refined mesh.

7.4 Discussion and Outlook

In this chapter, I reformulated the a posteriori limiting ADER-DG method as a hybrid ADER-DG-FVM solver based on the observation that the set of cells that hold a troubled ADER-DG solution often remains unchanged over multiple time steps. I then extend the hybrid solver algorithm such that the process of curing FVM cells, i.e. the transitioning of these cells from being evolved with the FVM back to being evolved with ADER-DG, is merged into the time stepping iterations. I demonstrated that this solver significantly reduces recomputations and thus the communication steps of the a posteriori limiting ADER-DG method for problems with dynamically evolving FVM domain. In particular, it worked well if the ADER-DG scheme uses a high approximation order. The dynamically curing solver only stops its time stepping iterations when the limiter domain grows, i.e. a shock travels into the next cell. In the last part of this chapter, I detailed how limiter-criteria-guide refinement can be realised for this hybrid solver using the halo refinement and a posteriori refinement

techniques developed in Chapter 6.

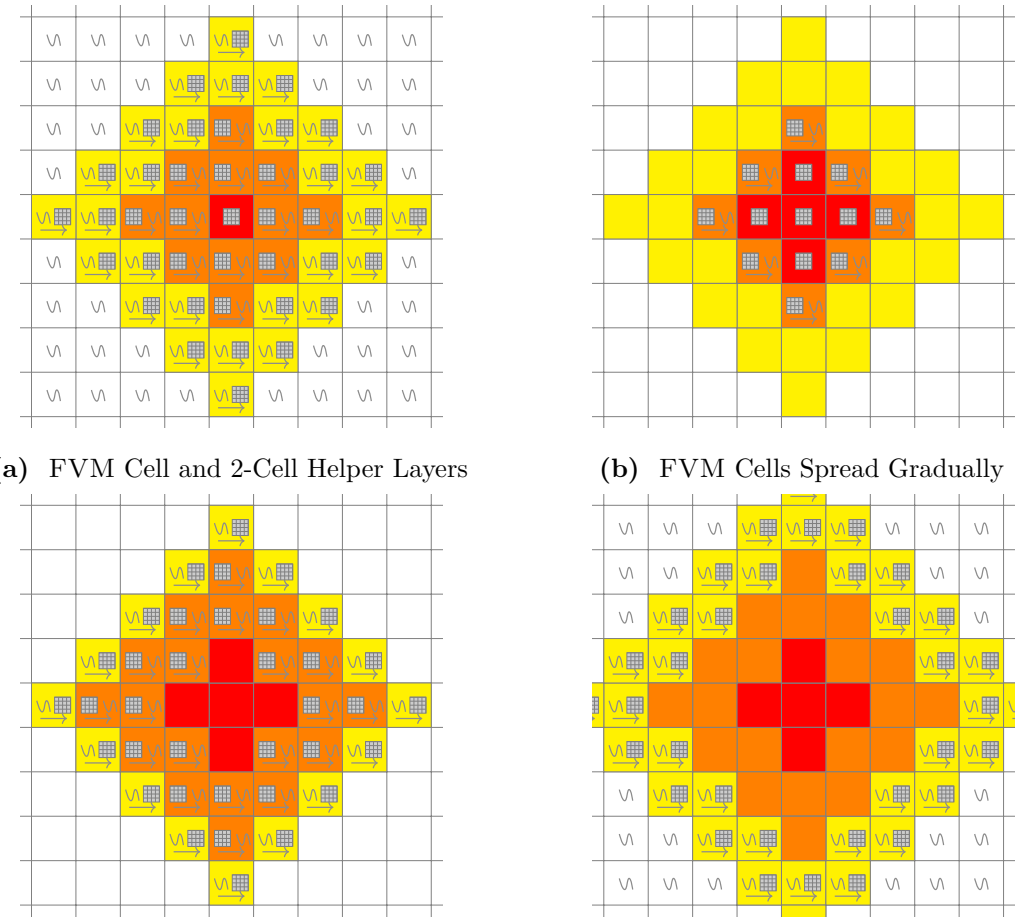
As a minor unforeseen result, I demonstrated that the a posteriori subcell limiting ADER-DG method achieves very competitive results in the Shu-Osher benchmark with a simple second-order MUSCL-Hancock method in combination with a cheap but dissipative Rusanov flux if the MUSCL-Hancock scheme uses a Koren slope limiter [74].

Lastly, I want to stress that the proposed horizontal coupling concepts are more general as they are presented in this chapter. ADER-DG and FVM can be readily replaced with any other numerical method. Furthermore, the coupling mechanisms can be applied to applications other than treating shocks and discontinuities. ExaHyPE uses it, e.g., to realise diffuse interface methods that require a high resolution along the interface. At their core, the presented algorithms realise a dynamic coupling of a macroscopic and a microscopic model. Future research could generalise the presented two-level schemes to multi-scale schemes that couple more than two different models.

Outlook: Dynamically Growing FV Domains

As wave speeds are limited, shock propagation can often be anticipated. Therefore, a natural next step would be to allow the FVM domain to also grow dynamically during the time stepping iterations. However, this cannot be accomplished by a simple extension to the dynamically curing hybrid solver as this could lead to situations where untroubled ADER-DG cells and troubled FVM cells would communicate directly with each other via the DG solver. In this case, the untroubled ADER-DG cell's solution could get polluted by spurious oscillations in the ADER-DG Riemann input data provided by the troubled cell. Additionally, troubled cells and their face-connected neighbours require FVM boundary data from neighbouring cells. Therefore, troubled cells and their neighbours cannot be placed directly next to a pure ADER-DG cell.

A dynamic growth of the FVM domain can be accommodated if at least one additional layer of ADER-DG-to-FVM and FVM-to-ADER-DG projection cells is introduced around troubled cells. The more of these layers are introduced the more freely the FVM domain can grow but the more numerical dissipation is introduced in areas where it is not required and potentially harmful. Therefore, a high-order finite volumes schemes or limiter-criteria-guided AMR should be used to keep the numerical dissipation under control. The cartoon Fig. 7.5



(a) FVM Cell and 2-Cell Helper Layers

(b) FVM Cells Spread Gradually

(c) FVM-to-ADER-DG Layers Recover

(d) ADER-DG-to-FVM Layers Recover

Fig. 7.5: Diffusive Approach: Expansion of FVM subdomain and recovery of helper layers within 3 time steps. The communication between the cells is never interrupted. (a) Initial configuration. (b) The FVM subdomain (red) grows but its cells are surrounded by FVM and FVM-to-ADER-DG cells (orange). Therefore, no FVM recomputation step is necessary. The time stepping can continue. (c) The FVM-to-ADER-DG has recovered. The FVM-to-ADER-DG cells are only surrounded by ADER-DG-to-FVM (yellow) and FVM cells. The time stepping can continue. (d) The ADER-DG-to-FVM layers have recovered, too.

shows how such a growth of the limiter domain could play out.

Outlook: Balanced Cost

The hybrid solvers in this chapter have been presented under the assumption that the computational cost of an FVM cell update is lower or at least equal to the cost of the ADER-DG cell update (including predictor and Riemann step). However, in practice, the FVM update can be significantly more expensive than the ADER-DG update; see Fig. 7.6. If a recomputation is run after most time steps, the time to solution of the hybrid solver might be above that of the original approach.

However, note that the results shown in Fig. 7.6 have been collected on a Skylake-X machine that supports `AVX512` instructions. (The exact setup and the used software is described in Sec 9.3.) While ExaHyPE aggressively optimises the ADER-DG kernels, it currently completely relies on automatic compiler optimisations for the FVM kernels (state: June 2019). A vectorisation of the FVM compute kernels should thus be subject of future work. In addition, this study could investigate if the FVM kernels can be multi-threaded. Another study could investigate FVM patch sizes other than $(2\,p+1)^d$ where p is the polynomial order of the ADER-DG scheme and d the space dimension. To enable these studies, ExaHyPE could provide a specification file option to choose the FVM patch size arbitrarily.

Outlook: DMP Tuning Parameters

The discrete maximum principle (DMP) in the a posteriori limiting ADER-DG framework is generic up to two tuning parameters. In the framework and ExaHyPE's hybrid solver implementation, they appear as an order- and mesh-independent constant; however, they show a dependence on both and the dynamic range of the solution in many experiments. Different choices of these parameters can lead to a vastly different behaviour of the method. An investigation of the dependence of the DMP tuning parameters on the other named discretisation parameters should be subject to further research.

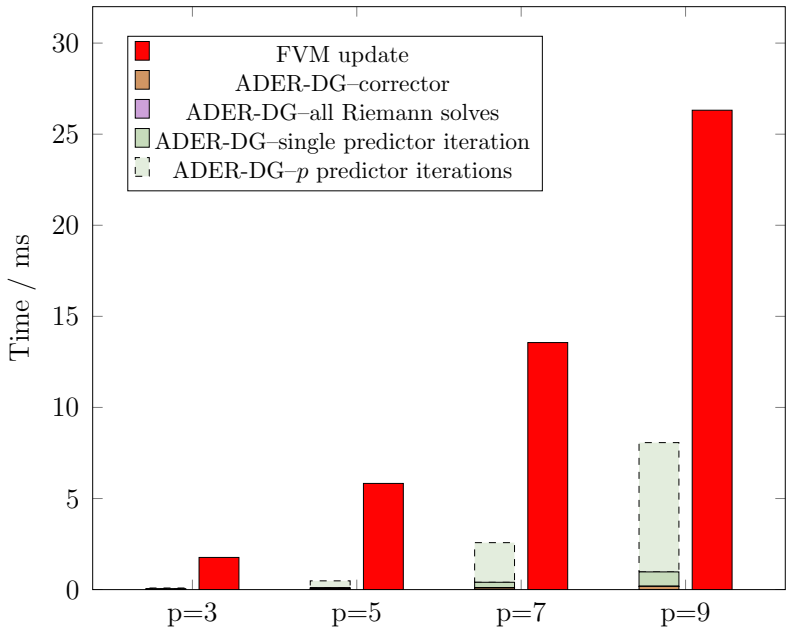


Fig. 7.6: Compressible Euler: Typical cost of the MUSCL-Hancock cell update in comparison to the cost of optimised ADER-DG operations. In the best case, ADER-DG runs a single predictor iteration. In the worst case, $p + 1$ iterations are run. *Data was collected on SuperMUC-NG. Shown is the average out of 50 measurements per operation.*

8

Communication-Avoiding Low-Storage ADER-DG

In this chapter, I present ADER-DG realisations that are efficient in terms of memory footprint and memory access. The ADER-DG method is a generic recipe for writing explicit high-order discretisations in space and time that lends itself straightforwardly to adaptivity in space and time However, the space-time memory footprint of the method limits the size of problems that can be solved as the method's memory footprint is dominated by space-time quantities. To address this issue, the first part of this chapter presents a low-storage realisation of ADER-DG that reduces the method's memory footprint significantly for approximation orders $p \geq 3$. I accomplish this by moving the volume integral and boundary extrapolation substeps of ADER-DG into the earlier predictor phase. This requires to store the boundary-extrapolated predictor and normal flux, but it has the advantage that space-time predictor and volume flux become temporary variables. I then apply a second storage optimisation on top: If the used Riemann solver acts linearly on its inputs, the boundary-extrapolated space-time predictor and volume flux, both can be averaged in time directly after the predictor substep; i.e. the time integration of the ADER-DG corrector's face integral is moved into the earlier predictor step, too. To the best of my knowledge, all of ExaHyPE's applications use a linear Riemann solver. On top of the storage optimisations, I construct communication-avoiding

ADER-DG variants. These variants minimise data movement between CPU and main memory additionally.

Contributions

In this chapter, I propose communication-avoiding low-storage realisations of ADER-DG and compare them in terms of their memory footprint and memory access behaviour. The presented realisations reduce the memory footprint of ADER-DG by more than a factor of 10 for high polynomial orders. This chapter provides detailed pseudocode for all discussed algorithms. The majority of my optimisations apply to other predictor-corrector schemes, too. In ExaHyPE, I have applied them to the FVM and the hybrid ADER-DG-FVM solver. The preprint [39] presents parts of this chapter.

Related Work

Low-storage Runge-Kutta methods are a topic of research since the 50s [60]; a comprehensive overview is given in [70]. These methods reduce the vectors required to store the outcome of the individual Runge-Kutta stages, which reduces memory footprint and can improve cache usage. These techniques are not applicable to the ADER-DG method, which has always two stages for any chosen temporal approximation order: the predictor stage followed by the corrector stage. The output of both stages cannot be swapped. Different techniques need to be developed for ADER-DG.

Structure

Section 8.1 discusses the memory requirement of the straightforward ADER-DG realisation from Chapter 2, the baseline. Section 8.2 presents my storage optimisations for the ADER-DG methods. Section 8.3 presents communication-avoiding fused ADER-DG realisations. Section 8.4 provides a theoretical analysis of all proposed algorithms. In Section 8.5, I demonstrate that the presented schemes correctly implement the ADER-DG method by means of a convergence test. Furthermore, I compare the performance of the sole low-storage implementation of ADER-DG against the performance of the fused variant. This chapter concludes with a discussion in Section 8.6.

8.1 Memory Analysis of the Baseline

The memory analysis conducted and the algorithms proposed in this chapter are motivated by an idealised hardware model. In this external memory machine [20], each CPU accesses their local main memory not directly but via a cache. The model assumes that access to the main memory and receiving data from other CPUs via a network is magnitudes more expensive than access to the cache [21]. The cache is assumed to implement an idealised write-back policy: When running a loop over the ADER-DG cells or faces, ADER-DG solution and auxiliary fields are assumed to be held in cache until the last modification to them has occured during that loop. Only then, these data are written back to main memory.

> While Chapter 9 presents a hybrid parallelisation that hides network communication behind computations, this chapter is concerned with minimising the communication between CPU and main memory. To obtain optimal performance, data movement between main memory and CPU must be minimised. Once data is loaded into one of the CPU caches, it must be reused as often as possible.

Algorithm 8.1 extends Algorithm 2.3 with additional lines indicating read and write access to persistently stored data during the time stepping loops. Here and in the following, I assume that the cells and faces of a mesh are traversed in a cache-aware order that ensures that the data associated with these mesh entities are read and written only once per loop. Furthermore, I assume that temporary variables such as quadrature weights stay always in cache.

Algorithm 8.1 (Straightforward Invariant-Time-Step-Size ADER-DG)**.** *One time step of the simplest version of ADER-DG without limiter as pseudo code. The original algorithm is extended by read and write operations (green) during the three characteristic ADER-DG time stepping phases, predictor, Riemann solves, and corrector.*

```
1  T ← 0
2  initialiseADERDG( )
3
4  while T < T_final do
5    for cell K ∈ 𝒯 do
```

```
    read q_h(·,T)|_K
    ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K, ΔT)
    q_h(·,T+ΔT)|_K ← q_h(·,T)|_K
    write q*_h|_K, F(q*_h)|_K                    # assumes q_h(·,T)|_K is reused
  end for
  for face−connected cells K_a, K_b ∈ T do
    read q_h(·,T+ΔT)|_{K_a}, q*_h|_{K_a}, F(q*_h)|_{K_a} (if not already cached)
    read q_h(·,T+ΔT)|_{K_b}, q*_h|_{K_b}, F(q*_h)|_{K_b} (if not already cached)
    ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
    ( q*_h|_{K_b}|_{∂K_b∩∂K_b}, n·F(q*_h)|_{K_b}|_{∂K_b∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
    G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
                                q*_h|_{K_b}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
    q_h(·,T+ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h)(n·n_{K_a}), ∂K_a∩∂K_b, ΔT)
    q_h(·,T+ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h)(n·n_{K_b}), ∂K_a∩∂K_b, ΔT)
    write q_h(·,T+ΔT)|_{K_a} (if all faces done)
    write q_h(·,T+ΔT)|_{K_b} (if all faces done)
  end for
  for cell K ∈ T do
    read q_h(·,T+ΔT)|_K, F(q*_h)|_K
    q_h(·,T+ΔT)|_K += volumeIntegral(F(q*_h)|_K, ΔT)
    write q_h(·,T+ΔT)|_K
  end for
  T ← T+ΔT
end while

function initialiseADERDG( )
  for cell K ∈ T do
    q_h(·,0)|_K ← represent q(·,0)|_K as polynomial
  end for
end function
```

Algorithm 8.1 is not optimal for solving nonlinear PDE systems. For these, the admissible time step size can change. Using Algorithm 8.1 would require us to use a minimum time step size throughout the whole simulation to ensure that we do not violate the CFL condition. This introduces unnecessary numerical dissipation and makes the simulation take longer than necessary. Therefore, it is better to adapt the time step size throughout the whole simulation. To this end, Algorithm 8.2 extends Algorithm 8.1 with additional lines that compute a CFL-stable time step size ΔT_{adm}. Furthermore, Algorithm 8.2 introduces more lines for reducing the time step size when multiple cores are used plus further lines that highlight

where the neighbour exchange starts and ends. The latter are wrapped around the predictor and Riemann loops as the boundary-extrapolated space-time predictor and space-time flux are inputs to the `Riemann` substep. Depending on the implementation, the neighbour process may send the full space-time predictor and space-time volume flux, which are then stored in a ghost cell, or only the Riemann inputs for a particular face (i.e. the boundary-extrapolated space-time data). In Algorithm 8.2, some overlap between computations and neighbour exchange is assumed. If no overlap is realised by the hardware or by a polling thread, the neighbour exchange must be conducted after the predictor loop and before the Riemann solve loop.

Algorithm 8.2 (Straightforward Variable-Time-Step-Size ADER-DG)**.** *One time step of ADER-DG. The algorithm is extended by time step size computation code, global communication (green), and neighbour communication (blue). If the latter is performed asynchronously, it is conducted between begin and end point. Otherwise, neighbour communication starts and finishes after the predictor loop, right before the Riemann solves.*

```
1  T ← 0
2  ΔT ← initialiseADERDG()
3
4  while T < T_final do
5    begin neighbour exchange                  # either ghost cells or extrapolated values
6    for K ∈ 𝒯 do                                                          # prediction
7      read q_h(·,T)|_K, q*_h|_K, F(q*_h)|_K
8      ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K, ΔT)
9      q_h(·,T+ΔT)|_K ← q_h(·,T)|_K
10     write q*_h|_K, F(q*_h)|_K
11   end for
12   for face−connected cells K_a, K_b ∈ 𝒯 do                              # Riemann solves
13     read q_h(·,T+ΔT)|_{K_a}, q*_h|_{K_a}, F(q*_h)|_{K_a} (if not already cached)
14     read q_h(·,T+ΔT)|_{K_b}, q*_h|_{K_b}, F(q*_h)|_{K_b} (if not already cached)
15     ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
16     ( q*_h|_{K_b}|_{∂K_b∩∂K_b}, n·F(q*_h)|_{K_b}|_{∂K_b∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_b}, F(q*_h)|_{K_b})
17     G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
18                                 q*_h|_{K_b}|_{∂K_a∩∂K_b}, n·F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
19     q_h(·,T+ΔT)|_{K_a} += faceIntegral(G(q*,+_h, q*,−_h)(n·n_{K_a}), ∂K_a ∩ ∂K_b, ΔT)
20     q_h(·,T+ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h)(n·n_{K_b}), ∂K_a ∩ ∂K_b, ΔT)
21     write q_h(·,T+ΔT)|_{K_a} (if all faces done)
22     write q_h(·,T+ΔT)|_{K_b} (if all faces done)
23   end for
```

```
24    end neighbour exchange
25    ΔT_adm ← ∞
26    for cell K ∈ 𝒯 do                                          # correction
27      read q_h(·, T + ΔT)|_K, F(q*_h)|_K
28      q_h(·, T + ΔT)|_K += volumeIntegral (F(q*_h)|_K, ΔT)
29      ΔT_adm ← min{ΔT_adm, calcTimeStep(q_h(·, T + ΔT)|_K)}
30      write q_h(·, T + ΔT)|_K
31    end for
32    T ← T + ΔT
33    begin global reduction (ΔT_adm) end global reduction
34    ΔT ← ΔT_adm
35  end while
36
37  function INITIALISEADERDG()
38    ΔT ← ∞                                                     # compute first time step size
39    for K ∈ 𝒯 do
40      q_h(·, 0)|_K ← represent q(·, 0)|_K as polynomial
41      ΔT ← min{ΔT, calcTimeStep(q_h(·, 0)|_K)}
42    end do
43    begin global reduction (ΔT) end global reduction
44    return ΔT
45  end function
```

Table 8.1 lists the individual ADER-DG substeps and their memory reads and writes with respect to the number of state variables m, the space dimensions d, and the polynomial order p. Let us introduce $M_d := m\,(p+1)^d$ as the typical number of DOFs per ADER-DG cell.

Definition *In this chapter, memory footprint always refers to the storage required to store the ADER-DG scheme's solution data plus the scheme's auxiliary variables such as the space-time predictor and space-time volume flux.*

The straightforward ADER-DG realisations Algorithm 8.1 and Algorithm 8.2 require to persistently store the auxiliary variables of ADER-DG, i.e. space-time predictor and the space-time volume flux, for all cells since these information have to be kept till the third loop, the corrector loop. If we neglect time stepping and grid metadata, the footprint per cell of

both ADER-DG realisations is thus (in doubles):

$$\begin{aligned}\mathrm{MEM}_{\text{ADER-DG}} &= \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T)|_K} + \underbrace{m\,(p+1)^{d+1}}_{\text{size of } q_h^*|_K} + \underbrace{d\,m\,(p+1)^{d+1}}_{\text{size of } F(q_h^*)|_K} \\ &= m\left((p+1)^d + (d+1)\cdot(p+1)^{d+1}\right) \\ &= (d+1)M_{d+1} + M_d.\end{aligned}$$

Despite the explicit character of the scheme, the memory footprint $\mathrm{MEM}_{\text{ADER-DG}}$ of straightforward realisations of ADER-DG is dominated by the space-time term $(d+1)M_{d+1}$. Not surprisingly, the memory reads and writes are dominated by the same term. I break them down into reads $\mathrm{R}_{\text{ADER-DG}}$ and writes $\mathrm{W}_{\text{ADER-DG}}$ per cell per time step. The straightforward realisation's memory reads and writes are (in doubles):

$$\mathrm{R}_{\text{ADER-DG,predictor}} = \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T)|_K} + \underbrace{m\,(p+1)^{d+1}}_{\text{size of } q_h^*|_K} + \underbrace{d\,m\,(p+1)^{d+1}}_{\text{size of } F(q_h^*)|_K} = (d+1)\,M_{d+1} + M_d,$$

$$\mathrm{W}_{\text{ADER-DG,predictor}} = \underbrace{m\,(p+1)^{d+1}}_{\text{size of } q_h^*|_K} + \underbrace{d\,m\,(p+1)^{d+1}}_{\text{size of } F(q_h^*)|_K} = (d+1)\,M_{d+1},$$

$$\mathrm{R}_{\text{ADER-DG,Riemann}} = \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T+\Delta T)|_K} + \underbrace{m\,(p+1)^{d+1}}_{\text{size of } q_h^*|_K} + \underbrace{d\,m\,(p+1)^{d+1}}_{\text{size of } F(q_h^*)|_K} = (d+1)\,M_{d+1} + M_d,$$

$$\mathrm{W}_{\text{ADER-DG,Riemann}} = \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T)|_K} = M_d,$$

$$\mathrm{R}_{\text{ADER-DG,corrector}} = \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T+\Delta T)|_K} + \underbrace{d\,m\,(p+1)^{d+1}}_{\text{size of } F(q_h^*(\cdot,T)|_K)} = d\,M_{d+1}, + M_d,$$

$$\mathrm{W}_{\text{ADER-DG,corrector}} = \underbrace{m\,(p+1)^d}_{\text{size of } q_h(\cdot,T)|_K} = M_d.$$

The total memory reads and writes are consequently (in doubles):

$$\begin{aligned}\mathrm{R}_{\text{ADER-DG}} &= \mathrm{R}_{\text{ADER-DG,predictor}} + \mathrm{R}_{\text{ADER-DG,Riemann}} + \mathrm{R}_{\text{ADER-DG,volumeIntegral}} \\ &= (3\,d+2)\,M_{d+1} + 3\,M_d, \\ \mathrm{W}_{\text{ADER-DG}} &= \mathrm{W}_{\text{ADER-DG,predictor}} + \mathrm{W}_{\text{ADER-DG,Riemann}} + \mathrm{W}_{\text{ADER-DG,volumeIntegral}} \\ &= (d+1)\,M_{d+1} + 2\,M_d.\end{aligned}$$

The large space-time arrays will result in frequent bandwidth-intense access to the main memory. This pushes straightforward ADER-DG realisations towards the memory-bandwidth-bound regime.

Table 8.1: Read and written doubles per ADER-DG substep. The data is normalised per cell or per cell pair (Riemann). Quadrature weights are assumed to be held in cache. The substeps are grouped along the three main steps predictor, Riemann and corrector.

Task	in	out	Remarks
predictor:			
`predictor`	$m \cdot (p+1)^d$	$(d+1) \cdot m \cdot (p+1)^{d+1}$	determine $q_h^*\|_K$ and $F(q_h^*)\|_K$
Riemann:			
`extrapolate`$_{e \subset \partial K}$	$(d+1) \cdot m \cdot (p+1)^{d+1}$	$2 \cdot m \cdot (p+1)^d$	extrapolate $q_h^*\|_K$ and $F(q_h^*)\|_K \cdot n$ to the face e with normal n of cell K
`Riemann`	$2 \cdot 2 \cdot m \cdot (p+1)^d$	$2 \cdot m \cdot (p+1)^d$	solve Riemann problem and obtain numerical flux
corrector:			
`faceIntegral`	$m \cdot (p+1)^d$	$m \cdot (p+1)^d$	perform face integral with numerical flux
`volumeIntegral`	$d \cdot m \cdot (p+1)^{d+1}$	$m \cdot (p+1)^d$	integrate predicted $F(q_h^*)\|_K$
`calcTimeStep`	$m \cdot (p+1)^d$	1	calculate a time step size with updated solution

8.2 Low-Storage ADER-DG

Clearly, we can easily reduce the memory footprint of the ADER-DG method if we do not allocate space-time predictor and volume flux at all, but instead recompute them from the ADER-DG solution whenever we need them. However, experimental data indicates that the `predictor` substep dominates the runtime of medium- and high-order ADER-DG methods; see Fig. 8.1. (Section 9.3 provides information on the setup and used software.) Therefore, I constrain my search to low-storage implementations where the `predictor` substep is only run once per cell per time step. In addition, I focus on problems where the ADER-DG solution snapshots $q_h(\cdot, T)$ do not fit into the CPU caches entirely while a cell's space-time predictor $q_h^*|_K$ and space-time volume flux $F(q_h^*)|_K$ fit.

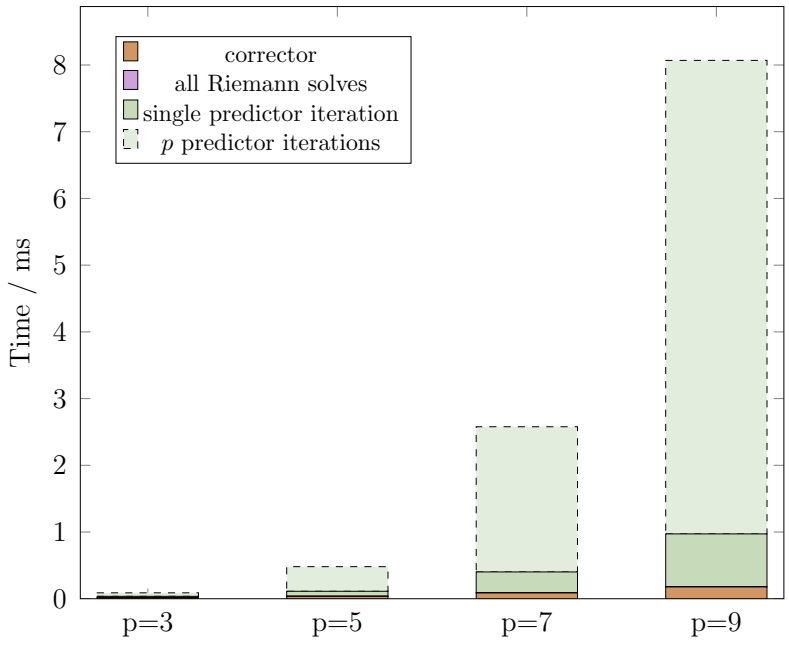


Fig. 8.1: Typical cost of the ADER-DG substeps and their contribution to the cost of a single ADER-DG time step when solving the nonlinear compressible Euler equations. In the best case, the scheme runs a single predictor iteration. In the worst case, $p + 1$ iterations are run. *Data was collected on SuperMUC-NG. Shown is the average out of 50 measurements per operation.*

The processing of ADER-DG substeps (see Table 8.1) must adhere to a partial temporal order, which can be expressed in the form of a task graph; see Fig. 8.2. Additional secondary

tasks may enter this graph on demand: For example, a `plot` task might be inserted after the `calcTimeStep` task. If an adaptive mesh is used, a `refinementCriterion` task may be inserted behind the `calcTimeStep` task and so on.

The graph Fig. 8.2 reveals that the `volumeIntegral` substep that is associated with the ADER-DG method's corrector step needs not to be performed after the Riemann solve but can be run directly after the `predictor` substep. From a comparison with Table 8.1, we observe that the volume integral requires the (predicted) space-time volume flux $F(q_h^*)$ as input. Furthermore, space-time predictor q_h^* and flux $F(q_h^*)$ are required as input to the `extrapolate` step that is run directly after the `predictor` and before the `Riemann` substep. The central observation of this section is:

Observation. *Both space-time predictor and volume flux are not required anymore after the substeps* `extrapolate` *and* `volumeIntegral`.

Motivated by this observation, I transform Algorithm 8.1 and Algorithm 8.2 into low-storage algorithms by applying two techniques to the ADER-DG task graph:

Technique 1. *Move tasks from one logical step into an earlier step such that they are run as early as possible.*

Technique 2. *Store face-centered data instead of cell-centered data.*

I apply Technique 1 by moving the `volumeIntegral` and `extrapolate` substeps into the predictor phase/loop. This technique is data-centric as it helps to avoid capacity and conflict cache misses and register spilling. Moving the `extrapolate` substep out of the Riemann loop requires to store the boundary-extrapolated space-time predictor and the normal component of the space-time volume flux per cell. As the corresponding cell-centered quantities q_h^* and $F(q_h^*)$ are not required anymore after running the triad of `predictor`, `volumeIntegral`, and `extrapolate`, they can be deallocated. This realises Technique 2.

The calculation of the surface integral in the ADER-DG method's corrector step requires to integrate the (approximate) Riemann solution in time and over a cell's boundary. If the Riemann solver acts linearly on its inputs, i.e. the boundary-extrapolated space-time predictor and normal fluxes, the time integral can be directly applied to these inputs. This is a second application of Technique 1. However, note that this requires that neighbouring cells know the time stamp and time step size of their neighbours. This is the case for Algorithm 8.1 and

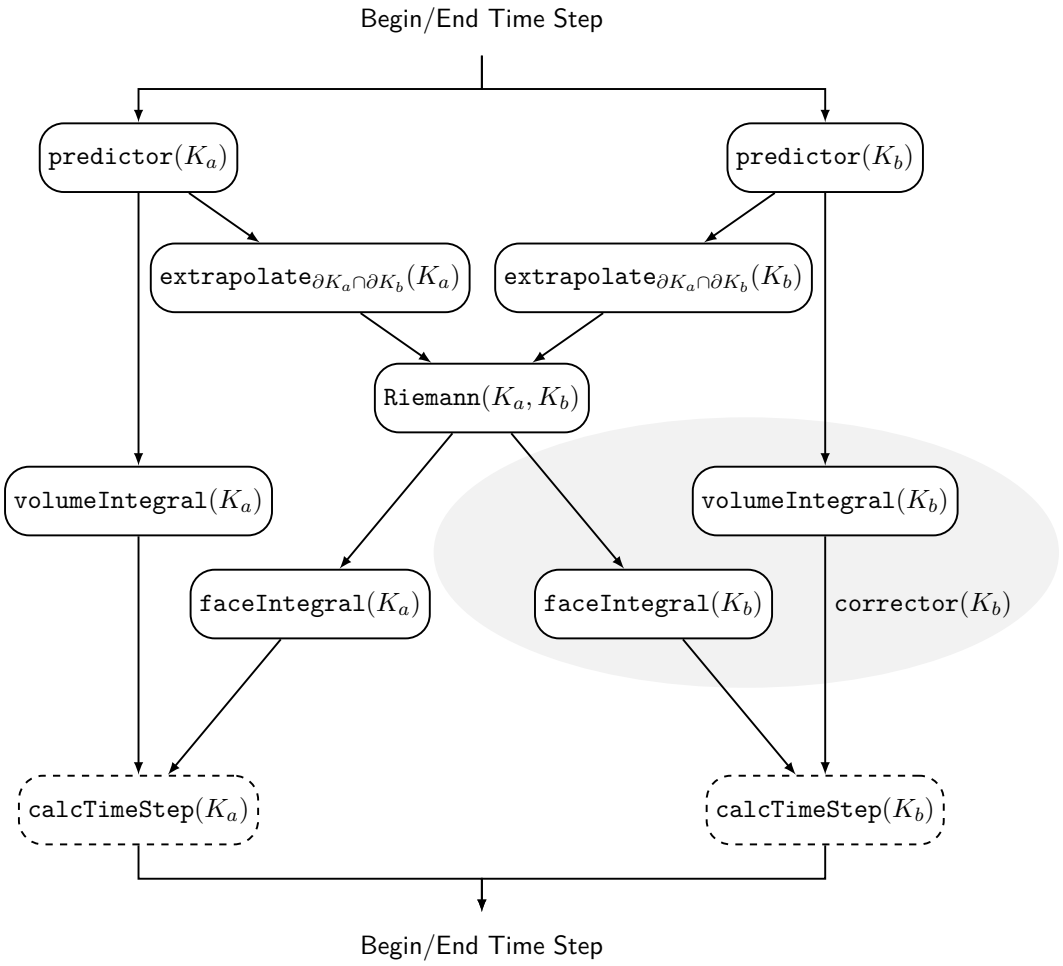


Fig. 8.2: An excerpt from the ADER-DG method's task graph, which showcases in which order tasks need to be run and where neighbouring cells need to synchronise. Only a single time step and only the tasks for two cells are shown. Imposition of boundary conditions and secondary tasks, e.g. plotting tasks, are not shown either. For brevity, only the respective cells are used to parameterise the tasks. Forking paths denote a broadcast while merging paths denote a reduction. The tasks forming the corrector step for cell K_b are drawn on top of a grey background.

Algorithm 8.2, which are both global time stepping schemes. Without a second neighbour communication step, it is in general not the case if anarchic local time stepping is used [53].

I summarise the first result of this section as follows:

Result (Low-Storage ADER-DG).

The space-time predictor $q_h^*|_K$ and the space-time volume flux $F(q_h^*)|_K$, have a combined memory footprint of

$$(d+1)\,m\,(p+1)^{d+1}$$

doubles per cell. Instead of storing these vectors, it suffices for an ADER-DG realisation to store the boundary-extrapolated space-time predictor and flux in the mesh persistently which have a combined memory footprint of

$$4\,d\,m\,(p+1)^{d}$$

doubles per cell. This footprint can be reduced to

$$4\,d\,m\,(p+1)^{d-1}$$

doubles per cell if the Riemann solver acts linearly on its inputs and if cells know the time stamp and time step size of their neighbours before the predictor substep.

Next, I apply two communication-avoiding techniques on top of the storage minimisation. The mesh traversals performing the tasks `calcTimeStep`, `Riemann`, and `faceIntegral` can be merged: Whenever the mesh traversal enters a cell to perform the `calcTimeStep` task, it analyses all of the cell's $2\,d$ adjacent faces whether they have been accessed before. For those that have not yet been accessed, it performs the `Riemann` and `faceIntegral` tasks. This realises the third technique:

Technique 3. *Run different tasks on different grid entities concurrently.*

Algorithm 8.2 is one particular way to realise the ADER-DG task graph (Fig. 8.2). An algorithm could also start each time step with the Riemann loop, continue with the corrector

loop, and finish with the predictor loop. Of course, it then requires a kick-off predictor loop before the first time step. This yields the fourth technique:

Technique 4. *Shift the tasks by half a step.*

Technique 3 and 4 make the solve require $2\,N + 1$ loops in total. The second and final result of this section is the following low-storage ADER-DG algorithm that applies Technique 1,2,3, and 4:

Algorithm 8.3 (Low-Storage ADER-DG)**.** *Low-storage variant of Algorithm 8.2 applying Techniques 1 – 4. The predictor loop is run last (blue) and the Riemann loop first per time step. The scheme requires a kick-off predictor loop (also blue). The* `faceIntegral` *sweep has been merged into the loop computing the next time step size (red). The* `volumeIntegral` *and* `extrapolate` *substeps are inserted directly after the* `predictor` *substep and thus, a cell's cell-centered space-time data can be directly deallocated after the* `volumeIntegral` *This comes at the cost of allocating* $2\,d$ *extrapolated space-time predictor and normal flux arrays, i.e. one per face and* $4\,d$ *arrays in total (green).*

```
1  T ← 0
2  ΔT ← initialiseADERDG()
3  begin neighbour exchange
4  prediction()                                              # kick off time stepping
5
6  while T < T_final do
7    ΔT_adm ← ∞
8    for cell K ∈ 𝒯 do                          # fused face integral and time step calc. loop
9      read q_h(·, T + ΔT)|_K (if not already cached)
10     for K_b ∈ 𝒯 : face−connected to K do                  # Riemann and face integral
11       if interface K ∩ K_b touched first time then
12         read q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b} (if not already cached)
13         read q_h(·, T + ΔT)|_{K_b}, q*_h|_{K_b}|_{∂K∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K∩∂K_b} (if not already cached)
14         G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{∂K∩∂K_b}, F(q*_h)|_{∂K∩∂K_b},
15                                     q*_h|_{∂K∩∂K_b}, F(q*_h)|_{∂K∩∂K_b}, n_K, n_{K_b}, n, ΔT)
16         q_h(·, T + ΔT)|_K += faceIntegral(G(q*,+_h, q*,−_h) (n · n_K), ∂K ∩ ∂K_b, ΔT)
17         q_h(·, T + ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h) (n · n_{K_b}), ∂K ∩ ∂K_b, ΔT)
18       end if
19     end for
20     ΔT_adm ← min{ΔT_adm, calcTimeStep(q_h(·, T + ΔT)|_K)}
21     write q_h(·, T + ΔT)|_K, q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
22   end for
```

```
23    end neighbour exchange
24    T ← T + ΔT
25    begin global reduction (ΔT_adm) end reduction
26    ΔT ← min{ΔT_adm, T_final − T}
27    if ΔT > 0 then
28      begin neighbour exchange
29      PREDICTION()
30    end if
31  end while
32
33  function PREDICTION()
34    for K ∈ 𝒯 do
35      read q_h(·,T)|_K
36      ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T)|_K)
37      for K_b ∈ 𝒯 : face−connected to K do
38        read q*_h|_K|_{∂K∩∂K_b}, F(q*_h)|_K|_{∂K∩∂K_b}
39        ( q*_h|_K|_{∂K∩∂K_b}, F(q*_h)|_K|_{∂K∩∂K_b} ) ← extrapolate_{∂K∩∂K_b}(q*_h|_K, F(q*_h)|_K)
40        write q_h(·,T)|_K, q*_h|_K|_{∂K∩∂K_b}, F(q*_h)|_K|_{∂K∩∂K_b}
41      end for
42      q_h(·,T+ΔT)|_K ← q_h(·,T)|_K + volumeIntegral(F(q*_h)|_K)
    # next volume integral; reuse q_h(·,T)|_K
43      write q_h(·,T+ΔT)|_K
44    end for
45  end function
```

8.3 Communication-Avoiding ADER-DG

This section presents communication-avoiding ADER-DG variants that reduce the memory reads and writes of the low-storage ADER-DG scheme further. I first consider the simplified scenario where the time step size remains invariant throughout the simulation or in between mesh adaptations. In this case, the predictor loop can be straightforwardly fused with the corrector loop, i.e. only a single loop over the mesh is required. If the resulting scheme is realised on top of a mesh traversal that exhibits good spatial and temporal locality [101] and the cache is sized appropriately, then this increases the chance that a cell's solution and auxiliary fields are loaded only once into the caches per time step. The first result of this section is:

Result (Fused Invariant-Time-Step-Size ADER-DG).

If the time step size remains invariant throughout a simulation or between mesh adaptions, then the ADER-DG time step can be realised with a single loop over the mesh that fuses the corrector and predictor loop of the low-storage ADER-DG method, i.e. the solution data is read only once per time step.

A time step of this algorithm is given below:

Algorithm 8.4 (Fused Invariant-Time-Step-Size ADER-DG)**.** *Using Techniques 1 – 5, all phases of Algorithm 8.1, where the time step size is invariant, are merged into a single loop.*

```
1  function fusedTimeStepForInvariantTimeStepSize()
2    begin neighbour exchange (if not already started)
3    for cell K ∈ 𝒯 do                                   # a single cell−wise traversal
4      read q_h(·, T + ΔT)|_K (if not already cached)
5      for K_b ∈ 𝒯 : face−connected to K do              # Riemann and face integral
6        if interface K ∩ K_b touched first time then
7          read q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b} (if not already cached)
8          read q_h(·, T + ΔT)|_{K_b}, q*_h|_{K_b}|_{∂K∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K∩∂K_b} (if not already cached)
9          G(q*,+_h, q*,−_h) ← Riemann(q*_h|_K|_{∂K}|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K}|_{∂K∩∂K_b},
10                                     q*_h|_{K_b}|_{∂K_b}|_{∂K∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_b}|_{∂K∩∂K_b}, n_K, n_{K_b}, n, ΔT)
11         q_h(·, T + ΔT)|_K += faceIntegral(G(q*,+_h, q*,−_h) (n · n_K), ∂K ∩ ∂K_b, ΔT)
12         q_h(·, T + ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h) (n · n_{K_b}), ∂K ∩ ∂K_b, ΔT)
13       end if
14     end for
15     ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·, T + ΔT), ΔT)   # use fixed known time step size
16     for K_b ∈ 𝒯 : face−connected to K do
17       ( q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b} ) ← extrapolate_{∂K∩∂K_b}(q*_h|_K, F(q*_h)|_K)
18       write q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
19     end for
20     q_h(·, T + ΔT)|_K ← q_h(·, T)|_K + volumeIntegral(F(q*_h)|_K)
   # anticipate next correction
21     write q_h(·, T + ΔT)|_{K_a}
22   end for
23   end neighbour exchange
24 end function
```

When simulating nonlinear PDE systems, the admissible time step size ΔT_{adm} may change in

each and every time step. Hence, the time step size ΔT must be adapted continuously. Global adaptive time stepping and classic local adaptive time stepping [25] both require the reduction and broadcast of a CFL-stable time step size. Due to this a synchronisation mechanism, a fifth technique is required to rephrase the ADER-DG method as a single-touch algorithm. In this case, I advise to

Technique 5 *Hope for the best (but prepare for the worst).*

I propose to construct an estimate ΔT^* for the next time step size based on the current time step size ΔT. With this estimate, the `predictor` substep can be run directly after the substeps of the corrector step. A moving average that approaches the admissible time step size or choosing the time step size slightly smaller than the previous admissible time step size, have shown to be good time step size estimators for many problems.

The observation that motivates Technique 5 is that wave speeds do not change dramatically during a simulation from one time step to the other. Still, any estimate may violate the CFL condition. Therefore, a scheme applying Technique 5 may need to rerun the `predictor` substep in a number of time steps, which requires an additional loop over the cells. As running the `predictor` substep does not modify the solution $q_h(\cdot, T)$ but only auxiliary fields (boundary-extrapolated space-time predictor and volume flux normal), it can be run twice without causing issues. Moreover, an update vector $\Delta q_h|_K$ must be allocated per cell K as the volume integral contribution must be discarded if the time step size estimate did violate the CFL condition. The `predictor` substeps are more computationally intense than any other ADER-DG substep; however, their processing is embarrassingly parallel. A time step of the scheme is given below:

Algorithm 8.5 (Fused Variable-Time-Step-Size Time Step)**.** *ADER-DG time step for variable ΔT relying on all introduced optimisation techniques. Only a single loop over the cells is employed. The face integral is performed directly before the time step size calculation. The time step size calculation is followed by the prediction which anticipates the next time step and directly computes the volume integral. Space-time volume flux and predictor are deallocated after this step. A persistently stored update vector (green) is introduced as the volume integral contribution might be discarded if the estimated time step size is not admissible. This information is available after the time step.*

```
1  function fusedTimeStep(ΔT, ΔT*)
2    begin neighbour exchange (if not already started)
3    ΔT_adm ← ∞
4    for cell K ∈ 𝒯 do                                   # a single cell−wise traversal
5      read q_h(·, T+ΔT)|_K (if not already cached)
6      for K_b ∈ 𝒯: face−connected to K do               # Riemann and face integral
7        if interface K ∩ K_b touched first time then
8          read q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b} (if not already cached)
9          read q*_h|_{K_b}|_{∂K∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K∩∂K_b} (if not already cached)
10         G(q*,+_h, q*,−_h) ← Riemann(q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b},
11                                     q*_h|_{K_b}|_{∂K∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K∩∂K_b}, n_K, n_{K_b}, n, ΔT)
12         q_h(·, T+ΔT)|_K += faceIntegral(G(q*,+_h, q*,−_h)(n · n_K), ∂K ∩ ∂K_b, ΔT)
13         read q_h(·, T+ΔT)|_{K_b} (if not already cached)
14         q_h(·, T+ΔT)|_{K_b} += faceIntegral(G(q*,+_h, q*,−_h)(n · n_{K_b}), ∂K ∩ ∂K_b, ΔT)
15       end if
16     end for
17     read Δq_h|_K
18     q_h(·, T+ΔT)_K ← q_h(·, T)|_K + Δq_h|_K
19     ΔT_adm ← min{ΔT, calcTimeStep(q_h(·, T+ΔT)|_K)}
20     ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·, T+ΔT), ΔT*)   # directly run next prediction
21     for K_b ∈ 𝒯: face−connected to K do
22       read Δq_h|_K, q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
23       ( q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b} ) ← extrapolate_{∂K∩∂K_b}(q*_h|_K, F(q*_h)|_K)
24       write q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
25     end for
26     Δq_h|_K ← volumeIntegral(F(q*_h)|_K)                # anticipate next correction
27     write q_h(·, T+ΔT)|_K, Δq_h|_K
28   end for
29   end neighbour exchange
30   begin global reduction (ΔT_adm) end global reduction
31   return ΔT_adm
```

```
32 end function
```

I propose that the FusedTimeStep loop should not be stopped as soon as the chosen time step size ΔT^* was deemed inadmissible. As the loop also runs Riemann and corrector substeps, an early termination of the loop would require rerunning these substeps, too. An early termination may further result in rippling loop restarts, where FusedTimeStep and the predictor loop are restarted and run, respectively, multiple times in a row. Once we follow this convention, i.e. complete a whole sweep and then evaluate whether one space-time predictor has violated the CFL condition, the fused ADER-DG scheme for variable ΔT requires at maximum two loops over the mesh, i.e. the solution data is at maximum read twice:

Result (Fused Variable-Time-Step-Size ADER-DG).

The ADER-DG method for variable ΔT can be realised with $1 + C \leq 2$ reads of the solution data per time step, where $C \in [0, 1]$ is the fraction of predictor reruns per simulation time steps. In particular, $C = 0$ if the admissible time step size is non-decreasing throughout the simulation.

The implementation is realised by fusing the corrector and predictor loop of the low-storage ADER-DG variant where the predictor substep uses a time step size estimate. If this estimate is not stable according to the CFL-condition, which is checked after the time step, the predictor loop is rerun.

The full algorithm that combines the FusedTimeStep loop from Algorithm 8.5 with the rerun mechanism is given below:

Algorithm 8.6 (Fused Variable-Time-Step-Size ADER-DG Time Stepping)**.** *The ADER-DG variant that applies all five optimisation techniques. If the time step size estimate was not admissible, the predictor step must be run again with a stable time step size. This stable time step size is available after the fused time step.*

```
1 T ← 0
2 ΔT ← initialiseADERDG()
3 ΔT* ← ΔT
```

```
4
5  begin neighbour exchange
6  predictionForFused(ΔT)                                   # kick off time stepping
7  while T < T_final do
8    ΔT_adm ← max{fusedTimeStep(ΔT, ΔT*), T_final − T}
9    # end neighbour exchange −− is done at the end of the function call
10   if ΔT_adm < ΔT* and T < T_final then
11     begin neighbour exchange
12     predictionForFused(ΔT)
13   end if
14   T ← T + ΔT
15   ΔT ← ΔT_adm
16   ΔT* ← α ΔT
17 end while
18
19 function predictionForFused(ΔT)
20   for K ∈ 𝒯 do                                           # prediction
21     read q_h(·,T)|_K, Δq|_K
22     ( q*_h|_K, F(q*_h)|_K ) ← predictor(q_h(·,T), ΔT)
23     for K_b ∈ 𝒯 : face−connected to K do
24       read q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
25       ( q*_h|_K|_{∂K∩∂K_b}, F(q*_h)|_K|_{∂K∩∂K_b} ) ← extrapolate_{∂K∩∂K_b}(q*_h|_K, F(q*_h)|_K)
26       write q*_h|_K|_{∂K∩∂K_b}, n · F(q*_h)|_K|_{∂K∩∂K_b}
27     end for
28     Δq_h|_K ← volumeIntegral(F(q*_h)|_K)                # anticipate next correction
29     write Δq|_K
30   end for
31 end function
```

8.4 Theoretical Comparison

Next, I compare memory footprint of access of the straightforward and the proposed low-storage and fused ADER-DG realisations. Furthermore, I identify where it makes sense to employ the fused variants over the low-storage realisation from the memory access perspective. From the annotated pseudocode, I calculate memory footprint and memory access of the low-storage ADER-DG realisations; see the algorithms in Section 8.2 and Section 8.3. In general, low-storage ADER-DG and both fused ADER-DG variants are superior to the straightforward in terms of memory footprint and access for polynomial orders $p \geq 3$; see Fig. 8.3. For fused variable-time-step-size ADER-DG, this holds for any predictor rerun

factor C as it holds for $C = 1.0$; see Fig. 8.3. This scheme uses an additional persistently stored update vector with cardinality $m \cdot (p+1)^d$ in comparison to low-storage ADER-DG and fused ADER-DG for invariant ΔT. The consequence is a larger memory footprint that is more noticeable if time-averaging is employed. Furthermore, the scheme requires a rerun of the prediction step every time the CFL condition was violated. Depending on the ratio of reruns per time step C, the memory reads and writes of fused variable-time-step-size ADER-DG might surpass those of low-storage ADER-DG; see Fig. 8.4. If boundary-extrapolated predictor and flux are time averaged on top, the break-even point decreases with the order p in favour of the low-storage scheme; see Table 8.2.

Table 8.2: The ratio of reruns per time step from which on the memory access of the fused variable-time-step-size ADER-DG surpasses that of low-storage ADER-DG. TA indicates that boundary-extrapolated data is averaged in time.

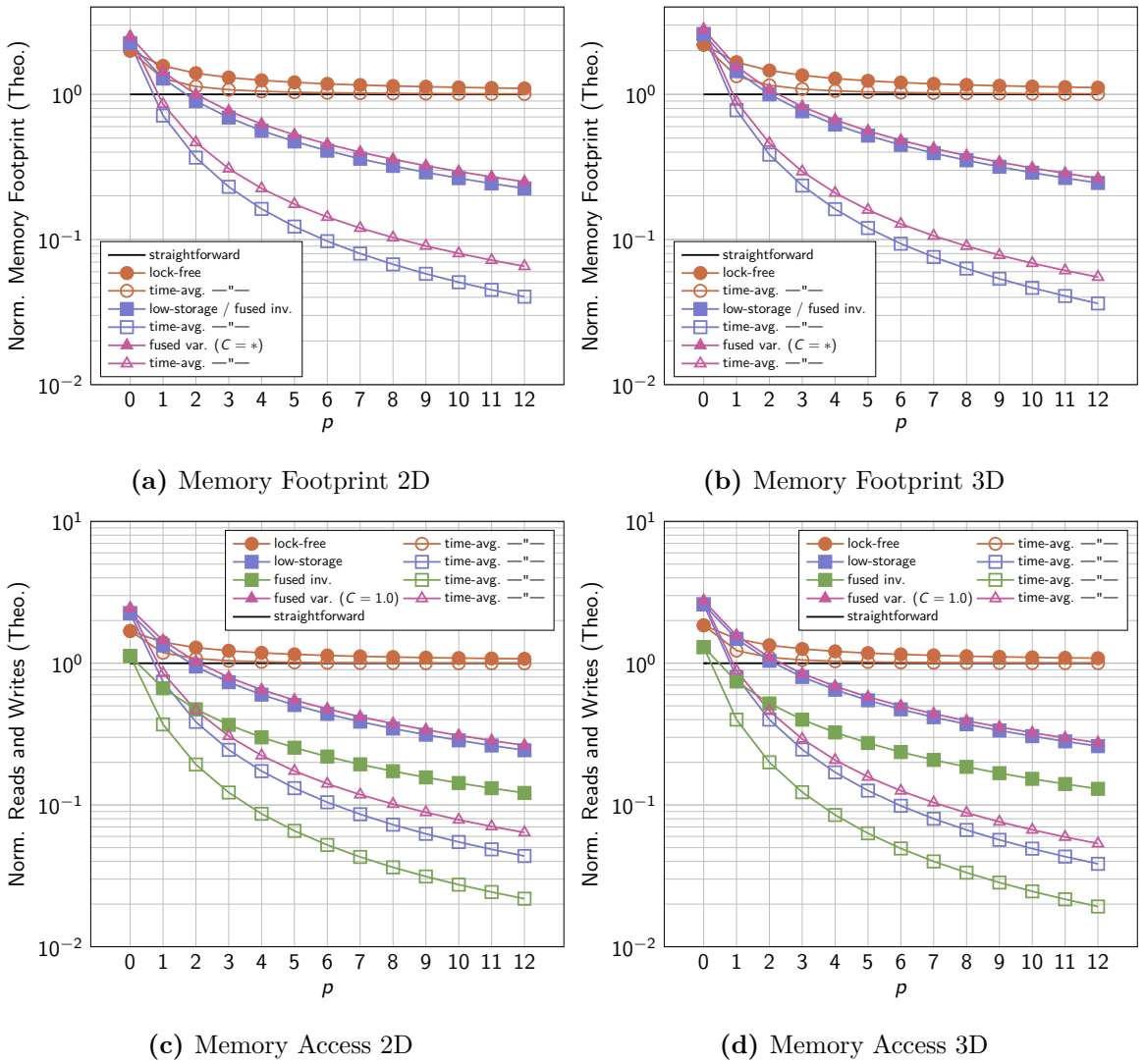
p	0	1	2	3	4	5	6	7	8	9	10	11	12
2D	84%	···	···	···	···	···	···	···	···	···	···	···	84%
2D, TA	84%	73%	64%	57%	52%	47%	43%	40%	37%	35%	33%	31%	29%
3D	89%	···	···	···	···	···	···	···	···	···	···	···	89%
3D, TA	89%	80%	73%	67%	62%	57%	53%	50%	47%	44%	42%	40%	38%

8.5 Experimental Comparison

The theoretical memory analysis reveals that low-storage ADER-DG and fused invariant-time-step-size ADER-DG are superior to the straightforward realisation for all approximation orders $p \geq 3$. The picture is less clear for fused variable-time-step-size ADER-DG as this scheme allocates an additional update vector and may need to perform a rerun of the predictor step in a number of time steps. Moreover, the analysis reveals that the impact of these modifications is more severe when time-averaging of boundary-extrapolated data is employed. In this section, I try to answer the question whether fused variable-time-step-size ADER-DG time stepping can improve performance in practice.

Correctness

Before running performance studies, I validated that the implementation of all presented methods is correct via a compressible Euler convergence test from [65]. In this benchmark, a



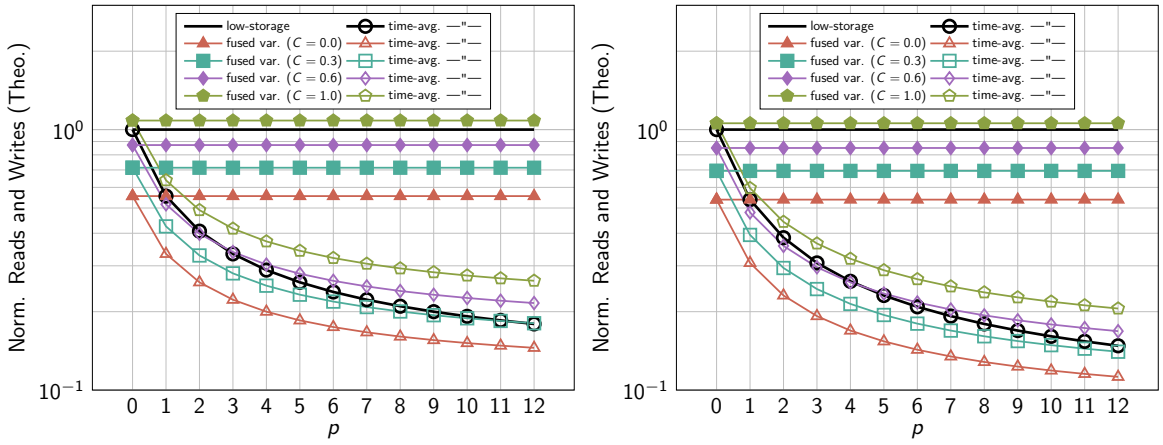
(a) Memory Footprint 2D

(b) Memory Footprint 3D

(c) Memory Access 2D

(d) Memory Access 3D

Fig. 8.3: (a) – (b): Theoretical 2D and 3D memory footprint of the proposed ADER-DG realisations in relation to the straightforward realisation and the lock-free realisation from Section 9. (c) – (d): Theoretical memory reads and writes of the proposed ADER-DG realisations in relation to the straightforward realisation. The worst-case ($C = 1.0$) is shown for fused variable-time-step-size ADER-DG.



(a) Memory Access vs. Reruns 2D

(b) Memory Access vs. Reruns 3D

Fig. 8.4: (a) – (b): Theoretical memory access of the 2D and 3D fused variable-time-step-size ADER-DG realisation vs. low-storage ADER-DG. Different polynomial orders p and numbers of reruns per time step C are considered.

constant velocity field advects a sinusoidal density profile:

$$\rho(x,y,t) = 1 + 0.1 \cdot \sin(\pi \cdot ((x - v_x\, t) + (y - v_y\, t))),$$
$$v_x = 2.5,$$
$$v_y = 2.4.$$

The background pressure of the ideal gas remains invariant throughout the simulation, $P = 1$. I run the simulation till $T = 1.0$ non-dimensional time and use the analytical solution as Dirichlet boundary conditions. Both the low-storage and the fused variable-time-step-size ADER-DG method converged with optimal rates in this test; see Fig. 8.5. The latter used time step size estimates of the form $\Delta T^* = \alpha\, \Delta T$, where the time-step-size-under-estimation factor α was chosen as $\alpha \in \{0.5, 0.7, 0.99\}$. The choice of α appears not to have a significant impact on the discretisation errors in this test. However, I emphasise that the test used Dirichlet boundary conditions as ExaHyPE does currently not support periodic boundary conditions. The latter are better suited to investigate the numerical dissipation of numerical methods.

Hardware and Build Environment

I ran my performance studies on up to 4 nodes of LRZ's SuperMUC Phase 2 [80]. Each node is equipped with 64 GB of RAM (random access memory) and two Intel Xeon processors of

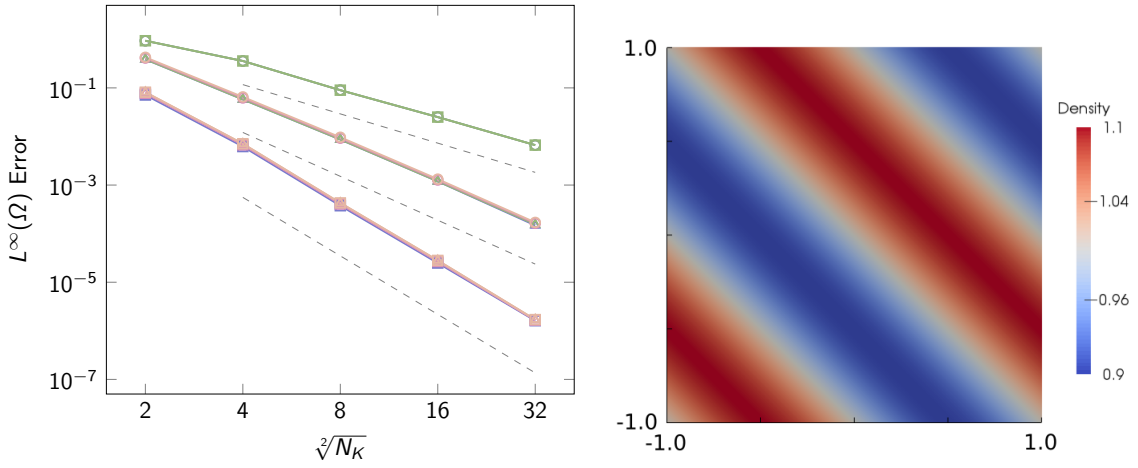


Fig. 8.5: Left: Discretisation errors measured in the $L^\infty(\Omega)$ norm for the low-storage ADER-DG scheme and the fused scheme for variable ΔT. The latter was tested with a time step size estimate of the form $\Delta T^* = \alpha\,\Delta T$, where $\alpha \in \{0.5, 0.7, 0.99\}$. The dashed lines indicate the theoretical convergence with rate 2, 3, and 4 for orders $p = 1$, $p = 2$, and $p = 3$, respectively. Right: Initial solution of the entropy wave benchmark problem.

type E5-2697 v3, which run their 14 cores at 2.6 GHz. ExaHyPE was configured to switch off Peano's recursion unrolling feature (see Section 9.1) as this feature does not necessarily preserve the locality properties of the mesh traversal. Turning off the feature makes Peano traverse the mesh along the space-filling curve, which interweaves the access of cells with that of their adjacent faces. Otherwise, Peano might extract regular subgrids where it traverses all faces before the cells.

Software	Version or Revision
ExaHyPE (git branch and revision)	master – e5249766
ExaSeis (git branch and revision)	master – 826b554d
Peano (git branch and revision)	master – fa5dfda4
Intel Compiler	18.0.2 20180210
GNU Compiler	5.4.0
IBM MPI	1.4

The applications have been compiled according to the following ExaHyPE-specific environment variables:

- `EXAHYPE_CC=mpiicpc`

- `COMPILER=Intel`
- `MODE=Release`
- `DISTRIBUTEDMEM=MPI`
- `SHAREDMEM=None`
- `USE_IPO=on`

All experiments were performed with optimised ADER-DG kernels as generated by the ExaHyPE toolkit, which tailor all vectorisation to the `AVX2` instructions of the Intel Xeon processors. On top, I used the Intel compiler with its most aggressive optimisation. I configured it to eliminate expensive virtual function calls within the compute-intense routines.

Performance Comparison

The first experiment used ExaSeis's linear elastic wave equations solver that employs geometry-aligned meshes to approximate the material distribution and the Cauchy-Kowalewsky procedure to construct the space-time predictor. I ran the LOH.1 benchmark [2] with this solver. The second experiment simulated an entropy wave, a sinusoidal density perturbation moving in a constant velocity field, with the compressible Euler equations ($m = 5$). The nonlinear PDE system was discretised in time with the local space-time DG procedure [52]. The procedure requires more Picard iterations in the vicinity of strong gradients of the solution. As the time step size remains constant during the simulation, fused variable-time-step-size ADER-DG requires no reruns in this test and fused invariant-time-step-size ADER-DG can also be employed. Per considered polynomial order and mesh, I ran the experiments once in serial and once with 28 MPI (Message Passing Interface) ranks.

In both experiments, the fused methods obtain a robust speedup over low-storage ADER-DG for most p and mesh sizes except for the Euler 3D test with $p = 7$; see Fig. 8.6 and Fig. 8.7. The experimental data implies that the fully-fused schemes pays off the most for small meshes and lower polynomial orders. The largest speedups have been measured in the MPI experiments. The optimisation is tailored towards upscaling.

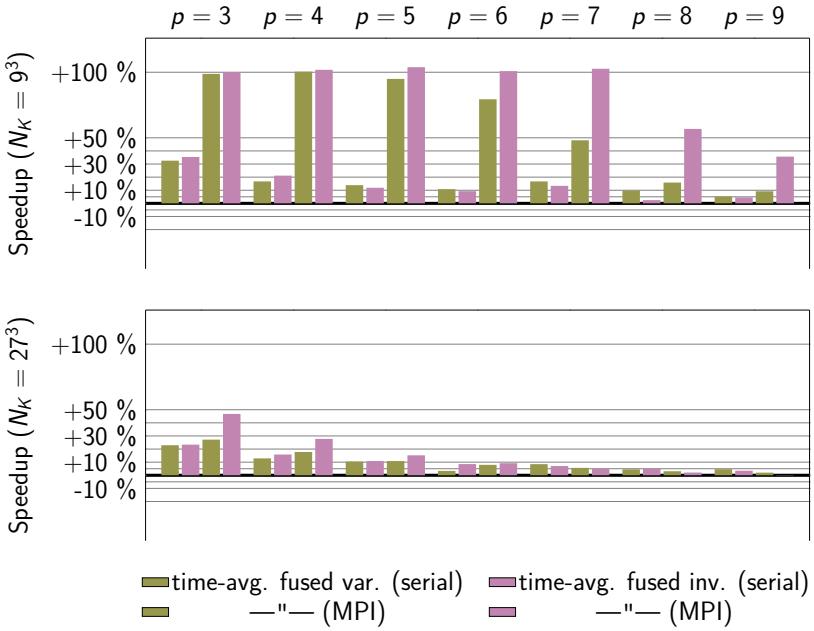


Fig. 8.6: Speedup of the two fused ADER-DG vs. low-storage ADER-DG for the linear elastic wave equations benchmark. In the MPI experiments, the computational work was distributed among 27 ranks; one additional rank was used to perform administrative tasks.

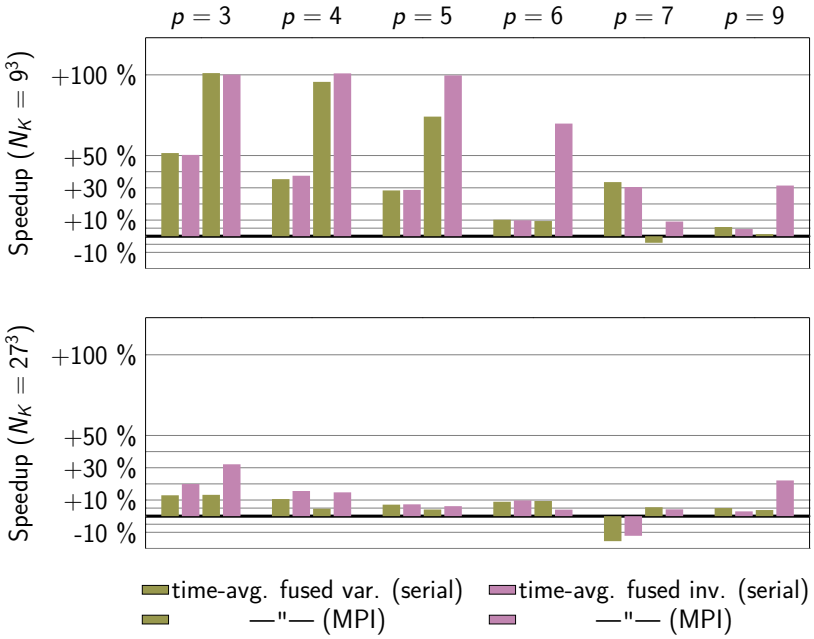


Fig. 8.7: Speedup of the two fused ADER-DG vs. low-storage ADER-DG for the nonlinear compressible Euler test problem. In the MPI experiments, the computational work was distributed among 27 ranks; one additional rank was used to perform administrative tasks.

8.6 Discussion

In this chapter, I presented efficient realisations of ADER-DG in terms of memory footprint and memory access. The memory footprint of straightforward ADER-DG realisations is dominated by space-time predictor and volume flux fields, which have a memory footprint that is proportional to $(d+1) \cdot (p+1)^{d+1}$. I proposed to store the boundary-extrapolated predictor and normal flux instead. They have a memory footprint that is proportional to $4\,d \cdot (p+1)^d$. If the boundary-extrapolated data is additionally averaged in time, the saving in storage is even more significant: For polynomial orders $p \geq 7$, an ADER-DG scheme that applies both optimisations requires more than $10\times$ less memory compared to a straightforward realisation. Memory reads and writes are reduced by roughly the same factor. They key idea for this optimisation is to move the volume integral and boundary-extrapolation substeps of the ADER-DG method into the earlier predictor phase.

On top of the storage optimisation, I investigated fusing all algorithmic phase of the ADER-DG method to minimise data movement further. The key idea is to directly run the predictor after the corrector step. For the linear ADER-DG scheme, where the time step size remains constant as long as the mesh does not change, this was an straightforward exercise that is able to reduce memory access by a further margin. The task is more difficult for the ADER-DG method for nonlinear PDE systems. Adaptive time stepping variants reduce an admissible time step size after every time step. Wave speeds–and thus the global admissible time step size–can change from time step to time step when simulating nonlinear PDE systems. To realise a fully-fused variant of ADER-DG also in this case, I proposed an optimistic approach where the time step size for the next predictor computation is estimated based on the previous value. Per cell, the predictor is then run with this estimate directly after the corrector step. After the time step, the maximum wave speed in the domain is available and the algorithm checks if the used estimated time step size is stable with respect to the CFL condition. If this is not the case, the computationally costly but embarrassingly parallel predictor phase of the ADER-DG scheme is rerun. I supplied analysis to determine up to what number of predictor reruns the scheme still yields an performance improvement over the low-storage ADER-DG method.

In the last part of the chapter, I provided experimental evidence for the correctness of the

methods. Furthermore, I showed experimental evidence where the fused ADER-DG variants yield a robust speedup over the low-storage ADER-DG scheme. This speedup is more pronounced for lower approximation orders and coarser meshes. Arithmetic intensity is lower in the first case, i.e. the code is more memory sensitive. As the fused ADER-DG realisations use half as many loops over the mesh as low-storage ADER-DG in the best case, they are less affected by overhead linked to the mesh traversal.

Limitations of the Presented Analysis

The theoretical analysis that I present in this chapter only focuses on memory aspects. A comprehensive analysis of the fully-fused ADER-DG variant for variable ΔT should take the computational cost into account, too. Moreover, the additional numerical dissipation of the scheme due to the time step size underestimation should be investigated more carefully.

9

Hybrid Parallelisation

In this chapter, I discuss how ExaHyPE's algorithms leverage multi-core CPUs to parallelise computation and to hide communication. Modern supercomputers connect multiple single *compute nodes* via a network. Each compute node hosts one or more multi-core CPUs. Therefore, modern supercomputers support (at least) three levels of parallelism. First, every CPU core has a number of large SIMD registers to apply the same instruction simultaneously on multiple data values. Second, each compute node's processors have multiple CPU cores to process multi-threaded programs in parallel on the compute node. Third, the supercomputer's network provides the communication infrastructure to distribute substeps of an application among multiple nodes. A program must leverage all of them for optimal performance. Distributed-memory computing allows running applications that require more memory than available on a single compute node. In regimes where communication cost is low compared to computational work, distributed-memory computing can be used to speedup program execution, too. This chapter presents ExaHyPE's hybrid distributed-memory shared-memory parallelisation, which is realised via Peano and its technical infrastructure that rely on MPI and TBB (Threading Building Blocks) internally. While Chapter 6 discusses Peano's spacetree partitioning procedure, which introduces distributed-memory concurrency into ExaHyPE, this chapter is concerned with introducing shared-memory concurrency into the code and how

this interplays with the spacetree partitioning.

Contributions

I propose a task-based parallelisation that introduces shared-memory concurrency into MPI-only PDE solvers [38]. A concurrent priority queue ensures that critical solver operations along the boundary to a remote rank are processed before less critical work. This allows overlapping communication with communication. I regard the invasiveness of the approach as small as targeted codes only need to express existing function calls as prioritised tasks. This chapter presents pseudocode for the implementation on top of the ADER-DG method.

Note: *This is joint work with Tobias Weinzierl and Benjamin Hazelwood. We published a preprint on aspects of this chapter [38]. My contribution is twofold: I rephrase all PDE solvers of the* ExaHyPE *engine in a task language, and I interface these tasks with* Peano*'s priority tasking system. To realise the latter, I specify which tasks have to be spawned with what priority during the spacetree traversal and where to wait for the completion of tasks – and what to do in the meantime.*

Note: *To be able to make statements about the scalability of a code requires that the code's single-core performance reasonably high relative to the theoretical peak. ExaHyPE uses highly optimised compute kernels that rely on compiler-induced SIMD parallelism and inlined assembler instructions via LIBXSMM [68]. Jean-Matthieu Gallard develops a generic code generator that generates optimised compute kernels for the ADER-DG method and the projectors between ADER-DG and FVM solution that are required for the limiting ADER-DG method [58]. I rely on his work in this thesis. At the stage of writing this thesis, there were no optimised compute kernels available for the Godunov and MUSCL-Hancock finite volume methods in* ExaHyPE*. Hence, I solely relied on automatic compiler optimisations in the experiments discussed in this thesis. In the meantime, optimised FVM kernels have been contributed to the* ExaHyPE *repository.*

Structure

In the first section of this chapter, I detail how ExaHyPE introduces shared-memory concurrency into its numerical methods via Peano's recursion unrolling technique. This parallelisation is well-suited for parallelising regular spacetree partitions. If the spacetree partitions are highly adaptive or small, the technique is less effective. Small, very adaptive mesh partitions

per process are common for high-order discretisations such as ADER-DG, which have a large memory footprint per cell. Therefore, I introduce additional shared-memory concurrency into ExaHyPE's algorithms via Peano's tasking infrastructure (Section 9.2). Operations associated with the mesh cells are not directly run by the thread(s) traversing the mesh. Instead, they are put into a background queue. The traversal becomes a *producer* thread or multiple producer threads where it is run in parallel. Other *consumer* threads consistently check the queue for new tasks and run the ones they find. A priority system ensures that time critical operations whose results serve as input to inter-process communication or inter-grid transfer operations are completed first. Peano's existing tasking infrastructure is extended for this purpose. Furthermore, I identify local synchronisation points where the traversal threads have to wait for the completion of tasks before they can continue. During the wait time, they progress MPI communication or steal tasks of the type that hinder their progress. The presented scheme does not introduce global synchronisation points. Hence, work from previous algorithmic phases can be overlapped with the phase that is currently run by the traversal threads. The chapter concludes with experimental evidence (Section 9.3) and a discussion (Section 9.4).

9.1 Hybrid Parallelisation via Recursion Unrolling

ExaHyPE is built upon the Peano framework. Peano provides distributed-memory and shared-memory concurrency "out of the box". It further abstracts the corresponding programming models such that they can be used via pre-generated user callbacks. While Peano realises a distributed-memory parallelisation via its spacetree partitioning procedure, it can instrument its user applications with shared-memory concurrency in two ways:

1. Peano provides a mechanism for identifying regular subgrids in the adaptive mesh [55][102]. It can then be instrumented to run parallel for-loops on these subgrids, while the code by default uses a recursive depth-first traversal of the spacetree otherwise. Data races are prevented either via semaphores or colouring. The implementational burden for Peano user applications to use this feature is very low.

2. Peano comes with tasking infrastructure. User applications such as ExaHyPE are then required to actively identify tasks and to delegate them to the tasking infrastructure of Peano. The latter then ensures that the tasks are run by the appropriate multi-

threading backend.

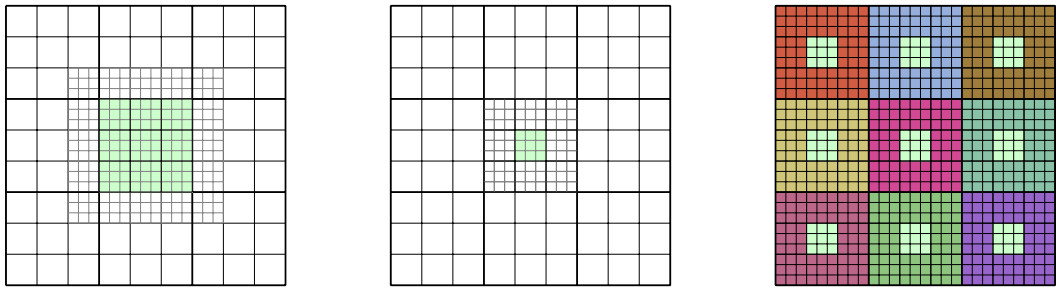
The internals of Peano's shared-memory parallelisation can be configured to use TBB, OpenMP (Open Multi-Processing), or C++ threads. Compared to previous Peano releases [105], this thesis extends Peano's tasking infrastructure to take priorities into account [38].

9.1.1 Recursion Unrolling

A shared-memory parallelisation of ExaHyPE applications can be realised via Peano's recursion unrolling feature [55]. This feature detects regular mesh regions in the spacetree that can be processed in parallel. Regular mesh regions found by the feature always consist of $3^{d \cdot l}$ cells where l is the depth of the corresponding regular subtree (Fig. 9.1). Wherever Peano's spacetree traversal automaton encounters a regular subtree during the top-down part of the traversal, it does not descend into the subtree cell by cell. Instead, it invokes a parallel for loop for each level in the subtree starting from the coarsest level. During the bottom-up traversal part, the traversal automaton runs the level-wise parallel for loops in reversed order. Peano applications (such as ExaHyPE) plug into Peano's spacetree traversal via adapter classes called mappings. These mappings subscribe to traversal events triggered by the traversal automaton. For example, whenever the automaton loads a cell, it invokes the `enterCell` method of all mappings registered for the current spacetree traversal. Peano applications can request that spacetree traversal events are processed concurrently in regular subtrees. Recursion unrolling, i.e. treating regular regions within the mesh as Cartesian array traversed by parallel for loops, has high impact for setups where the mesh is topologically smooth [104]. In the context of hyperbolic PDE solvers, this holds for example for many finite volumes schemes [97].

9.1.2 Multi-Threaded Finite Volumes and ADER-DG

All algorithmic phases of ExaHyPE's finite volumes solvers are embarrassingly parallel. This also applies to the neighbour exchange loop (see Algorithm 2.2). During the neighbour exchange loop, the solvers copy boundary layers from one patch to the other's ghost layers. At no time, two threads write to the same ghost values, or read from the same boundary layers. No data race is possible. The predictor and corrector step of the ADER-DG method are embarrassingly parallel, too. However, the Riemann phase of the ADER-DG implementation in Algorithm 2.3 is not. Here, the Riemann solution is added directly to the solution vector of two neighbouring cells via the `faceIntegral` operation. Data races might occur if multiple



(a) Adaptive Mesh 1 **(b)** Adaptive Mesh 2 **(c)** Partitioned Mesh

Fig. 9.1: Mesh regularity and spacetree partitioning impact recursion unrolling. (a) The recursion unrolling feature detects one regular grid of size $3^{2 \cdot d}$ at the center of the mesh. (b) Recursion unrolling detects one subgrid of size $3^{1 \cdot d}$ for a slightly different mesh. (c) A regular mesh is distributed among 9 processes. They all border the global master at the domain boundary. Recursion unrolling detects one subgrid of size $3^{1 \cdot d}$ on each mesh partition.

Riemann solves are processed in parallel by different threads. They might overwrite each others contribution to a cell's solution vector.

Sequentialising write access to a cell's solution vector prevents data races during the Riemann solve loop in Algorithm 2.3. This can be accomplished with a locking mechanism: The first thread that tries to compute a face integral acquires a lock, performs the face integral, and then frees the lock again. Other threads have to wait until the first thread is done, which is signalled by the freed lock, before they can run a face integral that writes to the same solution vector. The performance gain achievable with this implementation depends on how time demanding the face integral is, i.e. how long the sequentialised code section is. An alternative implementation allocates additional arrays per face of the ADER-DG cell to store the Riemann solution for that face. While the Riemann problems are still solved during the loop over the faces, the face integral can be moved into the loop over the cells that runs the volume integral; Applying these modifications to Algorithm 2.3 results in Algorithm 9.1, which maps ADER-DG to three Peano spacetree traversals and fits perfectly to the recursion unrolling paradigm. I will focus on this algorithm in the remainder of this chapter. However, note that the optimised ADER-DG algorithms presented in Chapter 8 store Riemann data per face too and thus do not need locks either.

Notation. *The algorithms in this chapter only consider leaf cells, i.e. cells that hold an*

ADER-DG solution or finite volumes patch.

Algorithm 9.1 (A Lock-Free ADER-DG Scheme)**.** *The Riemann solve result is written back to the boundary flux arrays (blue) and the face integrals are moved into the volume integral loop (green). An invariant time step size is assumed.*

```
1  T ← 0
2  initialiseADERDG( )
3
4  while T < T_final do
5    for cell K ∈ 𝒯 do
6      ( q_h^*|_K, F(q_h^*)|_K ) ← predictor(q_h(·,T)|_K, ΔT)
7      q_h(·,T+ΔT)|_K ← q_h(·,T)|_K
8    end for
9    for face−connected cells K_a, K_b ∈ 𝒯 do
10     ( q_h^*|_{K_a}|_{∂K_a∩∂K_b}, n·F(q_h^*)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q_h^*|_{K_a}, F(q_h^*)|_{K_a})
11     ( q_h^*|_{K_b}|_{∂K_b∩∂K_b}, n·F(q_h^*)|_{K_b}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q_h^*|_{K_b}, F(q_h^*)|_{K_b})
12     G(q_h^{*,+}, q_h^{*,−}) ← Riemann(q_h^*|_{K_a}|_{∂K_a∩∂K_b}, n·F(q_h^*)|_{K_a}|_{∂K_a∩∂K_b},
13                                      q_h^*|_{K_b}|_{∂K_a∩∂K_b}, n·F(q_h^*)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
14     n·F(q_h^*)|_{K_a}|_{∂K_a∩∂K_b} ← G(q_h^{*,+}, q_h^{*,−})
15     n·F(q_h^*)|_{K_b}|_{∂K_a∩∂K_b} ← G(q_h^{*,+}, q_h^{*,−})
16   end for
17   for cell K ∈ 𝒯 do
18     for K_b ∈ 𝒯 : face−connected to K do
19       q_h(·,T+ΔT)|_{K_a} += faceIntegral(n·F(q_h^*)|_K|_{∂K∩∂K_b}, ∂K∩∂K_b, ΔT)
20     end for
21     q_h(·,T+ΔT)|_K += volumeIntegral (F(q_h^*)|_K, ΔT)
22   end for
23   T ← T + ΔT
24 end while
25
26 function initialiseADERDG( )
27   for cell K ∈ 𝒯 do
28     q_h(·,0)|_K ← represent q(·,0)|_K as polynomial
29   end for
30 end function
```

Limited Concurrency for High-Order Methods

Peano's recursion unrolling technique excludes cells along spacetree partition boundaries from the search for regular mesh regions as messages along partition boundaries must be sent

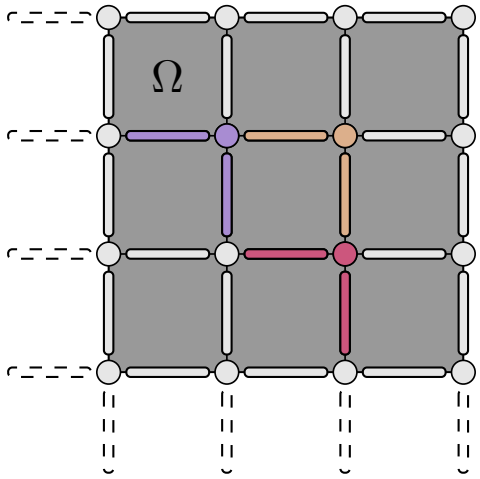


Fig. 9.2: Riemann solves between ADER-DG cells must be performed via the vertices in Peano as no faces exist as grid entities. In ExaHyPE's ADER-DG implementation, every vertex only performs a Riemann solve for the faces to its left and below (highlighted). Due the lock-free implementation, all vertices can be processed in parallel without the chance of data races. Faces that lie outside of the domain are skipped by the processing thread.

out and received in order. In addition, the order of fine grid and coarse grid operations at mesh resolution transitions is important for the implementation of a number of algorithms. Therefore, Peano also excludes fine grid and coarse grid cells along mesh resolution transitions from the search [55]. As high-order ADER-DG and patch-based finite volumes have a significant memory footprint per cell, our number of cells per partition is typically small. Due to memory limitations on commodity cluster nodes (typically around 32 – 64 GB) the spacetree partitions seldom have more than 27^3 cells in 3D. A brief thought experiment demonstrates that the recursion unrolling feature does not introduce sufficient shared-memory concurrency into ExaHyPE's algorithms if it is applied on top of a distributed-memory parallelisation: When Peano's spacetree partitioning procedure distributes the tree among a number of processes, it is common that the spacetree partition of a process is completely surrounded by partitions of other processes. In this case, there is a partition boundary on every side of the spacetree partition. Amdahl's Law clarifies that the additional speedup that can be achieved on such a mesh partition is limited; see Table 9.1. For the partition of size 27^3, we can expect a speedup of less than $2\times$ from an additional shared-memory parallelisation.

The combination of AMR plus MPI in the high-order world of ADER-DG renders the

Table 9.1: Assume a process's spacetree partition is surrounded by the partitions of other processes. The tables gives the maximum achievable additional speedup if a shared-memory parallelisation is realised via Peano's recursion unrolling technique on top of Peano's distributed-memory parallelisation.

Cells	Serially Proc. Cells 3D (2D)	Serial Fraction 3D (2D)	Maximum Speedup 3D (2D)
9^d	702 (72)	0.963 (0.889)	1.04 (1.13)
27^d	10,422 (288)	0.529 (0.395)	1.89 (2.53)
81^d	109,566 (936)	0.206 (0.143)	4.85 (7.01)
243^d	1,036,854 (2,880)	0.072 (0.049)	13.84 (20.50)
729^d	9,487,422 (8,712)	0.024 (0.016)	40.84 (61.00)

potential of recursion unrolling limited. I therefore use Peano's tasking infrastructure to improve shared-memory concurrency where recursion unrolling is not sufficient. Furthermore, my approach explicitly overlaps communication and computation.

9.2 Enclave Tasking

Peano's recursion unrolling technique introduces limited concurrency into ExaHyPE's time stepping algorithms. These run patch-based finite volumes and the high-order ADER-DG on distributed, adaptive meshes where the spacetree partition per process is small and does not exhibit much regularity. In [38], we propose a hybrid programming model, *enclave tasking*, that introduces more shared-memory concurrency under such conditions.

9.2.1 Tasking Runtime

The standard way to overlap inter-process communication and computational work is to use an additional background thread [108]. Per process, one CPU core is booked solely for exchanging data with other processes (Fig. 9.3 (a)). This core is not available for computations and mesh refinement operations. It idles while no messages come in or need to be sent. The applicability of this approach to ExaHyPE is limited as Peano's recursion unrolling technique typically does not introduce enough shared-memory concurrency into ExaHyPE's meshes. Therefore, the parallel for loops initiated by Peano's mesh traversal would employ only a fraction of the available CPU cores while the rest would idle due to insufficient work.

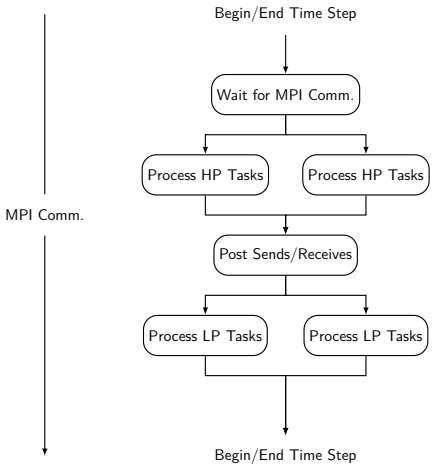
To move the computations into the background instead of the communication could be regarded

as the inverse approach to the standard approach. Here, the traversal threads instantiate tasks during the spacetree traversal and puts them into a background queue. Other consumer threads run in the background and continuously check the queue for new tasks. While the computations are performed in the background by the consumer threads (Fig. 9.3 (b)), the traversal threads run through the spacetree and perform mesh refinement operations and spawns further tasks. While the traversal threads wait for the computations to complete, they complete all inter-process communication and then start the next time step. No additional core must be booked for inter-process communication. If a simulation runs on multiple nodes, it is advantageous that tasks whose outcome are input to a communication routine are performed before other tasks. The inverse approach can be extended straightforwardly to take task priorities into account. ExaHyPE accomplishes this by storing tasks in a priority queue before delivering them to the consumer threads. When consumer threads try to take tasks from the queue, it serves them the tasks with highest priority. This sums up the description of the runtime that is required to realise the *enclave tasking* approach (Fig. 9.3 (c)). In the following, I will outline how ExaHyPE's ADER-DG implementation interfaces with this runtime.

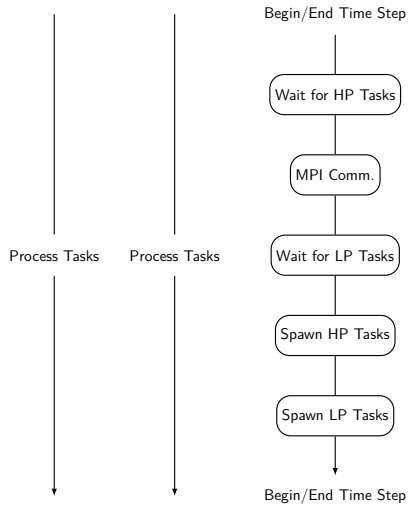
9.2.2 Enclave Tasking for ADER-DG

In this section, I wrap the original cell-wise ADER-DG operations into tasks. The producer threads spawn these tasks during the mesh traversal, i.e. they push them to the enclave tasking priority queue. They are then available to the consumer background threads that continuously check the queue. I introduce a $\texttt{PredictorTask}_K$ per leaf cell K that wraps the cell-wise `predictor` substep of the ADER-DG method plus the swapping or copying of old and new cell-wise solution (cf. Algorithm 9.1). Similarly, I introduce a $\texttt{CorrectorTask}_K$ task that wraps the ADER-DG corrector substeps `faceIntegral` and `volumeIntegral`. The space-time-predictor computation along the boundary of a process's spacetree partition must be completed before the ADER-DG solver can send out Riemann solve input data to a neighbouring process. I assign predictor tasks that are spawned from the cells along an MPI boundary a high priority (Fig. 9.4). I assign other predictor tasks a low priority.

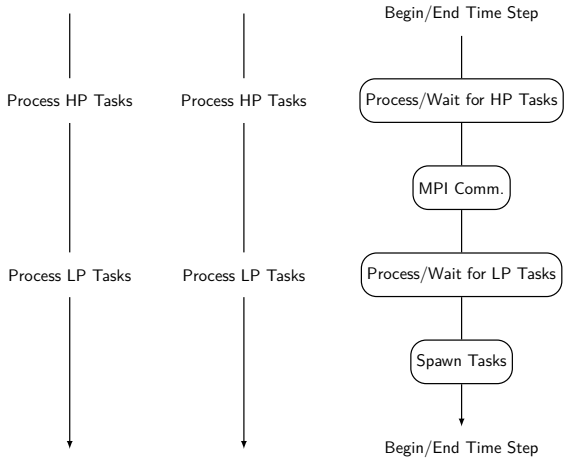
Corrector tasks have high priority, too, as their execution might overlap with the next predictor loop. They should be prioritised versus newly spawned low priority predictor tasks. In ExaHyPE's enclave tasking implementation, the face-wise Riemann problems are solved



(a) MPI Thread

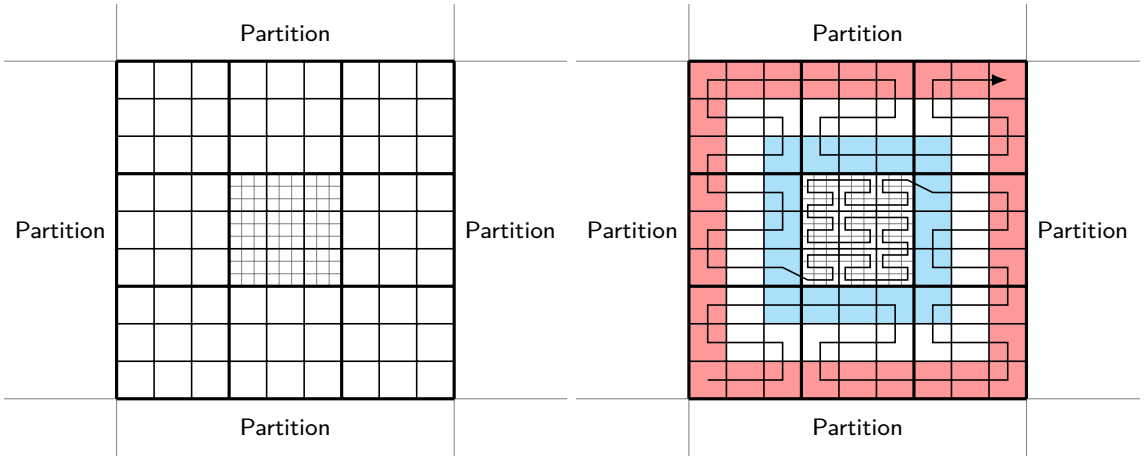


(b) Task-based



(c) Enclaves

Fig. 9.3: Hybrid parallelisation approaches sketched for a setup with 3 threads per process. (a) The standard approach that uses a communication background thread. It processes the computational work with parallel for loops. High priority tasks are processed first before the low priority ones. (b) The inverse approach that uses background threads that process tasks in any order. Therefore, the traversal thread has to spawn tasks according to an order that aligns with the task priorities. (c) The proposed enclave tasking approach where the background threads take task priorities into account. The traversal thread does not need to spawn tasks in an order that aligns with the priorities as the tasks are stored in a priority queue, which delivers high priority tasks first to the background threads. Moreover, it allows the traversal thread to steal tasks and process them itself.



(a) Adaptive Mesh **(b)** Traversal Order and Priorities

Fig. 9.4: (a) The adaptively refined mesh partition of a process is surrounded by partitions belonging to other processes. Neighbouring processes exchange data associated with the ADER-DG cells adjacent to the boundary between the partitions. (b) The spacetree traversal automaton processes cells in an immutable order along the Peano SFC. ExaHyPE employs enclave tasking to spawn prioritised tasks along the spacetree traversal. It assigns cells adjacent to the partition boundary (red) a higher priority than cells in the interior. Tasks adjacent to adaptivity boundaries (blue) are prioritised in ExaHyPE, too.

by the traversal threads. Their parallelisation relies on the recursion unrolling technique. Spawning them as tasks would introduce dependencies between the tasks that are difficult to model with priorities when algorithmic phases overlap. The Riemann solves require that the predictor task in adjacent cells has completed, while spawning a predictor task requires that the corrector task for the same cell has completed. If spacetree traversal threads would wait here, this would slow down the production of the next tasks. Traversal threads also have access to the task queue. Therefore, they can steal tasks of the same priority that they are waiting for. If a hybrid parallelisation is used, they can perform inter-process communication at these waiting points, too.

Algorithm 9.2 introduces enclave tasking to Algorithm 9.1. Per face, it places two of the aforementioned waiting mechanisms in front of the Riemann solve. One for each cell adjacent to the face. Another is placed right before the predictor tasks are spawned. Additionally, the algorithm indicates where neighbour communication begins and ends and where Riemann input data needs to be retrieved from a communication buffer.

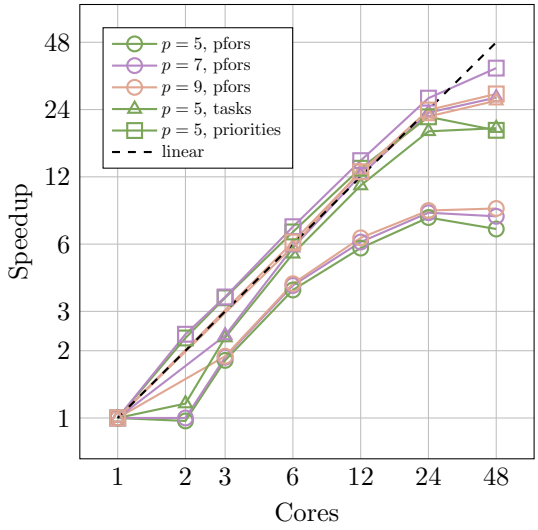
Algorithm 9.2 (Lock-Free Enclave Tasking ADER-DG)**.** *Predictor tasks are spawned as high priority if the corresponding* ADER-DG *cell is adjacent to a partition boundary or a finer grid (blue). Corrector tasks are spawned with high priority as their execution might overlap with the next predictor loop (green). Before a new predictor task can be spawned or a Riemann solve can be performed, the previous corrector or predictor task must be completed, respectively. While the main thread waits for the completion, it can steal and run tasks itself or perform inter-process communication (red). The neighbour communication phase starts with the predictor loop and ends before the Riemann solve loop. Riemann input data from a neighbour partition is stored in a communication buffer. There exist one such buffer per face along the partition boundary.*

```
1  while T < T_final do
2    begin neighbour exchange
3    for cell K ∈ 𝒯 do
4      while CorrectorTask_K not completed do
5        progress inter−process communication
6        steal and run tasks with high priority
7      end while
8      if K adjacent to partition boundary or finer grid then
```

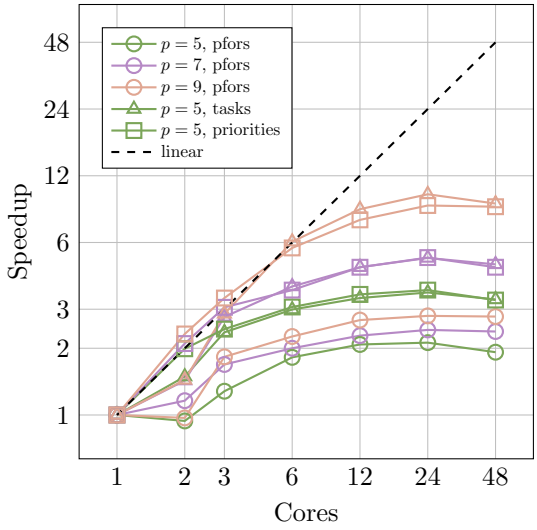
```
      spawn PredictorTask_K with high priority
    else
      spawn PredictorTask_K with low priority
    end if
  end for
  while not all high priority jobs completed do
    progress inter−process communication
    steal and run high priority tasks
  end while
  end neighbour exchange

  for face−connected cells K_a, K_b ∈ 𝒯 do
    if K_a belongs to neighbour partition then
      ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← readFromCommBuffer(∂K_a ∩ ∂K_b)
    else
      while PredictorTask_{K_a} not completed do
        progress inter−process communication
        steal and run tasks with low priority
      end while
      ( q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ) ← extrapolate_{∂K_a∩∂K_b}(q*_h|_{K_a}, F(q*_h)|_{K_a})
    end if
    if K_b belongs to neighour partition then
      # similar as above
      # ...
    end if
    G(q*,+_h, q*,−_h) ← Riemann(q*_h|_{K_a}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b},
                                q*_h|_{K_b}|_{∂K_a∩∂K_b}, n · F(q*_h)|_{K_b}|_{∂K_a∩∂K_b}, n_{K_a}, n_{K_b}, n, ΔT)
    n · F(q*_h)|_{K_a}|_{∂K_a∩∂K_b} ← G(q*,+_h, q*,−_h)
    n · F(q*_h)|_{K_b}|_{∂K_a∩∂K_b} ← G(q*,+_h, q*,−_h)
  end for

  for cell K ∈ 𝒯 do
    spawn CorrectorTask_K with high priority        # run face and volume integral
  end for
  T ← T + ΔT
end while
```

9.3 Experimental Evidence

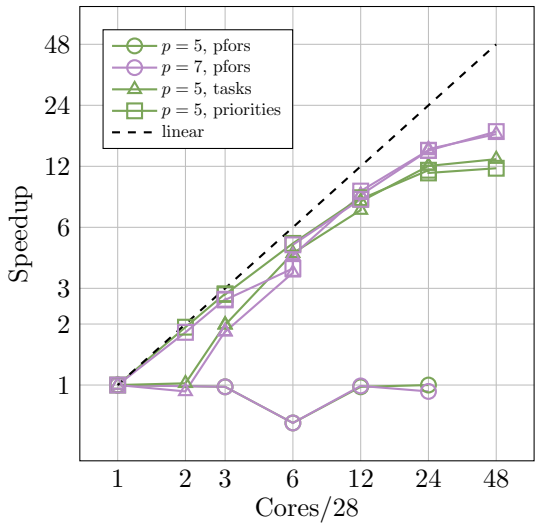
In this section, I compare the performance of tasking without priorities and enclave tasking, i.e. tasking with priorities and the possibility to steal tasks, against the sole recursion unrolling



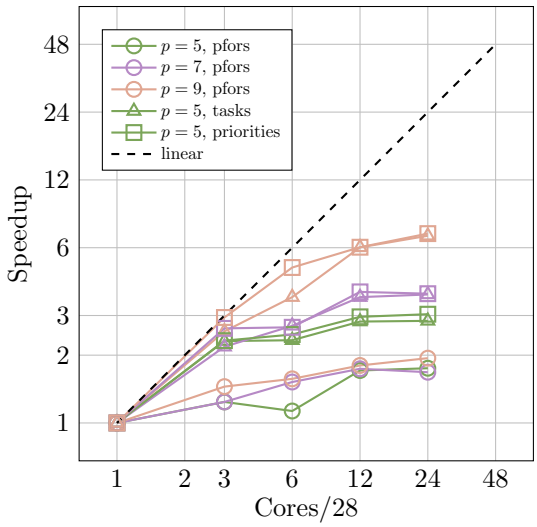
(a) Euler (Adaptive) – Single-Node



(b) Elastic (Adaptive) – Single-Node



(c) Euler (Regular) – 28 Processes



(d) Elastic (Adaptive) – 28 Processes

Fig. 9.5: Single-node and multi-node speedups for applications that simulate the compressible Euler and the linear elastic wave equations. Both applications used a regular base grid of size 27^3. The 27 worker processes that participated in the multi-node experiments with 28 processes obtained a base grid partition of size 9^3. Additional adaptive refinement was added on top of the regular base grid in three experiments. The hybrid parallelisation approaches are indicated via the marker symbol while the polynomial order is indicated via the line colour. "pfor", "tasks", and "priorities" refer to recursion unrolling, tasking without priorities, and enclave tasking, respectively.

technique. In my experiments, I used an ADER-DG implementation that is a merger of the enclave tasking scheme Algorithm 9.2 and the memory-efficient fused ADER-DG method that I present in Chapter 8. I ran a single-node experiment using 1 MPI process and a multi-node experiment using 28 MPI processes. In both experiments, I increased the number of TBB threads that are available to each rank. I considered two applications: A spherical explosion modelled with the compressible Euler equations [54] and the LOH.1 benchmark for the linear elastic wave equations [2]. A static mesh was used for both applications and used up to 1 level of AMR. The compressible Euler equations are a nonlinear PDE system where shocks are present. Hence, I solved the application with local space-time ADER-DG method plus a posteriori subcell limiting. I ran the LOH.1 benchmark with the Cauchy-Kowalewsky formulation of ADER-DG.

Hardware and Build Environment

I performed all experiments in this chapter on one or more nodes of SuperMUC-NG [79]. Each node is equipped with two 24 core Intel Xeon Platinum 8174 processors and 96 GB memory. The 48 processor cores are clocked with 3.10 GHz (base frequency) [9]. Each core has 64 KB of L1 and 1024 KB of L2 cache. The cores of a socket share 33 MB of non-inclusive L3 cache, i.e. data in a core's L2 cache must not necessarily be found in the socket's L3 cache [18]. I used the following software to build and run the experiments:

Software	Version or Revision
ExaHyPE (git branch and revision)	master – 9e1d39a2
ExaSeis (git branch and revision)	master – 3c50ca73
Peano (git branch and revision)	master – e027f18e
Intel Compiler	19.0.3.199 20190206
GNU Compiler	7.3.0
Intel TBB	TBB 2019 Update 4 (interface version: 11004)
Intel MPI	Update 4 Build 20190429 (id: cbdd16069)

For hybrid MPI + TBB experiments, the applications have been compiled according to the following ExaHyPE-specific environment variables:

- `EXAHYPE_CC=mpiicpc`
- `COMPILER=Intel`
- `MODE=Release`

- `DISTRIBUTEDMEM=MPI`
- `SHAREDMEM=TBB`
- `USE_IPO=on`

For TBB-only experiments, I used the same options but set `DISTRIBUTEDMEM=None`, while I set `SHAREDMEM=None` for MPI-only experiments. For serial runs, I set both environment variables to `None`. All experiments are performed with optimised ADER-DG kernels and optimised projectors between ADER-DG solution and finite volumes patch as generated by the ExaHyPE toolkit. All vectorisation is tailored to the `AVX2` instructions of the Intel Xeon processors. The auto-generated Makefiles of the applications, added the following performance-related compiler and linker flags:

- `COMPILER_CFLAGS+=-xCORE-AVX512 -fma -O3 -ip`
- `COMPILER_LFLAGS+=-xCORE-AVX512 -fma`

Results

I collect the results of the experiments in Fig. 9.5. Both tasking approaches perform significantly better than the baseline, the sole recursion unrolling technique ("pfors"). This is true for the single-node and especially for the multi-node experiments. In both compressible Euler experiments and the single-node linear elastic wave equation experiment, enclave tasking ("priorities") appears superior in the low core count regime to tasking without priorities ("tasks"). This trend is less pronounced in the multi-node experiment that I ran for the linear elastic wave equations. The performance of the baseline in the multi-node experiments agrees with the theoretical estimates in Table 9.1.

9.4 Summary

In this chapter, I presented two hybrid distributed-memory shared-memory parallelisation approaches for ExaHyPE's ADER-DG and finite volume method. I highlighted the limitations of the first approach that relies on the identification of regular subtrees in the adaptive spacetree. ExaHyPE's meshes are small as it uses the high-order ADER-DG method and patch-based finite volumes, which both have a large memory footprint per cell. To introduce more concurrency into the code, I proposed *enclave tasking*. In this approach, computationally intense cell-wise operations of ADER-DG and the finite volume method are not run directly

by the threads that traverse the spacetree. Instead, they are spawned as prioritised tasks that are put into a concurrent priority queue. In the background, consumer threads monitor the queue and grab a fair share of the available tasks. I provided experimental evidence that the application of this producer-consumer pattern can indeed increase the concurrency in ExaHyPE's implementations of ADER-DG and of the finite volumes methods. In comparison to an implementation that directly assigns tasks to cores without an intermediate priority queue, enclave tasking showed superior performance in the low core count regime. In this regime, the ability of the traversal threads to steal tasks with a certain priority whenever they have to wait has the biggest impact. The other tasking variant does not allow such stealing. Further experiments have to show if the priorisation or the stealing is the cause of the superior performance of enclave tasking in this regime.

Outlook: A Second Load Balancing Layer

In the ExaHyPE project, we currently work on a second layer of load balancing on top of Peano's spacetree partitioning. This additional layer uses task-offloading, i.e. ADER-DG predictor and corrector tasks are not run directly on overloaded processes but they and associated data are sent to processes that have less computational work (*victims*). In order to be successful this approach requires that the tasks are executed with high priority on the victim processes and sent back immediately before a Riemann solve on the original process requires the outcome of the tasks. ExaHyPE's enclave tasking infrastructure already provides a priority system. Therefore, it is well-suited to support this load balancing scheme.

Outlook: Multi-Solvers

An additional embarrassingly parallel dimension of parallelism can be introduced to simulations if the mesh does not only hold the cell-wise solution of one but of multiple solvers. Every time the spacetree traversal touches a cell or face, it runs solver operations in a parallel for loop or spawns these operations as tasks. This reduces overhead of the mesh traversal and speeds up the task production in the enclave tasking approach. I wrote ExaHyPE such that it supports multi-solver simulations.

10

Multi-Node Performance Studies

In this chapter, I aim to study the performance of ExaHyPE. However, studying the performance of an engine like ExaHyPE is not a well-defined problem. Its performance depends on the characteristics of the particular application that is considered. ExaHyPE is able to solve a very diverse range of applications that differ in their PDE terms — flux, non-conservative product, etc. ExaHyPE supports two different generic ADER-DG implementations for linear and nonlinear PDE systems plus two FVM solvers, approximation order and FVM patch size can be chosen freely. Moreover, mesh adaptivity and dynamic limiting can change the compute characteristics of ExaHyPE applications significantly. This list is not complete. Covering this vast parameter space demands substantial compute resources. Therefore, I focus on examining the performance of three applications that have been realised on top of ExaHyPE. Two applications solve the linear elastic wave equations. Both are of practical importance and will be developed further after the ExaHyPE project ends [4]. They both rely on the CK procedure to construct the space-time predictor but differ in their strategy to model topography and material distribution. One application relies on ExaHyPE's hybrid ADER-DG-FVM solver (see Chapter 7), the other on ExaHyPE's implementation of the plain ADER-DG method. The third application is from the fluid dynamics area. It solves the compressible Euler equations. I selected this application as it

uses the local space-time ADER-DG predictor instead of the CK procedure.

Structure

This chapter is structured as follows: Section 10.1 introduces the three applications, Section 10.2 presents the results of the performance studies, and Section 10.3 discusses them.

Hardware and Build Environment

I performed all experiments in this chapter on one or more nodes of SuperMUC-NG [79]. The same build setup as described in Chapter 9 was used. I briefly reiterate the hardware details: Each node of SuperMUC-NG is equipped with two 24 core Intel Xeon Platinum 8174 processors and 96 GB memory. The 48 processor cores are clocked with 3.10 GHz (base frequency) [9]. Each core has 64 KB of L1 and 1024 KB of L2 cache. The cores of a socket share 33 MB of non-inclusive L3 cache.

General Test Parameters

Due to the large number of features and parameters to test, I focus my performance studies on the performance of ExaHyPE's time stepping phase. Furthermore, I focus on the fused variable-time-step-size ADER-DG method presented in Chapter 8 plus the hybrid parallelisation *enclave tasking* from Chapter 9. The performance of other building blocks such as the mesh refinement and limiter recomputation phases will be subject of future studies.

10.1 Applications

This section details the ExaHyPE applications that I examined.

10.1.1 Seismology

I solve the linear elasticity problem with two methods: a diffuse interface hybrid ADER-DG-FVM method and a geometry-aligned ADER-DG method, which models complex topography via cell-wise curvilinear transformations. Mesh generation for complex topographies is one of the main difficulties in seismology simulations [92]. It is typically done in a semi-automatic fashion, which can take weeks to months for realistic scenarios. The methods discussed in this section aim to automate this aspect. The implementations have been obtained from the ExaSeis repository created and maintained by Leonhard Rannabauer. The development of the geometry-aligned ADER-DG method was driven by Leonhard Rannabauer and Kenneth Duru, while the diffuse interface method was developed by Maurizio Tavelli and then ported

to C++ by Leonhard Rannabauer.

Problem Description

I ran the *layer over halfspace* problem (LOH.1) [2]. The governing equations take the form

$$\begin{aligned} \frac{\partial \sigma}{\partial t} - E(\lambda, \mu) \cdot \nabla v &= S_\sigma, \\ \frac{\partial v}{\partial t} - \frac{1}{\rho} \nabla \cdot \sigma &= S_v. \end{aligned} \tag{10.1}$$

In this test, the material parameters are distributed specified in terms of compressional and shear velocity (P- and S-waves) ;see Table 10.1. Stress σ and velocity v are initially set to zero. Wave propagation is initiated by a point source δ_0 placed at $(0, 0, 2\text{km})^T$. The ExaSeis implementation of the benchmark uses outflow boundary conditions, i.e. state variables and flux from the outside are chosen as copies of the inside data. It performs static AMR such that a third of the computational domain around the point source location is refined to the maximum user-specified refinement level. In my performance studies, I used zero or one levels of AMR, and I switched off all receivers as I was only interested in the performance of the solvers.

Table 10.1: Material Distribution.

	$v_p = \sqrt{\frac{\lambda+2\mu}{\rho}}$ / (m/s)	$v_s = \sqrt{\frac{\mu}{\rho}}$ / (m/s)	ρ / (kg/m^3)
Layer (depth $<$ 1 km)	4,000	2,000	2,600
Halfspace (depth $>$ 1 km)	6,000	3,464	2,700

Geometry-Aligned ADER-DG Method

The geometry-aligned ADER-DG method applies curvilinear transformations to map each cell of an adaptive Cartesian mesh over the unit cube to a subset of the modelled geometry. In ExaSeis's implementation of the LOH.1 benchmark, the transformations are used to align the mesh to the material distribution of the LOH.1 problem. Let $F_K \colon \hat{K} \to K$ denote such a transformation from a reference cell $\hat{K}$ to physical cell K that is continuously differentiable and invertible. Given $\hat{x} \in \hat{K}$ and $x = F_K(\hat{x}) \in K$, define spatial ADER-DG basis functions ϕ and reference basis functions $\hat{\phi}$ such that $\phi(x) = \hat{\phi}(\hat{x})$. Then, the gradient on the physical cell K transforms as follows:

$$\frac{\partial \phi(x)}{\partial x} = \frac{\partial \hat{\phi}(\hat{x})}{\partial x} = \frac{\partial \hat{\phi}}{\partial \hat{x}} \frac{\partial \hat{x}}{\partial x} = \frac{\partial \hat{\phi}}{\partial \hat{x}} \frac{\partial F_K^{-1}(x)}{\partial x}.$$

The inverse function theorem links the derivatives of the inverse mapping to the inverse of the mapping's Jacobian matrix $\mathrm{D}F_K$, which is straightforward to compute:

$$\frac{\partial F_K^{-1}(x)}{\partial x} = \mathrm{D}F_K^{-\mathrm{T}}.$$

The mapping and its first derivatives are known and can be pre-computed. ExaSeis's geometry-aligned ADER-DG method stores them as additional material parameter per degree of freedom, i.e. per support point of the spatial Lagrange basis functions. The ExaSeis implementation stores 13 additional material parameters per degree of freedom, i.e. it requires twice as much storage as an implementation on cuboid cells. They are set during the initial mesh creation phase.

Diffuse Interface Method

ExaSeis's diffuse interface method utilises a plain adaptive Cartesian mesh. Instead of shaping the mesh according to the geometry it embeds the geometry within the mesh. This is done with a colouring function $\alpha \in [0,1]$ that assumes the value 1 inside of the solid and 0 inside of the air. Along the solid-air interface, there is a smooth transition between the two states. ExaSeis uses ExaHyPE's hybrid ADER-DG-FVM method to solve the linear system. Far away from the solid-air interface, it solves (10.1) with the ADER-DG method without applying any transformations, i.e. no derivatives need to be stored. In the vicinity of the interface, it places FVM patches. On these FVM patches, a different PDE is solved [92],

$$\begin{aligned} \frac{\partial \sigma}{\partial t} - E(\lambda,\mu) \cdot \frac{1}{\alpha} \nabla(\alpha\, v) + \frac{1}{\alpha} E(\lambda,\mu) \cdot v \otimes \nabla\alpha &= S_\sigma, \\ \frac{\partial \alpha v}{\partial t} - \frac{\alpha}{\rho} \nabla \cdot \sigma - \frac{1}{\rho} \sigma \nabla\alpha &= S_v, \\ \frac{\partial \alpha}{\partial t} = 0, \qquad \frac{\partial \lambda}{\partial t} = 0, \qquad \frac{\partial \mu}{\partial t} = 0, \qquad \frac{\partial \rho}{\partial t} &= 0, \end{aligned} \tag{10.2}$$

that takes the smooth interface parameter α into account.

Per nodal ADER-DG degree of freedom and per volume on each FVM patch, the diffuse interface method stores only 13 doubles, i.e. half of the 26 doubles the geometry-aligned method stores per ADER-DG degree of freedom. If the number of FVM patches is small, this allows the method to run larger problems with the same amount of compute resources.

10.1.2 Compressible Euler

As fluid dynamics test case, I consider a spherical gas explosion, which essentially simulates a multi-dimensional Riemann problem; see [54]. The flow is modelled by the compressible Euler

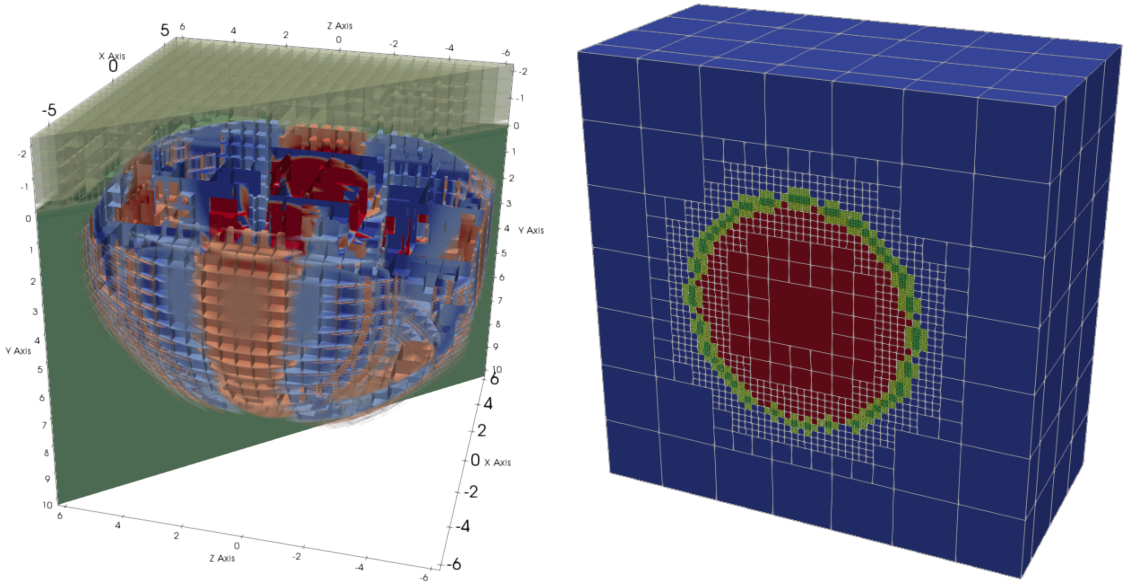


Fig. 10.1: Typical simulation snapshots of the two benchmark problems. Mesh resolution and approximation order were chosen significantly lower than in the performance studies. Left: Snapshot of the wave field when running the LOH.1 benchmark with the diffuse interface method. *Reprinted from* [83]. Right: Snapshot of the initial condition of the spherical explosion benchmark simulated with the Euler solver. The finite volumes patches are highlighted in green (troubled cells: dark shade, face-connected neighbours: lighter shade).

equations,

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\, v\\ \rho\, E\end{pmatrix}+\begin{pmatrix}\rho v\\ \rho\, v\otimes v+P\, I_{d\times d}\\ v(\rho E+P)\end{pmatrix}=0, \tag{10.3}$$

where the conserved state variables are constructed from mass density ρ, velocity v and energy density E. The square matrix $I_{d\times d}$ with size d is the identity matrix. The pressure P is computed according to the EOS of an ideal gas with adiabatic index γ:

$$P=(\gamma-1)\,(\rho E-\frac{1}{2}\,\rho v\cdot v).$$

I ran this test with ExaHyPE's hybrid ADER-DG-FVM solver. Figure 10.2 shows the density initial condition plus the initial configuration of the FVM subdomain for a typical setup.

10.2 Results

I measured the scalability of the applications in single-node and multi-node studies. The latter always employed 28 or 731 ranks. Both rank numbers achieve a fair distribution of the spacetree leaf cells for regular spacetrees, i.e. in setups where the AMR levels are 0; see Chapter 6 for details. The largest runs used 731 nodes. All experiments where run for 10 time steps. The reported measurements are based on the average time these time steps took.

Seismology

Fig. 10.2 compares the performance of diffuse interface hybrid ADER-DG-FVM solver and geometry-aligned ADER-DG method for orders 3,5, and 7. The performance study considered meshes of size 25^3 and 79^3 plus 0 or 1 levels of AMR. Note that the usage of 731 ranks with the smallest mesh with 25^3 cells results in very coarse partitions with approximately 22 cells per rank (see Table 10.2 in the discussion). The figure shows the degree of freedom (DOF) updates the schemes performed per second. The geometry-aligned method reaches an at least 10 times higher DOF throughput than the diffuse-interface method in this study. However, the diffuse interface method shows a better scalability. The scalability of both methods improves with increasing order; see also Fig. 10.3.

I present the measurements for the compressible Euler application for orders $p = 5$ and $p = 7$ in Fig. 10.4. Compared to the seismology diffuse interface method, the method achieves a higher maximum DOF throughput. Both hybrid solver applications show good scalability.

Fluid Dynamics

I present the measurements for the compressible Euler application for orders $p = 5$ and $p = 7$ in Fig. 10.4. Compared to the seismology diffuse interface method, the method achieves a higher maximum DOF throughput. Both hybrid solver applications show good scalability.

10.3 Discussion

In this chapter, I investigate the scalability of three applications that have been realised upon ExaHyPE. Two are from the seismology area while the other is from fluid dynamics. One seismology application relies on the sole ADER-DG method while the other utilises ExaHyPE's hybrid ADER-DG-FVM solver. The fluid dynamics application is built upon ExaHyPE's hybrid solver, too. Moreover, it uses the local space-time DG predictor as the

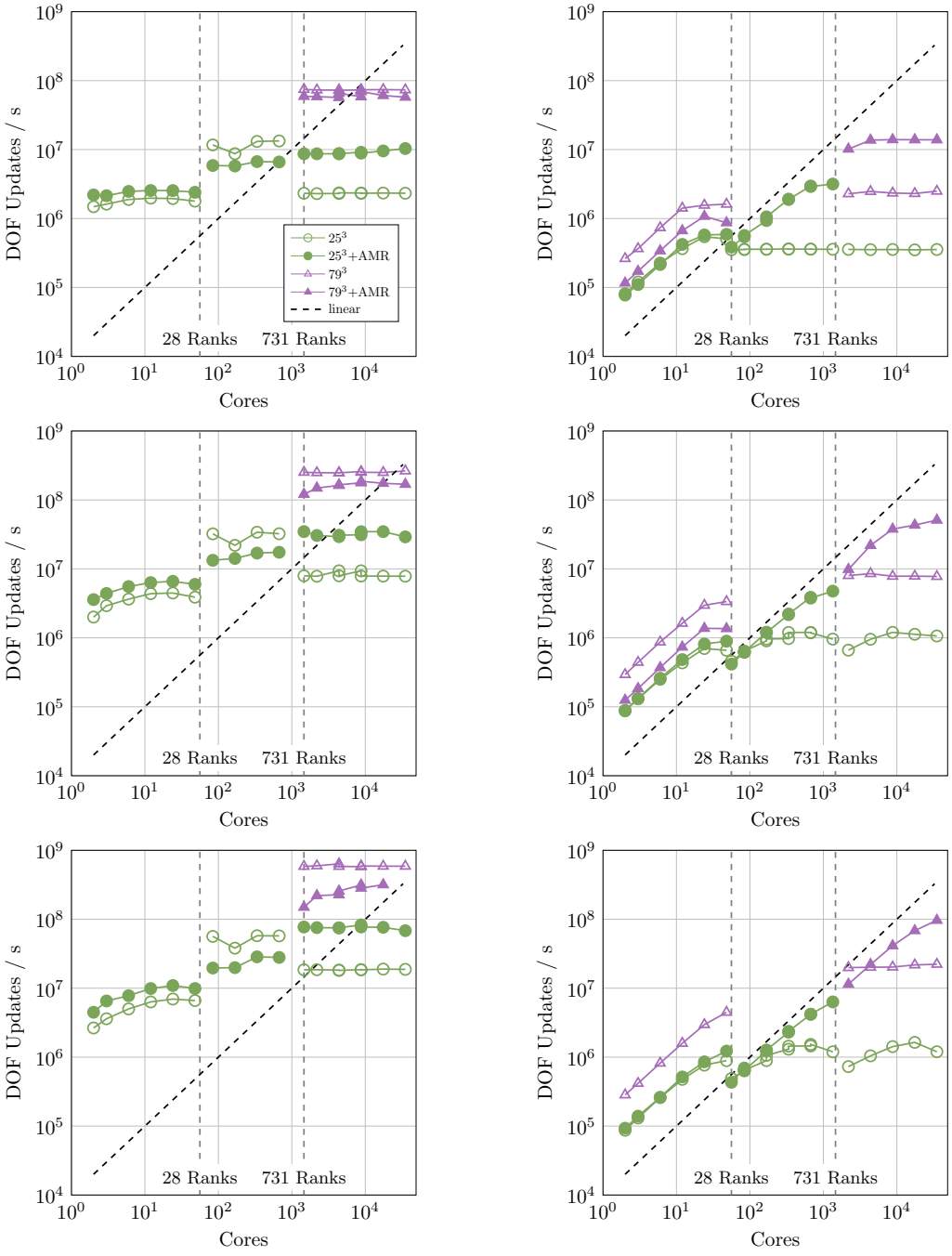


Fig. 10.2: Scaling of the seismology codes for the polynomial orders $p = 3$, $p = 5$ and $p = 7$ from top to bottom. The hybrid parallelisation used 1, 28, or 731 multi-threaded MPI processes. Left: Geometry-aligned ADER-DG method. Right: Diffuse interface method.

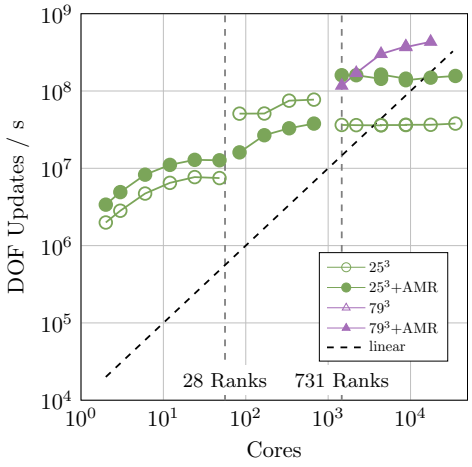


Fig. 10.3: Scaling of the geometry-aligned ADER-DG method for $p = 9$. The hybrid parallelisation used 1, 28, or 731 multi-threaded MPI processes.

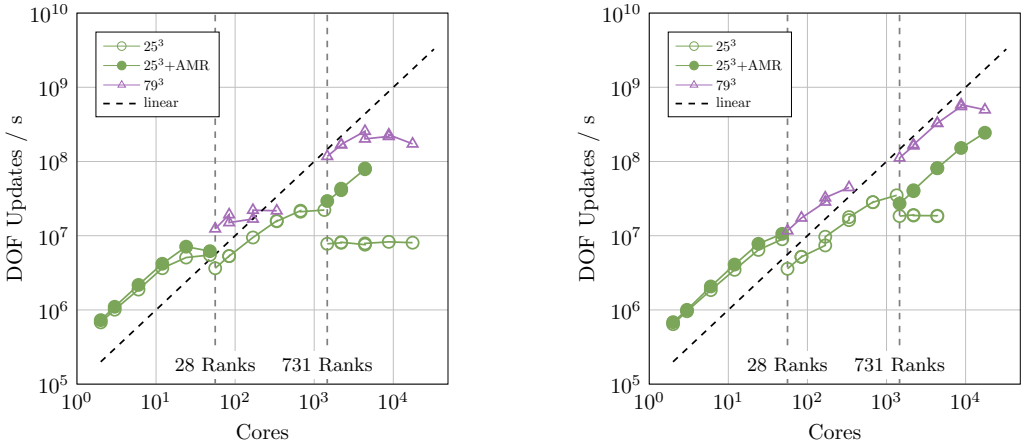


Fig. 10.4: Strong Scaling of the compressible Euler application. The hybrid parallelisation used 1, 28, or 731 multi-threaded MPI processes. Left: Order $p = 5$. Right: Order $p = 7$.

flux is nonlinear. The ADER-DG method of both seismology applications construct the space-time predictor via the Cauchy-Kowalewsky (CK) procedure. While the local-space time DG predictor requires up to $p+1$ iterations to converge, the CK procedure always performs the same number of operations. The number of iterations of the former varies from cell to cell. The seismology applications can use the CK procedure because the solved PDE systems is linear in areas where the ADER-DG method is used. The seismology diffuse interface method employs the FVM where the diffuse interface formulation is nonlinear, i.e. along the interface between solid and air.

In my experiments, all three applications were realised as a fused variable-time-step-size ADER-DG method (see Chapter 8). The schemes were parellelised via the enclave tasking hybrid parallelisation from Chapter 9. In general, I observed weak scaling for all applications, i.e. an increase of the problem size and a simultaneous increase of the compute resources resulted in an increase of processed degrees of freedom per second. However, the weak scaling behaviour of the applications has not been ideal. Good shared-memory scalability over multiple cores has been observed for the diffuse interface and Euler application. For the seismology application that used the sole ADER-DG solver, I only observed limited shared-memory scalability. The scalability recovered for very high orders. This is in line with the results from my preliminary studies in the results section of Chapter 9.

ExaHyPE's fused time stepping algorithm for adaptive meshes is mapped to two Peano mesh traversals per time step. I suspected that the strong-scaling bottleneck discovered with the geometry-aligned seismology application is linked to the traversal time. To follow up on this suspicion, I measured the performance of Peano's grid traversal and the cost of the substeps of the used considered solvers. To determine the mesh traversal cost, I constructed the grid according to the refinement criterion that the geometry-aligned ADER-DG LOH.1 implementation employs and then ran an empty mesh traversal, i.e. a traversal where the spacetree cells and faces of the mesh are traversed but none of ExaHyPE's data structures. I ran the experiments with 28 and 731 processes where the spacetree leaf levels were distributed among 27 and 729 ranks, respectively. This achieves a fair distribution of the spacetree leaf cells for regular spacetrees, i.e. in the setups where the AMR levels are 0. The experimental data reveals that the processing of a cell can take a couple of microseconds up to a couple of milliseconds; see Table 10.2. For the majority of experiments, the measured time is in the

microsecond range.

Table 10.2: Empty traversal cost: Time spent within a single mesh traversal that performs no operation despite the reduction at the end of the traversal. The mesh was created according to the refinement criterion of the geometry-aligned ADER-DG method. The number of cells includes only leaf cells. The listed timings are the average out of 50 iterations.

Base Mesh	AMR Levels	Cells	Ranks	Cells/Ranks	Time / s	Time/(Cells/Ranks) / µs
241^3	0	14M	28	500k	7.15 s	14.30
241^3	0	14M	731	19k	0.46 s	24.02
79^3	0	493k	28	17k	0.29 s	16.47
79^3	0	493k	731	675	0.22 s	326.1
79^3	1	1.2M	28	44k	2.16 s	49.03
79^3	1	1.2M	731	1.69k	0.28 s	165.9
25^3	0	16k	28	558	0.04 s	71.68
25^3	0	16k	731	22	0.21 s	9,825
25^3	1	58k	28	2.09k	0.19 s	90.90
25^3	1	58k	731	81	0.21 s	2,623
25^3	2	621k	28	22.2k	1.96 s	88.33
25^3	2	621k	731	850	0.24 s	282.3

The measured runtimes for the isolated solver substeps of the diffuse interface method are collected in Fig. 10.5. Each timing was computed as the average out of 500 runs. Noteworthy is that the runtime of the predictor substep is for low polynomial orders in the range of the time that it takes empty mesh traversals to process a cell. For enclave tasking, this implies that the production of predictor tasks takes approximately as long as their consumption. Consequently, concurrency is limited for low polynomial orders. Furthermore, we observe that the FVM update substep can be orders of magnitude more expensive than the CK predictor (Elasticity) or than a Picard iteration of the local space-time DG predictor (Euler). Even if it takes the predictor $p+1$ Picard iterations to converge, the FVM update is still at least 3 times more expensive (Euler, $p = 9$); see Fig. 7.6. We can expect that applications that use the sole ADER-DG solver have lower runtimes; however, applications that use the hybrid ADER-DG-FVM solver show better scalability with the enclave tasking procedure from Chapter 9 as the cell-wise update task is significantly more expensive than any ADER-DG task.

As the traversal speeds up the more processes are employed, it appears to be a good strategy

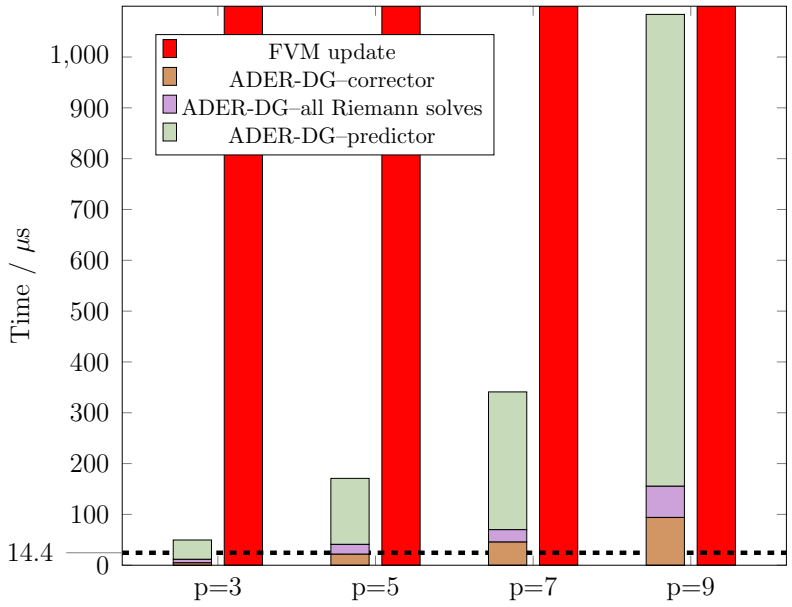


Fig. 10.5: Diffuse interface method: Minimum measured Peano cell processing time (14.4 μs) from Table 10.2 versus typical cost of the FVM patch update and of the (optimised) ADER-DG operations that are performed per cell. The plot cuts off the bars corresponding to the FVM measurements; the measurements are 4,407 μs, 14,006 μs, 31,887 μs, and 63,344 μs for orders $p = 3, 5, 7, 9$, respectively. Order p corresponds to an FVM patch with $2\,p + 1$ cells per coordinate direction. *Data was collected on SuperMUC-NG. Shown is the average of 500 measurements per FVM and ADER-DG operation.*

to have a large number of MPI processes plus a limited number of TBB tasks to work in the performance sweet spot where enough tasks are generated to keep the background task consumer threads busy.

Future versions of ExaHyPE might not run the mesh traversal at all and instead memorise all solver operations as tasks. If the mesh is adapted, tasks are added to the task set or removed. The tasks can then be distributed equally among the consumer threads at the begin of every time step.

11 Conclusion

This chapter summarises the main achievements of this thesis and gives an outlook on future research directions.

11.1 Summary

This thesis describes the development of the core algorithms of the software engine ExaHyPE. ExaHyPE enables the rapid development of parallel simulation codes for hyperbolic partial differential equations in first-order form. The main vision of ExaHyPE is: Users focus solely on the physics, while the engine takes care of the parallelisation and hides the algorithmic complexity of its advanced numerics. This approach cleanly separates the concerns of application scientists, developers of data structures and algorithms, and performance engineers. The engine has been applied to a wide range of applications including elastic wave propagation, shallow water flow and general relativity [83]. ExaHyPE is built upon numerical methods and algorithms that have been selected based on their accuracy and their excellent performance on today's petascale machines. It combines the high-order ADER-DG method with parallel dynamic adaptive mesh refinement to improve the accuracy locally and applies a posteriori limiting to treat discontinuities in the solution. Discontinuities introduce non-physical oscillations and solution values into the ADER-DG solution. A posteriori limiting recomputes the

solution with a robust FVM method in areas where this is the case.

Within the next five years, supercomputing will advance to the exascale era, where massively parallel machines are able to perform more than 10^{18} floating point calculations per second. To obtain optimal performance on these machines, it is mandatory that software is written in a communication-avoiding way. Memory access and inter-process communication must be as minimal and overlapped with computations. ADER-DG is a predictor-corrector method. Straightforward realisations of the method allocate a space-time predictor vector in addition to the solution vector. In case of nonlinear PDE systems, they further allocate a space-time volume flux. These auxiliary variables have a memory footprint that is by at least a factor of $p + 1$ larger than that of the solution, where p is the approximation order of ADER-DG. I identified the large memory footprint of ADER-DG due to the allocation of these space-time fields as a roadblock to optimal performance of the method on exascale machines. To tackle this issue, I present three communication-avoiding low-storage ADER-DG variants in Chapter 8 of this thesis. For the a posteriori limiting ADER-DG method, I accomplished an additional significant reduction in communication steps by reformulating the scheme as a hybrid ADER-DG-FVM method. This is covered in Chapter 7 of this thesis.

Chapter 9 presents *enclave tasking*, a hybrid parallelisation for ExaHyPE's numerical methods. ExaHyPE is built upon the meshing framework/PDE engine Peano, which was originally developed for realising multigrid solvers for low-order finite difference and finite element discretisations. Peano's shared-memory parallelisation is well-suited to parallelise the execution of such solvers. ExaHyPE's ADER-DG and FVM methods store a large number of degrees of freedom per cell. Hence, the meshes are in general smaller than those of low-order discretisations. I demonstrated that Peano's on-board shared-memory parallelisation does only introduce limited shared-memory concurrency into the code in this case. The proposed enclave tasking technique introduces additional shared-memory concurrency by spawning computationally intense ADER-DG and FVM substeps as tasks. In the background, consumer threads process the spawned tasks. To enforce that tasks along MPI subdomain are processed first, the presented parallelisation required to tweak the underlying TBB runtime to take priorities into account.

Dynamic adaptivity is important for ExaHyPE's numerical algorithms to resolve complicated geometries and solution features such as shock waves. I present ExaHyPE's mesh data

structure and mesh adaptation algorithms in Chapter 6 of this thesis. This mesh data structure allows to realise ADER-DG on arbitrarily adaptive meshes and further allows to identify vertical communication synchronisation points in the underlying spacetree data structure that is powered by Peano. These are then eliminated consequently. To merge the concepts of a posteriori limiting and adaptive mesh refinement, I introduced two novel mesh refinement techniques called *halo refinement* and *a posteriori refinement* that pre-refine the mesh around interesting existing and emerging solution features.

11.2 Outlook

My personal conclusion is that ExaHyPE has not shown its full potential yet. In particular, the parallelisation requires further work. However, I also think that we have not fully exploited ExaHyPE's already existing potential yet. Especially multi-solver and hybrid solver functionality offer potential for new interesting applications such as multi-level Monte Carlo methods or fluid-particle hybrid solvers. This section lists further ideas to improve ExaHyPE.

Features

The integration of a number of important features is still ongoing work. Infrastructure for supporting more efficient time stepping variants, such as local time stepping and anarchic local time stepping, was provided. However, the last implementation steps have not been performed yet. Furthermore, ExaHyPE can already run multiple simulations simultaneously on the same mesh; however, the coupling of different solvers was not addressed so far.

Performance

There is ongoing work on task offloading mechanisms where overbooked ranks offload some of their computationally intense work, e.g. space-time predictor tasks, to ranks with less work. This is a second order load balancing strategy that works on top of Peano's existing spacetree partitioning procedure. The latter and AMR in general might lead to severe load imbalances. Moreover, work per rank might not be predictable due to limiting and the variable number of predictor Picard iterations that ADER-DG requires when solving a nonlinear PDE system. The number of iterations may vary from cell to cell.

The performance studies in Chapter 9 and Chapter 10 reveal the limitations of the presented

enclave tasking hybrid parallelisation. Here, the computationally efficient mesh-aligned ADER-DG method showed limited shared-memory scalability compared to two other applications that coupled ADER-DG to a more expensive FVM solver. I identified Peano's mesh traversal as scalability bottleneck of the mesh-aligned ADER-DG solver. It has to be avoided and tasks have to be distributed directly at the beginning of a time step to improve scalability for this solver. I propose that Peano should extend its specification file language to support such algorithms.

Hybrid Solver Performance

ExaHyPE's hybrid ADER-DG-FVM solver shows good shared-memory and distributed-memory scalability. However, this can be to the most extent attributed to the costly FVM updates. Updating an FVM patch with $2\,p+1$ subcells per dimensions can be up to 3 to 4 times slower than running $p+1$ iterations of an ADER-DG method's predictor computation. This cost imbalance should be tackled from two directions: First, ExaHyPE's finite volumes methods should be optimised with as much care as the ADER-DG method. They have not been subject to optimisations yet as it was not anticipated in the original project proposal that the FVM patches are that costly. Second, ExaHyPE should offer the possibility to choose smaller (or larger) FVM subgrid sizes if the a posteriori limiting ADER-DG method is used. The ExaHyPE toolkit can generate appropriate ADER-DG-to-FVM and FVM-to-ADER-DG projectors. Other researchers likely made similar observations. The authors of [77] use a FVM subgrid size of $p+1$ in their hybrid DG-FVM method.

Future of ExaHyPE

ExaHyPE has been selected as one of ten European Union flagship codes by the consortium of the *Center of Excellence (CoE) for Exascale in Solid Earth (SE)* (ChEESE) [4]. They will investigate how ExaHyPE's seismology applications can be developed into seismology services.

Bibliography

[1] List of finite element software packages. URL: https://en.wikipedia.org/wiki/List_of_finite_element_software_packages.

[2] Problem wp2_loh1. [Online; accessed 07/05/2019]. URL: http://www.sismowine.org/model/WP2_LOH1.pdf.

[3] ASPECT: Advanced Solver for Problems in Earth's Convection. [Online; accessed 19/05/2019]. URL: https://aspect.geodynamics.org/.

[4] ChEESE Flagship Codes. URL: https://cheese-coe.eu/results/flagship-codes.

[5] DUNE. URL: https://www.dune-project.org.

[6] ExaHyPE Guidebook (User manual). [Online; accessed 18/04/2019]. URL: http://www.peano-framework.org/exahype/guidebook.pdf.

[7] ExaHyPE. [Online; accessed 19/05/2019]. URL: http://www.peano-framework.org/index.php/exahype/.

[8] ExaHyPE — Second project report. URL: https://www.ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5be4411bf&appId=PPGMS.

[9] Intel®Xeon®Platinum 8174 Processor. [Online; accessed 06/05/2019]. URL: https://ark.intel.com/content/www/us/en/ark/products/136874/intel-xeon-platinum-8174-processor-33m-cache-3-10-ghz.html.

[10] JSON Schema. [Online; accessed 09/05/2019]. URL: http://jinja.pocoo.org.

[11] JSON. [Online; accessed 09/05/2019]. URL: http://json.org.

[12] Jinja. [Online; accessed 09/05/2019]. URL: http://jinja.pocoo.org.

[13] NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/, Release 1.0.22 of 2019-03-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller and B. V. Saunders, eds. URL: http://dlmf.nist.gov/.

[14] OpenFOAM. URL: https://openfoam.com/.

[15] Peano Cookbook (Quick Start Guide). [Online; accessed 06/04/2019]. URL: http://www.peano-framework.org/peano/p3/cookbook.pdf.

[16] PeanoClaw. URL: https://github.com/www5sccs/peanoclaw.

[17] PyClaw. URL: http://www.clawpack.org/pyclaw/index.html.

[18] Skylake (server) - Microarchitectures - Intel. [Online; accessed 06/05/2019]. URL: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server).

[19] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*. Dover Publications, 1965.

[20] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, September 1988. URL: http://doi.acm.org/10.1145/48529.48535, doi:10.1145/48529.48535[4].

[21] S. Ashby, P. Beckman, J. Chen, P. Colella, D. Crawford, J. J. Dongarra, R. Lusk, P. Messina, T. Mezzacappa, and M. Wright. Report on Exascale Computing. Technical Report, U.S. Department of Energy, Office of Science, 2010.

[22] M. Bader. *Space-Filling Curves*. Volume 9 of Texts in Computational Science and Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-31045-4 978-3-642-31046-1. URL: http://link.springer.com/10.1007/978-3-642-31046-1.

[23] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework. *Computing*, 82(2-3):103–119, July 2008. URL: http://link.springer.com/10.1007/s00607-008-0003-x, doi:10.1007/s00607-008-0003-x[5].

[4] https://doi.org/10.1145/48529.48535

[5] https://doi.org/10.1007/s00607-008-0003-x

[24] J. Behrens. *Example Applications*, pages 123–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. URL: https://doi.org/10.1007/3-540-33383-5_8, doi:10.1007/3-540-33383-5_8[6].

[25] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, May 1989. doi:10.1016/0021-9991(89)90035-1[7].

[26] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984. URL: http://www.sciencedirect.com/science/article/pii/0021999184900731, doi:http://dx.doi.org/10.1016/0021-9991(84)90073-1[8].

[27] M. Blatt and P. Bastian. The Iterative Solver Template Library. In B. Kågström, E. Elmroth, J. J. Dongarra, and J. Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699, pages 666–675. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. URL: http://link.springer.com/10.1007/978-3-540-75755-9_82, doi:10.1007/978-3-540-75755-9_82[9].

[28] M. Blatt and P. Bastian. On the generic parallelisation of iterative solvers for the finite element method. *International Journal of Computational Science and Engineering*, 4(1):56, 2008. URL: http://www.inderscience.com/link.php?id=21112, doi:10.1504/IJCSE.2008.021112[10].

[29] D. Bouche, J.-M. Ghidaglia, and F. Pascal. Error Estimate and the Geometric Corrector for the Upwind Finite Volume Method Applied to the Linear Advection Equation. *SIAM Journal on Numerical Analysis*, 43(2):578–603, January 2005. URL: http://epubs.siam.org/doi/10.1137/040605941, doi:10.1137/040605941[11].

[30] C. Burstedde, D. Calhoun, K. Mandli, and A. R. Terrel. ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. *arXiv preprint arXiv:1308.1472*, 2013. URL: http://arxiv.org/abs/1308.1472.

[6] https://doi.org/10.1007/3-540-33383-5_8
[7] https://doi.org/10.1016/0021-9991(89)90035-1
[8] https://doi.org/http://dx.doi.org/10.1016/0021-9991(84)90073-1
[9] https://doi.org/10.1007/978-3-540-75755-9_82
[10] https://doi.org/10.1504/IJCSE.2008.021112
[11] https://doi.org/10.1137/040605941

[31] C. Burstedde, L. C. Wilcox, and O. Ghattas. P4est : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, January 2011. URL: http://epubs.siam.org/doi/abs/10.1137/100791634, doi:10.1137/100791634[12].

[32] J. C. Butcher. *Numerical methods for ordinary differential equations*. Wiley, Chichester, England ; Hoboken, NJ, 2nd ed edition, 2008. ISBN 978-0-470-72335-7. OCLC: ocn191024153.

[33] J. C. Butcher. ON FIFTH AND SIXTH ORDER EXPLICIT RUNGE-KUTTA METHODS: ORDER CONDITIONS AND ORDER BARRIERS. *Canadian Applied Mathematics Quarterly*, pages 14, 2009.

[34] D. Calhoun and C. Burstedde. ForestClaw: A parallel algorithm for patch-based adaptive mesh refinement on a forest of quadtrees. *arXiv:1703.03116 [cs]*, March 2017. arXiv: 1703.03116. URL: http://arxiv.org/abs/1703.03116.

[35] J. Casper and H. L. Atkins. A Finite-Volume High-Order ENO Scheme for Two-Dimensional Hyperbolic Systems. *Journal of Computational Physics*, 106(1):62 – 76, 1993. URL: http://www.sciencedirect.com/science/article/pii/S0021999183710910, doi:http://dx.doi.org/10.1006/jcph.1993.1091[13].

[36] M. Castro, J. Gallardo, and C. Madroñal. High order finite volume schemes based on reconstruction of states for solving hyperbolic systems with nonconservative products. Applications to shallow-water systems. *Math. Comput.*, 75:1103–1134, 2006. doi:10.1090/S0025-5718-06-01851-5[14].

[37] D. E. Charrier, B. Hazelwood, E. Tutlyaeva, M. Bader, M. Dumbser, A. Kudryavtsev, A. Moskovsky, and T. Weinzierl. Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *The International Journal of High Performance Computing Applications*, pages 109434201984264, April 2019. URL: http://journals.sagepub.com/doi/10.1177/1094342019842645, doi:10.1177/1094342019842645[15].

[12] https://doi.org/10.1137/100791634
[13] https://doi.org/http://dx.doi.org/10.1006/jcph.1993.1091
[14] https://doi.org/10.1090/S0025-5718-06-01851-5
[15] https://doi.org/10.1177/1094342019842645

[38] D. E. Charrier, B. Hazelwood, and T. Weinzierl. Enclave Tasking for Discontinuous Galerkin Methods on Dynamically Adaptive Meshes. *arXiv:1806.07984 [cs]*, June 2018. arXiv: 1806.07984. URL: http://arxiv.org/abs/1806.07984.

[39] D. E. Charrier and T. Weinzierl. Stop talking to me – a communication-avoiding ADER-DG realisation. *arXiv:1801.08682 [cs]*, January 2018. arXiv: 1801.08682. URL: http://arxiv.org/abs/1801.08682.

[40] B. Cockburn, S. Hou, and C.-W. Shu. The Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. IV. The multidimensional case. *Mathematics of Computation*, 54(190):545–581, 1990.

[41] B. Cockburn, S.-Y. Lin, and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: one-dimensional systems. *Journal of Computational Physics*, 84(1):90–113, 1989.

[42] B. Cockburn and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. *Mathematics of computation*, 52(186):411–435, 1989.

[43] B. Cockburn and C.-W. Shu. The Runge-Kutta local projection P^1-discontinuous-Galerkin finite element method for scalar conservation laws. *RAIRO-Modélisation mathématique et analyse numérique*, 25(3):337–361, 1991.

[44] B. Cockburn and C.-W. Shu. Runge–Kutta Discontinuous Galerkin Methods for Convection-Dominated Problems. *Journal of Scientific Computing*, 16(3):173–261, September 2001. URL: https://doi.org/10.1023/A:1012873910884, doi:10.1023/A:1012873910884[16].

[45] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report, University of Tennessee, Knoxville TN, 37996, May 2019. URL: http://www.netlib.org/benchmark/performance.ps.

[46] J. J. Dongarra, M. A. Heroux, and P. Luszczek. A new metric for ranking high-performance computing systems. *National Science Review*, 3(1):30–35, March

[16] https://doi.org/10.1023/A:1012873910884

2016. URL: https://academic.oup.com/nsr/article-lookup/doi/10.1093/nsr/nwv084, doi:10.1093/nsr/nwv084[17].

[47] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. Van Straalen, and K. Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, December 2014. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731514001178, doi:10.1016/j.jpdc.2014.07.001[18].

[48] M. Dumbser, D. S. Balsara, E. F. Toro, and Claus-Dieter Munz. A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes. *Journal of Computational Physics*, 227(18):8209–8253, September 2008. URL: http://linkinghub.elsevier.com/retrieve/pii/S0021999108002829, doi:10.1016/j.jcp.2008.05.025[19].

[49] M. Dumbser, M. Castro, C. Parés, and E. F. Toro. ADER schemes on unstructured meshes for nonconservative hyperbolic systems: Applications to geophysical flows. *Computers & Fluids*, 38(9):1731–1748, October 2009. URL: https://linkinghub.elsevier.com/retrieve/pii/S0045793009000498, doi:10.1016/j.compfluid.2009.03.008[20].

[50] M. Dumbser, C. Enaux, and E. F. Toro. Finite volume schemes of very high order of accuracy for stiff hyperbolic balance laws. *Journal of Computational Physics*, 227(8):3971–4001, April 2008. URL: http://linkinghub.elsevier.com/retrieve/pii/S0021999107005578, doi:10.1016/j.jcp.2007.12.005[21].

[51] M. Dumbser, F. Fambri, M. Tavelli, M. Bader, and T. Weinzierl. Efficient implementation of ADER discontinuous Galerkin schemes for a scalable hyperbolic PDE engine. *arXiv:1808.03788 [physics]*, August 2018. arXiv: 1808.03788. URL: http://arxiv.org/abs/1808.03788.

[17] https://doi.org/10.1093/nsr/nwv084
[18] https://doi.org/10.1016/j.jpdc.2014.07.001
[19] https://doi.org/10.1016/j.jcp.2008.05.025
[20] https://doi.org/10.1016/j.compfluid.2009.03.008
[21] https://doi.org/10.1016/j.jcp.2007.12.005

[52] M. Dumbser and M. Käser. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case. *Geophysical Journal International*, 167(1):319–336, October 2006. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2006.03120.x, doi:10.1111/j.1365-246X.2006.03120.x[22].

[53] M. Dumbser, M. Käser, and E. F. Toro. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes - V. Local time stepping and p-adaptivity. *Geophysical Journal International*, 171(2):695–717, November 2007. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2007.03427.x, doi:10.1111/j.1365-246X.2007.03427.x[23].

[54] M. Dumbser, O. Zanotti, R. Loubère, and S. Diot. A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. *Journal of Computational Physics*, 278:47–75, 2014.

[55] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, 567–575. Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1882792.1882860.

[56] Martin S. Aln\a es, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 2015. doi:10.11588/ans.2015.100.20553[24].

[57] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Transactions on Mathematical Software*, 43(3):1–27, December 2016. URL: http://dl.acm.org/citation.cfm?doid=2988516.2998441, doi:10.1145/2998441[25].

[58] J.-M. Gallard, L. Krenz, L. A. Rannabauer, A. Reinarz, and M. Bader. Role Oriented Code Generation in an Engine for Solving Hyperbolic PDE Systems. In *SC'19*, 8. 2019.

[22] https://doi.org/10.1111/j.1365-246X.2006.03120.x
[23] https://doi.org/10.1111/j.1365-246X.2007.03427.x
[24] https://doi.org/10.11588/ans.2015.100.20553
[25] https://doi.org/10.1145/2998441

Submitted to “Software Engineering for HPC-Enabled Research” workshop at SC19.

[59] G. Gassner, M. Dumbser, F. Hindenlang, and C.-D. Munz. Explicit one-step time discretizations for discontinuous Galerkin and finite volume schemes based on local predictors. *Journal of Computational Physics*, 230(11):4232–4247, May 2011. URL: http://linkinghub.elsevier.com/retrieve/pii/S0021999110005802, doi:10.1016/j.jcp.2010.10.024[26].

[60] S. Gill. A process for the step-by-step integration of differential equations in an automatic digital computing machine. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 47, 96–108. Cambridge University Press, 1951.

[61] S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Mat. Sb. (N.S.)*, 47 (89):271–306, 1959.

[62] A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy. Uniformly High Order Accurate Essentially Non-oscillatory Schemes, III. *Journal of Computational Physics*, 131(1):3 – 47, 1997. URL: http://www.sciencedirect.com/science/article/pii/S0021999196956326, doi:http://dx.doi.org/10.1006/jcph.1996.5632[27].

[63] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A. Gabriel, C. Pelties, A. Bode, W. Barth, X. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers. In *SC ‘14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 3–14. November 2014. doi:10.1109/SC.2014.6[28].

[64] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Number 54 in Texts in applied mathematics. Springer, 2008.

[65] F. J. Hindenlang and G. J. Gassner. On the order reduction of entropy stable DGSEM for the compressible Euler equations. *arXiv:1901.05812 [math]*, January 2019. arXiv: 1901.05812. URL: http://arxiv.org/abs/1901.05812.

[66] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/alge-

[26] https://doi.org/10.1016/j.jcp.2010.10.024
[27] https://doi.org/http://dx.doi.org/10.1006/jcph.1996.5632
[28] https://doi.org/10.1109/SC.2014.6

braic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, September 2005. URL: http://portal.acm.org/citation.cfm?doid=1089014.1089020, doi:10.1145/1089014.1089020[29].

[67] H. Igel. *Computational seismology : a practical introduction*. Oxford University Press, Oxford, United Kingdom, 2017. ISBN 978-0-19-871740-9 0-19-871740-7 978-0-19-871741-6 0-19-871741-5.

[68] Intel. Libxsmm. 2019. [Online; accessed 06/05/2019]. URL: https://github.com/hfp/libxsmm.

[69] K. Jackson. How to liberate scientists from writing code. may 2018. [Online; accessed 18/05/2019]. URL: https://sciencenode.org/feature/How%20to%20liberate%20scientists.php.

[70] D. I. Ketcheson. Runge–Kutta methods with minimum storage implementations. *Journal of Computational Physics*, 229(5):1763 – 1773, 2010. URL: http://www.sciencedirect.com/science/article/pii/S0021999109006251, doi:https://doi.org/10.1016/j.jcp.2009.11.006[30].

[71] D. I. Ketcheson, K. Mandli, A. J. Ahmadia, A. Alghamdi, Manuel Quezada de Luna, M. Parsani, M. G. Knepley, and M. Emmett. PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. *SIAM Journal on Scientific Computing*, 34(4):C210–C231, January 2012. URL: http://epubs.siam.org/doi/10.1137/110856976, doi:10.1137/110856976[31].

[72] A. Knüpfer, C. Rössel, D. Mey an, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. Nagel E., Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, 79–91. Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[29] https://doi.org/10.1145/1089014.1089020
[30] https://doi.org/https://doi.org/10.1016/j.jcp.2009.11.006
[31] https://doi.org/10.1137/110856976

[73] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. Stanley Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, R. Stanley Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead. Technical Report, DARPA IPTO, 2008.

[74] B. Koren. A robust upwind discretization method for advection, diffusion and source terms. In C. B. Vreugdenhil and B. Koren, editors, *Numerical Methods for Advection-Diffusion Problems*, Notes on Numerical Fluid Mechanics, pages 117–138. Vieweg, Germany, 1993.

[75] L. Krenz, L. Rannabauer, and M. Bader. A High-Order Discontinuous Galerkin Solver with Dynamic Adaptive Mesh Refinement to Simulate Cloud Formation Processes. *arXiv:1905.05524 [physics]*, May 2019. arXiv: 1905.05524. URL: http://arxiv.org/abs/1905.05524.

[76] M. Kronbichler, T. Heister, and W. Bangerth. High accuracy mantle convection simulation through modern numerical methods: High accuracy mantle convection simulation. *Geophysical Journal International*, 191(1):12–29, October 2012. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2012.05609.x, doi:10.1111/j.1365-246X.2012.05609.x[32].

[77] J. Núñez-de la Rosa and C.-D. Munz. Hybrid DG/FV schemes for magnetohydrodynamics and relativistic hydrodynamics. *Computer Physics Communications*, 222:113–135, January 2018. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465517303338, doi:10.1016/j.cpc.2017.09.026[33].

[78] R. J. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002. doi:10.1017/CBO9780511791253[34].

[32] https://doi.org/10.1111/j.1365-246X.2012.05609.x
[33] https://doi.org/10.1016/j.cpc.2017.09.026
[34] https://doi.org/10.1017/CBO9780511791253

[79] LRZ. Hardware of supermuc-ng. 2019. [Online; accessed 06/05/2019]. URL: https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG.

[80] LRZ. Lrz: supermuc petascale system. https://www.lrz.de/services/compute/supermuc/systemdes (obtained 07/03/2019), 2019.

[81] C. Parés. Numerical methods for nonconservative hyperbolic systems: a theoretical framework. *SIAM Journal on Numerical Analysis*, 44(1):300–321, January 2006. URL: http://epubs.siam.org/doi/10.1137/050628052, doi:10.1137/050628052[35].

[82] W. H. Reed and T. R. Hill. *Triangular mesh methods for the neutron transport equation.* Oct 1973. URL: http://www.osti.gov/scitech/servlets/purl/4491151.

[83] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, Alice-Agnes Gabriel, Jean-Mathieu Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. ExaHyPE: An Engine for Parallel Dynamically Adaptive Simulations of Wave Problems. *arXiv:1905.07987 [cs, math]*, May 2019. arXiv: 1905.07987. URL: http://arxiv.org/abs/1905.07987.

[84] A. Reinarz, T. Dodwell, T. Fletcher, L. Seelinger, R. Butler, and R. Scheichl. Dune-composites – A new framework for high-performance finite element modelling of laminates. *Composite Structures*, 184:269 – 278, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0263822317321797, doi:https://doi.org/10.1016/j.compstruct.2017.09.104[36].

[85] S. Schoeder, K. Kormann, W. Wall, and M. Kronbichler. Efficient Explicit Time Stepping of High Order Discontinuous Galerkin Schemes for Waves. *arXiv:1805.03981 [cs]*, May 2018. URL: http://arxiv.org/abs/1805.03981.

[86] S. Schoeder, M. Kronbichler, and W. A. Wall. Arbitrary High-Order Explicit Hybridizable Discontinuous Galerkin Methods for the Acoustic Wave Equation. *Journal of Scientific Computing*, 76(2):969–1006, August 2018. URL: http://link.springer.com/10.1007/s10915-018-0649-2, doi:10.1007/s10915-018-0649-2[37].

[35] https://doi.org/10.1137/050628052
[36] https://doi.org/https://doi.org/10.1016/j.compstruct.2017.09.104
[37] https://doi.org/10.1007/s10915-018-0649-2

[87] S. Schoeder, W.A. Wall, and M. Kronbichler. ExWave: A high performance discontinuous Galerkin solver for the acoustic wave equation. *SoftwareX*, 9:49–54, January 2019. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352711018302024, doi:10.1016/j.softx.2019.01.001[38].

[88] C.-W. Shu and S. Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes, II. *Journal of Computational Physics*, 83(1):32 – 78, 1989. URL: http://www.sciencedirect.com/science/article/pii/0021999189902222, doi:https://doi.org/10.1016/0021-9991(89)90222-2[39].

[89] Erik Steiger. *Sampling methods with application in tsunami simulation*. PhD thesis, Technical University of Munich, September 2019.

[90] E. Strohmaier, J. J. Dongarra, H. Simon, and M. Meuer. Top500 - The List. [Online; accessed 27/05/2019]. URL: https://www.top500.org.

[91] P. Sweby. High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws. *SIAM Journal on Numerical Analysis*, 21(5):995–1011, 1984. URL: https://doi.org/10.1137/0721062, doi:10.1137/0721062[40].

[92] M. Tavelli, M. Dumbser, D. E. Charrier Charrier, L. Rannabauer, T. Weinzierl, and M. Bader. A simple diffuse interface approach on adaptive Cartesian grids for the linear elastic wave equations with complex topography. *Journal of Computational Physics*, 386:158 – 189, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0021999119300786, doi:https://doi.org/10.1016/j.jcp.2019.02.004[41].

[93] V. A. Titarev and E. F. Toro. ADER: Arbitrary high order Godunov approach. *Journal of Scientific Computing*, 17(1-4):609–618, 2002. URL: http://link.springer.com/article/10.1023/a:1015126814947.

[94] V. A. Titarev and E. F. Toro. ADER schemes for three-dimensional non-linear hyperbolic systems. *Journal of Computational Physics*, 204(2):715–736,

[38] https://doi.org/10.1016/j.softx.2019.01.001
[39] https://doi.org/https://doi.org/10.1016/0021-9991(89)90222-2
[40] https://doi.org/10.1137/0721062
[41] https://doi.org/https://doi.org/10.1016/j.jcp.2019.02.004

April 2005. URL: http://linkinghub.elsevier.com/retrieve/pii/S0021999104004358, doi:10.1016/j.jcp.2004.10.028[42].

[95] E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics: a practical introduction*. Springer, Dordrecht ; New York, 3rd ed edition, 2009. ISBN 978-3-540-25202-3 978-3-540-49834-6. OCLC: ocn401321914.

[96] K. Unterweger. *High-Performance Coupling of Dynamically Adaptive Grids and Hyperbolic Equation Systems (unpublished)*. PhD thesis, Technische Universität München, München, Germany, 2016.

[97] K. Unterweger, T. Weinzierl, D. Ketcheson, and A. Ahmadia. Peanoclaw - a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers. *Technische Universität München, Technical Reports*, 2013. URL: http://mediatum.ub.tum.de/attfile/1160344/hd2/incoming/2013-Jun/193139.pdf.

[98] B. van Leer. Towards the ultimate conservative difference scheme. V - A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32:101–136, July 1979. doi:10.1016/0021-9991(79)90145-1[43].

[99] B. van Leer. On the Relation Between the Upwind-Differencing Schemes of Godunov, Engquist–Osher and Roe. *SIAM Journal on Scientific and Statistical Computing*, 5(1):1–20, 1984. URL: https://doi.org/10.1137/0905001, doi:10.1137/0905001[44].

[100] J. VonNeumann and R. D. Richtmyer. A Method for the Numerical Calculation of Hydrodynamic Shocks. *Journal of Applied Physics*, 21(3):232–237, March 1950. URL: http://aip.scitation.org/doi/10.1063/1.1699639, doi:10.1063/1.1699639[45].

[101] T. Weinzierl. Peano. URL: http://www.peano-framework.org/index.php/peano-v-3/.

[102] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. PhD thesis, Technische Universität München, München, Germany, 2009.

[42] https://doi.org/10.1016/j.jcp.2004.10.028
[43] https://doi.org/10.1016/0021-9991(79)90145-1
[44] https://doi.org/10.1137/0905001
[45] https://doi.org/10.1063/1.1699639

[103] T. Weinzierl. The Peano Software—Parallel, Automaton-based, Dynamically Adaptive Grid Traversals. *ACM Transactions on Mathematical Software*, 45(2):1–41, April 2019. URL: http://dl.acm.org/citation.cfm?doid=3326465.3319797, doi:10.1145/3319797[46].

[104] T. Weinzierl, M. Bader, K. Unterweger, and R. Wittmann. Block Fusion on Dynamically Adaptive Spacetree Grids for Shallow Water Waves. *Parallel Processing Letters*, 24(03):1441006, September 2014. URL: http://www.worldscientific.com/doi/abs/10.1142/S0129626414410060, doi:10.1142/S0129626414410060[47].

[105] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM J. Sci. Comput.*, 33(5):2732–2760, 2011.

[106] T. Weinzierl and R. Wittmann. Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves. In *HiStencils*, 9. 2014.

[107] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computers in Physics*, 12(6):620, 1998. URL: http://scitation.aip.org/content/aip/journal/cip/12/6/10.1063/1.168744, doi:10.1063/1.168744[48].

[108] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *arXiv:1302.4280 [cs]*, February 2013. arXiv: 1302.4280. URL: http://arxiv.org/abs/1302.4280.

[109] O. Zanotti, F. Fambri, M. Dumbser, and A. Hidalgo. Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting. *Computers & Fluids*, 118():204 – 224, 2015. URL: http://www.sciencedirect.com/science/article/pii/S0045793015002030, doi:http://dx.doi.org/10.1016/j.compfluid.2015.06.020[49].

[110] A. Petitet, R. C. Whaley, J. J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. URL: http://www.netlib.org/benchmark/hpl/.

[46] https://doi.org/10.1145/3319797
[47] https://doi.org/10.1142/S0129626414410060
[48] https://doi.org/10.1063/1.168744
[49] https://doi.org/http://dx.doi.org/10.1016/j.compfluid.2015.06.020

[111] D. Calhoun. AMR Codes/Tools Overview. URL: https://math.boisestate.edu/~calhoun/www_personal/research/amr_software/.

[112] Intel. Intel® Trace Collector User and Reference Guide. URL: https://software.intel.com/en-us/itc-user-and-reference-guide.

[113] M. Kronbichler. ExWave. URL: https://github.com/kronbichler/exwave.

[114] O. Peckham. EuroHPC Summit Week Highlights Europe's Plan for Supercomputing Leadership. URL: https://www.hpcwire.com/2019/05/28/eurohpc-summit-week-highlights-europes-plan-for-supercomputing-leadership/.

[115] ORNL. Frontier - Direction of Discovery. URL: https://www.cray.com/company/news-and-media/doe-frontier-press-release.

[116] S. Moss. The race to exascale: A story of superpowers and supercomputers. March 2019. URL: https://www.datacenterdynamics.com/analysis/superpowers-supercomputers-and-race-exascale/.